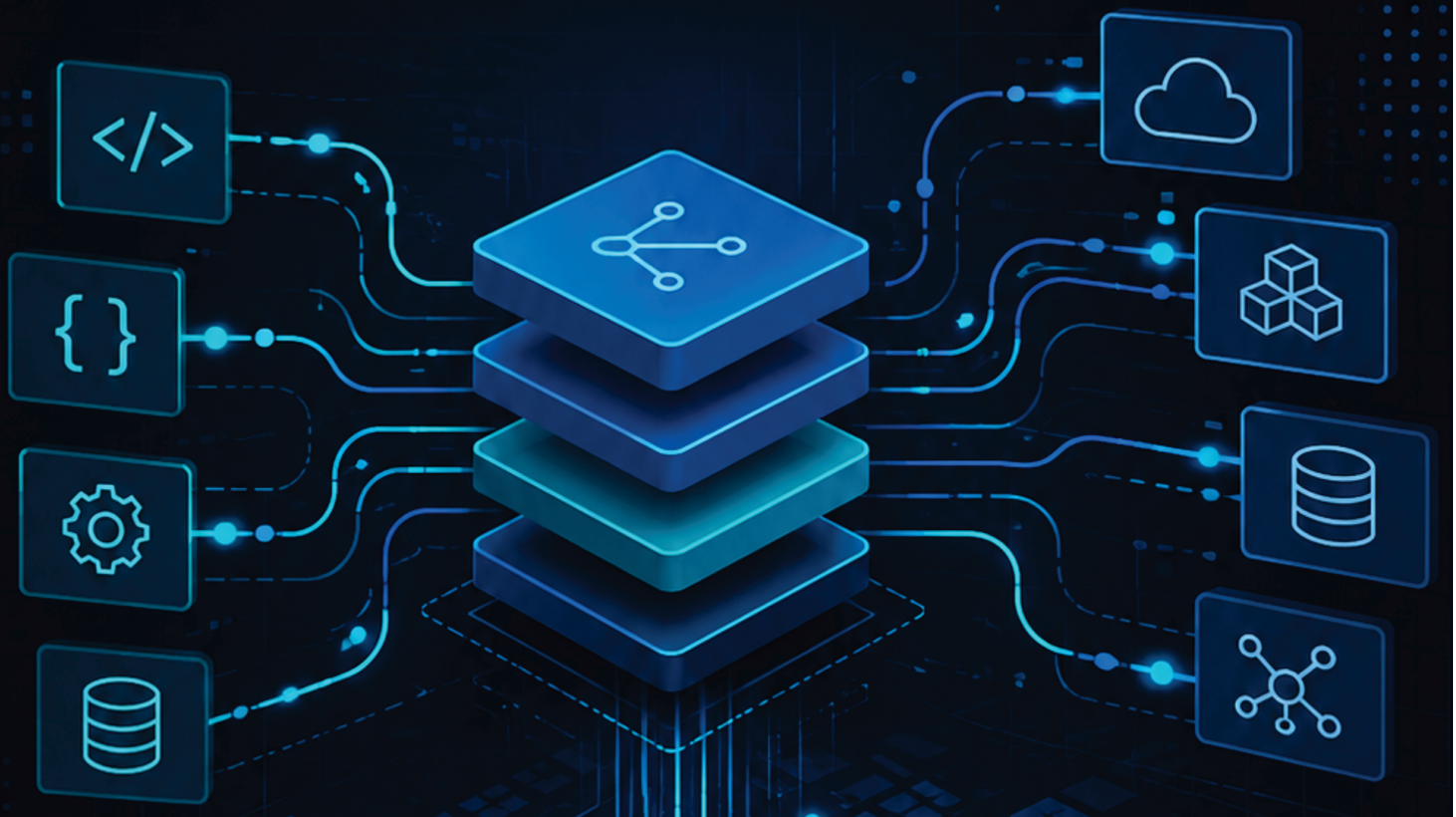
Paradigme de programare și arhitecturi software de integrare

Eduard-Cristian Popovici



MATRIX ROM
BUCUREȘTI

Eduard-Cristian POPOVICI

Paradigme de programare și arhitecturi software de integrare

MATRIX ROM
BUCUREȘTI, 2026

MATRIX ROM
Str. Politehnicii nr.3
060811 – BUCUREŞTI
tel. 021.4113617, 0733882137
email: office@matrixrom.ro
www.matrixrom.ro

Editura MATRIX ROM este acreditată de
CONSILIUL NAŢIONAL AL CERCETĂRII ŞTIINŢIFICE DIN ÎNVĂŢĂMÂNTUL SUPERIOR

Sursă copertă: www.chatgpt.com

ISBN: 978-606-25-1070-1

Prefață

Lucrarea *Paradigme de programare și arhitecturi software de integrare* propune o abordare sistematică a unor concepte utile pentru înțelegerea și dezvoltarea aplicațiilor software moderne. Într-un domeniu aflat într-o continuă schimbare, în care apar frecvent noi limbaje, tehnologii, platforme și metode de lucru, cunoașterea principiilor fundamentale rămâne indispensabilă.

Cartea urmărește să ofere cititorului o perspectivă coerentă asupra modului în care paradigmele de programare, modelele arhitecturale și tehnologiile de integrare contribuie la proiectarea sistemelor software. Accentul este pus pe înțelegerea relației dintre conceptele teoretice și aplicarea lor în contexte practice, astfel încât cititorul să poată evalua mai clar opțiunile disponibile în procesul de dezvoltare software.

Volumul de față este conceput ca parte a unei serii mai ample dedicate ingineriei software. În continuarea acestuia, sunt avute în vedere lucrări care vor aborda teme precum modelarea sistemelor software, procesele moderne de dezvoltare, securizarea aplicațiilor, serviciilor și sistemelor software, precum și impactul inteligenței artificiale asupra dezvoltării software.

Prin tematica abordată, lucrarea se adresează studenților, dezvoltatorilor software și tuturor celor interesați de proiectarea, organizarea și integrarea sistemelor informatice. Ea poate constitui atât un suport de studiu, cât și un punct de plecare pentru aprofundarea unor direcții actuale din ingineria software, oferind în același timp nivelul minim necesar pentru înțelegerea tehnologiilor și conceptelor mai vechi, care continuă să stea la baza soluțiilor moderne.

Autorul, București, 2026

Cuprins

Introducere

Software-ul începe, de multe ori, cu o problemă simplă și cu câteva linii de cod. Scriem o funcție, verificăm o condiție, repetăm un pas, calculăm un rezultat. Pentru programe mici, acest mod de lucru este suficient. Pe măsură ce aplicația crește, apar însă alte întrebări: unde punem datele, cine are voie să le modifice, cum împărțim codul în părți clare, cum testăm o schimbare, cum evităm ca o modificare locală să strice zone aparent fără legătură? Răspunsurile la aceste întrebări au dus, în timp, la **paradigme și arhitecturi software**.

O **paradigmă de programare** oferă un mod de a gândi codul. Programarea structurată pune accent pe pași clari, funcții și controlul fluxului de execuție. Orientarea spre obiecte grupează datele și comportamentul în clase și obiecte. Programarea funcțională adaugă perspectiva funcțiilor pure, a imutabilității și a reducerii efectelor laterale, idei care au influențat puternic limbajele moderne. Orientarea spre componente tratează anumite părți ale sistemului ca unități livrabile, cu interfețe și versiuni. Orientarea spre servicii mută această idee la nivel de procese și aplicații care comunică prin contracte de rețea, mesaje sau evenimente.

O **arhitectură software** privește sistemul de sus. Ea arată care sunt părțile importante, ce responsabilitate are fiecare parte și cum comunică între ele. Paradigma ne ajută să scriem codul în interiorul unei părți, pe când arhitectura ne ajută să organizăm întregul. Cele două sunt legate. Un sistem distribuit nu devine clar doar pentru că este împărțit în servicii. Un serviciu prost modularizat intern nu elimină problema, ci o mută în alt loc. La fel, un *framework* modern nu poate compensa lipsa unor contracte clare între module.

Cartea urmărește această **evoluție** pas cu pas. Începem cu **programarea structurată**, unde funcțiile pot fi privite ca module de comportament, iar structurile de date ca module de date. Un fișier antet (*header*) din C poate fi deja văzut ca un contract: spune ce operații sunt disponibile, fără să expună toate detaliile interne. De aici se poate înțelege mai bine de ce variabilele globale, deși par comode la început, ajung să creeze dependențe ascunse și schimbări greu de controlat.

Apoi trecem la **orientarea spre obiecte**. O clasă reunește date și operații, ascunde starea internă și expune metode prin care obiectul poate fi folosit controlat. Principiile OO, cum ar fi responsabilitatea unică, programarea spre interfețe, substituția Liskov, inversiunea dependențelor, coeziunea ridicată și cuplarea redusă, nu sunt reguli decorative. Ele sunt moduri practice de a păstra codul modificabil și testabil. Un programator real le simte valoarea mai ales atunci când trebuie să schimbe ceva fără să strice restul aplicației.

În această zonă este amintită și influența **programării funcționale** asupra limbajelor moderne. Chiar dacă firul principal al cărții rămâne orientat spre modularizare, obiecte, componente și servicii, concepte precum funcțiile pure, imutabilitatea, expresiile lambda și transformările declarative asupra colecțiilor contribuie la același scop: reducerea stării partajate și obținerea unui cod mai previzibil, mai ușor de testat și mai sigur în contexte concurente sau asincrone.

***Pattern*-urile de proiectare** duc discuția mai departe. Ele dau nume unor soluții care apar des. *Adapter* ajută două interfețe diferite să colaboreze. *Proxy* controlează accesul la un obiect fără să schimbe interfața acestuia. *Observer* permite notificarea mai multor părți interesate atunci când apare un eveniment. *Command* transformă o acțiune într-un obiect care poate fi executat, memorat, pus într-o coadă sau retransmis. *Singleton* arată atât utilitatea controlului unei instanțe unice, cât și riscul dependențelor globale ascunse.

După obiecte și *pattern*-uri, apare **orientarea spre componente**. O componentă are un contract public, o implementare ascunsă și o versiune. Ea poate fi testată, distribuită și înlocuită mai ușor decât o colecție întâmplătoare de clase. *JavaBeans* și EJB (*Enterprise JavaBeans*) sunt exemple istorice utile, chiar dacă astăzi multe aplicații folosesc *framework*-uri mai moderne. Ele au introdus sau popularizat idei importante: proprietăți, evenimente, introspecție, cicluri de viață și containere care gestionează componente.

***Framework*-urile** și inversarea controlului (IoC, *Inversion of Control*) schimbă apoi direcția controlului. În loc ca aplicația să creeze manual toate obiectele și să le lege între ele, *framework*-ul oferă structura principală și cheamă codul aplicației în punctele potrivite. Injectarea dependențelor (DI, *Dependency Injection*) face dependențele explicite și permite înlocuirea implementărilor, ceea ce ajută atât testarea, cât și evoluția sistemului.

Modelul **MVC** (*Model–View–Controller*) și variantele sale arată cum separăm datele, prezentarea și controlul interacțiunilor. Aceeași idee apare în aplicații desktop, aplicații web, API-uri (*Application Programming Interface*, interfețe de programare a aplicațiilor) și *frontend*-uri moderne. Nu este important doar numele *pattern*-ului, ci disciplina de a nu amesteca toate responsabilitățile în același loc.

În final, cartea ajunge la **arhitecturi software de integrare, servicii, mesagerie, evenimente, containerizare și cloud**. Aici modularitatea nu mai este doar o problemă de fișiere, clase sau pachete. Ea devine o problemă de procese, protocoale, baze de date, contracte de rețea, mesaje, evenimente, versiuni, monitorizare, diagnostic și operare. SOA, microserviciile, REST, mesageria, arhitecturile orientate spre evenimente, containerizarea și cloudul sunt răspunsuri diferite la aceeași problemă: împărțirea unui sistem mare în părți suficient de independente, dar capabile să colaboreze.

Ideea centrală a cărții este că **paradigmele nu se anulează între ele**. Programarea structurată rămâne importantă și în codul OO. Principiile OO rămân importante în componente și *framework*-uri. Conceptele funcționale, precum imutabilitatea și reducerea efectelor laterale, completează proiectarea orientată spre obiecte. Componentele rămân importante și în servicii. Iar un microserviciu bun are nevoie, în interior, de aceleași lucruri simple: funcții clare, date bine încapsulate, interfețe stabile, teste utile și responsabilități bine separate.

De aceea, **exemplele din carte** pornesc **de la cod simplu** și **cresc treptat**. Mai întâi funcții și structuri C, apoi clase și interfețe Java, apoi *pattern*-uri, componente, *framework*-uri, MVC, servicii, mesagerie, evenimente și cloud. Studiile de caz reiau aceeași funcționalitate în mai multe forme arhitecturale, de la monolit stratificat la monolit modular, servicii REST și coregrafie bazată pe evenimente. Scopul nu este de a acoperi exhaustiv fiecare tehnologie, ci de a vedea firul comun: controlul complexității prin modularitate, contracte clare, separarea responsabilităților și alegerea atentă a granițelor. Această perspectivă permite cititorului să privească tehnologiile nu ca soluții izolate, ci ca răspunsuri diferite la aceeași problemă fundamentală: organizarea schimbării într-un sistem software.

Materialul a fost elaborat printr-un proces de selecție, structurare și revizuire critică asumat de autor, cu utilizarea unor instrumente digitale de sprijin, menționate distinct în nota privind utilizarea instrumentelor de inteligență artificială.

Capitolul 1. De la programare structurată la servicii

Urmând principiile ingineriei software, o disciplină al cărei scop este să controleze **complexitatea** software-ului (programelor de calculator), acest capitol trece în revistă, în ordinea apariției istorice, principalele abordări ale programării: de la programarea structurată, la orientarea spre obiecte și principiile ei, *pattern*-urile OO, apoi componentele, *framework*-urile și principiile lor, pentru a ajunge, în final, la orientarea spre servicii. Capitolul urmărește aceste abordări ca straturi succesive de proiectare, nu ca înlocuiri definitive.

1.1. Programarea structurată

Programarea structurată este primul mare pas sistematic pentru a ține sub control complexitatea software-ului. Ideea-cheie este să construim programe din **blocuri** de comportament (**secvență**, **decizie** și **iterație**) evitând salturile arbitrare de execuție și separând clar **datele** de **operații** [1]. În practică, ea rămâne relevantă și azi: chiar dacă folosim orientarea spre obiecte, componente sau servicii, gândirea procedurală curată stă la baza oricărui flux corect și lizibil.

1.1.1. Elementele fundamentale: secvență, decizie, iterație

Un program structurat este o **compoziție de blocuri** care au un singur punct de intrare și unul de ieșire.

- **secvența**: instrucțiunile se execută pe rând, în ordinea în care apar;
- **decizia**: ramificarea controlului, adică programul alege o ramură în funcție de o condiție, de exemplu `if`, `else`;
- **iterația**: repetarea controlată, adică repetarea unui pas până când o condiție devine falsă, de exemplu `for`, `while`.

Aceste trei structuri sunt suficiente, din punct de vedere teoretic, pentru a descrie orice algoritm calculabil, rezultat asociat clasic cu teorema Böhm-Jacopini [2]. În practică, ele fac codul **previzibil**, deoarece pentru fiecare buclă se poate enunța o propoziție care rămâne adevărată pe tot parcursul ei, numită **invariant**. De exemplu, într-o buclă care adună elementele unui tablou, invariant este faptul că la începutul fiecărei iterații suma reține totalul elementelor deja parcurse. Astfel, se poate verifica ușor corectitudinea pașilor și cititorul codului poate urmări logica acestuia.

1.1.2. Modularizare și ascunderea informației

Pe măsură ce aplicațiile cresc, simpla existență a unor funcții bine scrise nu mai este suficientă. Devine necesară împărțirea programului în **module** cu scop clar și interfețe stabile. Fiecărui modul i se atașează un contract ușor de înțeles, iar detaliile interne rămân ascunse. Această abordare poartă numele de ascunderea informației (***information hiding***) și presupune expunerea doar a elementelor necesare utilizării, cu păstrarea implementării în spatele interfeței [3].

În programarea structurată, modularizarea poate fi înțeleasă prin două forme simple și foarte practice. Funcțiile pot fi privite ca **module de comportament**, deoarece grupează pași de execuție sub un nume clar și oferă o operație reutilizabilă. Structurile de date pot fi privite ca **module de date**, deoarece grupează valori care aparțin aceluiași concept. Împreună, funcțiile și structurile formează baza organizării programelor procedurale.

Un **exemplu minimal** (în limbajul C) este un cont bancar simplificat. Structura grupează datele contului, iar funcțiile exprimă operațiile permise asupra acestuia.

```c
#include <stdio.h>

typedef struct {
    int id;
    double balance;
} Account;

void deposit(Account* account, double amount) {
    if (!account || amount <= 0) {
        return;
    }
    account->balance += amount;
}

int withdraw(Account* account, double amount) {
    if (!account || amount <= 0) {
        return 0;
    }
    if (account->balance < amount) {
        return 0;
    }
    account->balance -= amount;
    return 1;
}

int main(void) {
    Account account = { 1, 100.0 };
    deposit(&account, 50.0);
    if (withdraw(&account, 30.0)) {
        printf("Balance: %.2f\n", account.balance);
    }
    return 0;
}
```

Exemplul este simplu, dar conține deja ideea de bază. `Account` este un **modul de date**, deoarece regrupează informațiile relevante despre cont. Funcțiile `deposit()` și `withdraw()` sunt **module de comportament**, deoarece definesc operații cu reguli proprii. Totuși, în această formă, toate detaliile sunt vizibile în același fișier. Orice cod poate accesa direct câmpul `balance`. Pentru programe mici acest lucru este acceptabil, dar în sisteme mai mari produce dependențe fragile.

Un pas mai bun este **separarea interfeței de implementare**.

Fișierul antet expune contractul.

```c
 /* account.h */
#ifndef ACCOUNT_H
#define ACCOUNT_H

typedef struct Account Account;

Account account_create(int id, double initial_balance);
int account_deposit(Account* account, double amount);
int account_withdraw(Account* account, double amount);
double account_balance(const Account* account);

#endif
```

Iar **fișierul sursă ascunde detaliile** de implementare.

```c
/* account.c */
#include "account.h"

struct Account {
    int id;
    double balance;
};
Account account_create(int id, double initial_balance) {
    Account account;
    account.id = id;
    account.balance = initial_balance >= 0 ? initial_balance : 0;
    return account;
}
int account_deposit(Account* account, double amount) {
    if (!account || amount <= 0) {
        return 0;
    }
    account->balance += amount;
    return 1;
}
int account_withdraw(Account* account, double amount) {
    if (!account || amount <= 0) {
        return 0;
    }
    if (account->balance < amount) {
        return 0;
    }
    account->balance -= amount;
    return 1;
}
double account_balance(const Account* account) {
    if (!account) {
        return 0;
    }
    return account->balance;
}
```

În această variantă, clientul nu mai utilizează direct câmpurile structurii. El folosește operațiile publice din antet. Implementarea poate schimba numele câmpurilor, regulile interne sau modul de reprezentare a soldului, fără ca `main.c` să fie modificat. Aceasta este forma procedurală a principiului ascunderii informației: interfața rămâne stabilă, implementarea poate evolua [3].

Contractul descrie modul de utilizare, nu implementarea. În mod uzual, el conține [4]:

- **operațiile** oferite (expuse), adică numele și parametrii funcțiilor sau metodelor;
- **precondiții**, ce trebuie să fie adevărat la intrare, de ex. argumente nenule sau intervale valide („`argumentul nu este null`”, „`indexul se afla in intervalul valid`” etc.);
- **postcondiții**, ce se garantează la ieșire, de ex. proprietăți ale rezultatului sau efecte persistente („`rezultatul este sortat`”, „`valoarea a fost salvata`” etc.);
- modalitatea de semnalare a erorilor, de ex. coduri de eroare, excepții sau valori speciale.

Atunci când restul aplicației interacționează doar cu contractul, iar datele și algoritmii rămân izolați în modul, apar **avantaje** clare:

- **modificările** de implementare rămân locale, de exemplu înlocuirea unei structuri de date cu alta fără impact extern;
- **testarea** devine mai simplă, atenția concentrându-se pe verificarea contractului;
- **colaborarea** în și între echipe se realizează mai ușor, granițele fiind stabilite prin interfețe.

După exemplul anterior, în care un modul C a fost separat în interfață și implementare, următorul exemplu mai mic arată **cum poate fi exprimat explicit contractul unei funcții** prin **precondiții**, **postcondiții** și **verificări** simple.

```
#include <assert.h>

// preconditii: a != NULL, n >= 0
// postconditie: rezultatul este suma elementelor a[0..n-1]
int sum_array(const int* a, int n) {

// guard: verificare rapida a preconditiei la intrare,
// daca nu este indeplinita se intoarce imediat un rezultat minim
if (!a) return 0;

    int s = 0;
    for (int i = 0; i < n; i++) {
        s += a[i];
    }
    // verificare simpla a postconditiei pe cazul n == 0
    // (corectitudinea generala rezulta din logica buclei)
    if (n == 0) { assert(s == 0); }
    return s;
}
```

Funcția **`sum_array()`** este tratată ca un **modul** cu scop clar, calculează suma unui tablou, iar detaliile interne rămân ascunse în corpul funcției, apelantul vede doar semnătura (numele funcției, lista de argumente cu tipurile lor și tipul valorii returnate, de exemplu `(const int* a, int n) -> int`) și regulile de utilizare, ceea ce ilustrează modularizarea și ascunderea informației.

Contractul este exprimat prin **precondiție** și **postcondiție**, precondiția arată cum se apelează corect funcția, iar postcondiția enunță ce se garantează la ieșire, astfel interacțiunea cu funcția devine previzibilă și testabilă fără a depinde de implementare [4]. Un „*guard*" este o verificare rapidă la începutul funcției, care confirmă precondițiile sau tratează cazul invalid imediat.

În contrast cu principiile contractului și ale ascunderii informației, practica „variabile globale peste tot" este un ***anti-pattern*** (o soluție aparent comodă, dar care produce probleme recurente). La început pare simplu ca mai multe funcții să folosească aceeași stare globală. Pe măsură ce programul crește, această stare comună devine greu de urmărit.

```
#include <stdio.h>
double global_balance = 100.0;

void deposit(double amount) {
    if (amount > 0)
        global_balance += amount;
}
void apply_fee(void) {
    global_balance -= 5.0;
}
void print_balance(void) {
    printf("Balance: %.2f\n", global_balance);
}
int main(void) {
    deposit(50.0);
    apply_fee();
    print_balance();
    return 0;
}
```

Problema nu este doar existența variabilei globale, ci faptul că **orice** funcție o poate citi sau modifica. Dacă ulterior decidem că soldul trebuie reprezentat printr-o structură mai bogată, de exemplu pentru a include moneda, limita de credit sau starea contului, modificarea nu mai este locală.

```
typedef struct {
    double value;
    char currency[4];
} Money;

/* obliga modificarea functiilor care foloseau global_balance */
Money global_balance;
```

Într-un **program real**, efectul este și mai dificil de controlat. Unele funcții citesc valoarea, altele o modifică, altele presupun o anumită ordine de inițializare. Depanarea devine grea, deoarece nu mai este clar cine a schimbat starea și când. O soluție mai sănătoasă este trecerea stării ca parametru explicit, gruparea datelor într-o structură și expunerea operațiilor printr-un contract clar. Astfel, dependențele sunt vizibile în semnăturile funcțiilor, iar schimbările rămân mai ușor de localizat.

Această trecere poate fi privită ca o **refactorizare** simplă. Refactorizarea înseamnă **reorganizarea internă a codului fără schimbarea comportamentului observabil din exterior**. În exemplul de mai sus, programul continuă să lucreze cu un sold și cu operații de depunere sau retragere, dar dependențele devin explicite, iar starea nu mai este ascunsă într-o variabilă globală. Scopul refactorizării este să reducă fragilitatea codului și să pregătească schimbări viitoare fără a altera funcționalitatea existentă.

1.1.3. Impactul modularizării structurate pentru depanare și testare

Erorile apar inevitabil, diferența o face locul și felul în care sunt descoperite și semnalate. O practică simplă este validarea la interfața modulului, adică verificarea datelor de intrare imediat ce intră în modul, urmată de un mecanism consecvent de raportare a erorilor. În limbajul C, de exemplu, funcțiile pot întoarce un cod numeric clar, iar textul de eroare se poate trimite în *log* (fișier sau sistem de înregistrare a mesajelor pentru diagnostic și audit). Astfel testele pot verifica exact acea valoare fără să depindă de detalii interne, iar **depanarea** devine mai ușoară deoarece locul unde s-a respins intrarea este bine delimitat.

În **testare**, scenariile se formulează pe contract, nu pe implementare. Mai întâi se verifică intrări invalide, apoi intrări valide și rezultatele promise, inclusiv valorile de eroare documentate. Această abordare reduce dependența de detalii interne și face suitele de test mai stabile în timp [5]. În același timp, separarea prin interfețe le permite echipelor să lucreze în paralel, iar modulul testat poate evolua intern fără a se schimba felul în care este folosit, atâta timp cât semnătura și regulile rămân aceleași. De aceea, exemplul anterior cu `Account` poate fi testat fără acces direct la câmpurile structurii, folosind doar funcțiile publice din `account.h`.

În limbajele care oferă **excepții**, acestea pot separa fluxul normal de cel de eroare. Ideea rămâne aceeași, validarea la interfața modulului respinge devreme intrările incorecte. Un mecanism consecvent de semnalare permite teste precise pe cazuri negative și pe cazuri limită (de exemplu valori minime sau maxime, liste goale, valori zero, praguri). Iar mesajele din jurnal pot include codul sau tipul de eroare și contextul, ceea ce face reproducerea problemelor mai directă.

Un **test unitar** verifică o unitate mică de cod, de obicei o funcție sau un modul, în izolare față de restul aplicației [5]. În C, o formă minimală de testare unitară poate fi construită cu **`assert()`**, fără un *framework* suplimentar; macro-ul este definit în antetul standard `<assert.h>` și, când aserțiunea este activă, semnalează încălcarea condiției verificate [6]. Scopul nu este să obținem o infrastructură completă de testare, ci să ilustrăm ideea: contractul modulului se verifică prin cazuri concrete.

```
#include <assert.h>
#include "account.h"

void test_create_account(void) {
    Account account = account_create(1, 100.0);
    assert(account_balance(&account) == 100.0);
}

void test_deposit_positive_amount(void) {
    Account account = account_create(1, 100.0);
    int result = account_deposit(&account, 50.0);
    assert(result == 1);
    assert(account_balance(&account) == 150.0);
}

void test_deposit_negative_amount_is_rejected(void) {
    Account account = account_create(1, 100.0);
    int result = account_deposit(&account, -10.0);
    assert(result == 0);
    assert(account_balance(&account) == 100.0);
}

void test_withdraw_too_much_is_rejected(void) {
    Account account = account_create(1, 100.0);
    int result = account_withdraw(&account, 150.0);
    assert(result == 0);
    assert(account_balance(&account) == 100.0);
}

int main(void) {
    test_create_account();
    test_deposit_positive_amount();
    test_deposit_negative_amount_is_rejected();
    test_withdraw_too_much_is_rejected();
    return 0;
}
```

Testele nu verifică modul intern în care este reprezentat contul, ci **efectele observabile** prin interfața publică. Implementarea internă poate evolua fără să invalideze testele, atâta timp cât contractul rămâne același. Aceasta este o consecință practică a modularizării: testarea urmărește comportamentul promis, nu detaliile ascunse. Pentru proiecte mai mari, se folosesc *framework*-uri dedicate de testare, rapoarte automate și integrare în procesul de build. Important este principiul: se testează comportamentul expus prin contract, iar detaliile interne rămân libere să evolueze.

1.2. Orientarea spre obiecte (OO)

După ce programarea structurată a adus claritate prin blocuri, iar modularizarea și ascunderea informației au stabilit granițe între părți, orientarea spre obiecte (OO - *object orientation*) le-a combinat într-un model în care tipurile definesc **împreună** datele și operațiile. O clasă descrie structura și comportamentul, un obiect este o instanță concretă, încapsularea continuă ideea de ascundere a detaliilor, iar moștenirea și polimorfismul adaugă mecanisme de reutilizare și extensibilitate [7]. Efectul practic este că regulile de utilizare devin interfețe stabile, implementările pot evolua separat, iar testarea și depanarea se sprijină pe contracte explicite.

1.2.1. Obiecte, clase, abstractizare, încapsulare, moștenire, polimorfism

În **abordarea orientată spre obiecte** (*object-oriented*, OO) se pornește de la ideea că lumea este percepută ca un ansamblu de entități care au stare și se manifestă prin comportamente. Un **obiect** din software **modelează** o astfel de entitate concretă, are identitate proprie (variabila obiect), deține date interne și oferă operații care pot fi invocate. Când un `comutator` este acționat, nu se vede o funcție, se vede un obiect asupra căruia alt obiect invocă o operație, de exemplu `comuta()`, iar schimbarea stării apare ca efect al acelei invocări. În program, obiectul reține starea relevantă și expune numai acțiunile permise, astfel modelul rămâne apropiat de realitate și inteligibil pentru cititorul codului.

Clasele abstractizează obiectele de aceeași **categorie**, ele sunt tipuri complexe care descriu atât structura datelor, cât și operațiile permise. O clasă poate fi privită simultan ca **tipar** pentru instanțe și ca **mulțime** de obiecte compatibile din punctul de vedere al semnăturilor de operații. Clasa `Comutator`, de exemplu, precizează că orice comutator are o stare și știe să pornească, să oprească și să comute. Obiectele create după această clasă au identități distincte, dar respectă aceleași reguli de utilizare stabilite de semnăturile publice.

Abstractizarea descrisă mai sus operează pe două niveluri care se leagă logic. Mai întâi, prin **modelare**, se selectează din lumea reală numai detaliile relevante pentru scopul aplicației, restul se ignoră. Apoi, prin clase, acele detalii sunt grupate în **categorii** generale care pot fi refolosite. Pentru comutator, la nivelul obiectului se păstrează starea și acțiunile de bază, la nivelul clasei se fixează contractul public care se aplică tuturor comutatoarelor. O abstractizare bună produce modele stabile la schimbări locale și ușor de extins, pentru că separă esențialul de accidental și fixează clar ce rămâne adevărat indiferent de implementare.

Încapsularea este forma specifică OO de modularizare și ascundere a informației, ea regrupează datele interne ascunse și operațiile care le manipulează, expunând o interfață publică prin semnături clare [3]. Accesul direct la starea obiectului este limitat, interacțiunea se face prin operații care aplică reguli și validări la interfața obiectului, exact cum în programarea structurată au fost tratate funcțiile și contractele. Avantajul este că detaliile interne se pot schimba fără impact asupra codului client, iar testarea se poate focaliza pe efectele observabile ale operațiilor, nu pe mecanismele interne.

Exemplul procedural anterior cu `Account` poate fi rescris în limbajul Java ca o clasă care reunește datele și operațiile. În loc ca structura să fie separată de funcțiile care o manipulează, clasa păstrează starea internă și expune metode publice controlate.

```
public final class Account {
    private final int id;
```

```
    private double balance;

    public Account(int id, double initialBalance) {
        this.id = id;
        this.balance = Math.max(0, initialBalance);
    }

    public int id() {
        return id;
    }

    public double balance() {
        return balance;
    }

    public boolean deposit(double amount) {
        if (amount <= 0) {
            return false;
        }
        balance += amount;
        return true;
    }

    public boolean withdraw(double amount) {
        if (amount <= 0 || amount > balance) {
            return false;
        }
        balance -= amount;
        return true;
    }
}
```

Utilizarea clasei devine simplă:

```
public class Demo {
    public static void main(String[] args) {
        Account account = new Account(1, 100.0);
        account.deposit(50.0);
        account.withdraw(30.0);
        System.out.println(account.balance());
    }
}
```

Față de varianta procedurală, diferența principală este locul în care se află regula. În C, funcțiile primeau explicit un *pointer* către structura `Account`. În Java, obiectul `account` conține starea, iar operațiile sunt apelate asupra lui. Câmpul `balance` este privat, deci nu poate fi modificat direct din exterior. Orice schimbare trece prin metodele `deposit()` și `withdraw()`, unde se pot aplica validări. Aceasta este încapsularea în forma ei practică: datele interne sunt protejate, iar codul client lucrează printr-un contract public.

Moștenirea introduce un nou nivel de **abstractizare între clase**, o clasă **generală** captează ceea ce este **comun**, **superclasă**, iar clasele **specializate**, **subclase**, adaugă sau particularizează comportamente. Cu cât o clasă este mai abstractă, cu atât este mai general valabilă și conține mai puține detalii. Cu cât clasa este mai specializată, cu atât este mai apropiată de lumea modelată și include mai multe detalii. Comutatorul inteligent poate deriva din comutatorul de bază și poate adăuga temporizări sau raportare, menținând contractul public. Moștenirea este potrivită când relația este de tip "**este un**". Altfel, se preferă compunerea pentru a evita ierarhii rigide și cuplare inutilă (conceptul va fi detaliat mai jos).

Polimorfismul permite folosirea aceleiași interfețe pentru obiecte de clase diferite, decizia privind comportamentul concret se face la rulare. Codul poate manipula o colecție de comutatoare obișnuite și inteligente prin același set de operații publice, de exemplu comută, fără să cunoască tipul exact al fiecăruia. În practică apar două forme complementare, **rescrierea metodelor** în subclase, adică adaptarea unei operații moștenite la un nou comportament, și **supraîncărcarea numelor metodelor**, adică mai multe operații cu același nume dar semnături diferite în aceeași clasă, utile pentru variații de parametri. Polimorfismul reduce dependența de implementări concrete și facilitează testarea, aceeași suită de teste se poate rula împotriva unor obiecte diferite care respectă aceeași interfață.

1.2.2. Principii fundamentale ale orientării spre obiecte

De-a lungul timpului, din bunele practici au rezultat o serie de principii, care oferă reguli și ghidează proiectarea pentru a obține controlul complexității, reducerea cuplării și creșterea testabilității, astfel încât implementările pot evolua fără a afecta codul client, iar echipele pot lucra în paralel pe granițe clare [8]. Ele răspund la câteva întrebări practice: cine are responsabilitatea, de ce depinde codul, cât de ușor poate fi schimbat și cât de ușor poate fi testat?

Principiul responsabilității unice (**SRP**, *Single Responsibility Principle*) afirmă că o clasă ar trebui să aibă un singur motiv clar de schimbare, adică o singură responsabilitate, pentru un mai bun control al complexității. Când o clasă acumulează roluri multiple, o modificare pentru un rol afectează neintenționat celelalte. Se aplică la proiectarea tipurilor de bază, de exemplu un tip care modelează comutatorul gestionează exclusiv starea și operațiile lui, jurnalizarea sau raportarea aparțin altor tipuri, astfel erorile se izolează, iar refactorizările rămân locale.

Principiul programării spre interfețe, nu implementări (**P2I**, *Program to an Interface*) recomandă legarea dependențelor la contractul public, nu la o clasă concretă, pentru flexibilitate și testabilitate mai bune. Un controler care depinde de o interfață `Comutator` poate colabora cu un comutator obișnuit sau inteligent fără să cunoască detalii interne. Se aplică la granițele dintre module: codurile apelantului cunosc operațiile, codurile acestor operații pot evolua independent, iar în testare se pot introduce dubluri ale operațiilor fără a modifica logica apelantului.

Principiul substituției Liskov (LSP, *Liskov Substitution Principle*) afirmă că orice instanță a unei subclase trebuie să poată înlocui o instanță a superclasei fără a surprinde apelantul, pentru o mai bună predictibilitate [9]. Așteptările stabilite de tipul de bază trebuie păstrate, precondițiile nu pot fi întărite, postcondițiile nu pot fi slăbite. Se aplică atunci când se introduc specializări, de exemplu comutatorul inteligent trebuie să rămână utilizabil oriunde este așteptat un comutator, altfel apar erori subtile și costisitoare.

Principiul fără schimbare acolo unde se poate (**IMM**, *Immutability*) recomandă ca starea să fie stabilită la creare și să nu se mai schimbe ulterior, pentru simplitate și siguranță. Aceleași intrări produc aceleași ieșiri, iar partajarea în contexte concurente devine sigură. Se aplică pentru valori și configurări, de exemplu parametrii care descriu caracteristicile unui comutator rămân neschimbați, operațiile produc obiecte noi sau efecte controlate, ceea ce ușurează testarea și depanarea.

Principiul semnalării timpurii a erorilor și validărilor la interfața obiectului (**FF**, *Fail Fast*) recomandă semnalarea timpurie a erorilor chiar la intrarea în operații, pentru reducerea costului de diagnostic. Datele invalide sunt respinse imediat, locul problemei rămâne bine delimitat. Se aplică în special în metodele publice, unde argumentele se verifică explicit la început, iar erorile se semnalează consecvent, fie prin cod numeric, fie prin excepție documentată. Astfel, testele se concentrează pe cazuri negative și pe cazuri limită, iar timpul de diagnostic scade.

Principiul coeziune ridicată și cuplare redusă (**HCLC**, *High Cohesion, Low Coupling*) afirmă că o clasă ar trebui să grupeze date și operații care aparțin aceluiași scop, iar dependențele către alte clase să fie puține și bine motivate, pentru o evoluție sănătoasă a sistemului. Când granițele sunt clare, înlocuirile devin simple și echipele lucrează în paralel fără interferențe. Se aplică la definirea pachetelor și a colaborărilor, de exemplu comutatorul colaborează cu un serviciu de raportare printr-o interfață, care nu depinde de implementarea sa, astfel că schimbările rămân locale și previzibile.

Principiul deschis la extindere, închis la modificare (**OCP**, *Open–Closed Principle*) afirmă că tipurile ar trebui să fie deschise pentru extindere și închise pentru modificare, pentru a permite adăugarea de comportamente noi fără a risca stricarea codului stabil. Se aplică prin introducerea de interfețe și puncte de extensie, de exemplu o clasă `Comutator` poate fi extinsă printr-un nou tip de strategie de temporizare, fără a modifica tipul de bază, ci doar adăugând o implementare nouă, care poate fi conectată sau înlocuită la configurare, fără a schimba codul existent (*pluggable*).

Principiul segregării interfețelor (**ISP**, *Interface Segregation Principle*) recomandă interfețe mici și specifice, nu interfețe „grădină zoologică", pentru reducerea dependențelor care nu sunt necesare. Se aplică prin divizarea unei interfețe mari în mai multe interfețe focalizate, de exemplu separarea operațiilor de citire a stării comutatorului de operațiile de control, utilizatorii alegând doar contractul de care au nevoie.

Principiul inversiunii dependențelor (**DIP**, *Dependency Inversion Principle*) cere ca modulele de nivel înalt să nu depindă de module de nivel scăzut, ambele să depindă de abstracții, pentru decuplarea părților stabile de detaliile volatile. Se aplică prin injectarea dependențelor la interfață, de exemplu controlerul depinde de o interfață `Comutator`, nu de implementarea concretă, iar implementarea concretă depinde de aceleași abstracții, nu invers.

Legea lui Demeter (LoD, *Law of Demeter*) promovează colaborări locale, „nu vorbi cu străinii", pentru limitarea lanțurilor fragile de dependențe [10]. Un obiect ar trebui să invoce operații ale propriilor colaboratori direcți, nu ale colaboratorilor colaboratorilor. Se aplică evitând apeluri de forma `a().b().c()`, în schimb se introduce o operație dedicată pe colaboratorul direct, care ascunde restul lanțului și reduce propagarea schimbărilor.

Principiul separării comandă–interogare (**CQS**, *Command–Query Separation*) spune că o operație fie modifică starea, fie returnează informație, dar nu ambele, pentru predictibilitate și testare simplificate. Când o metodă are efecte și întoarce rezultate, testele devin greu de citit. Se aplică definind operații de tip „comandă" pentru acțiuni și „interogare" pentru citiri, de exemplu `comuta()` schimbă starea și nu întoarce valori relevante, `estePornit()` doar citește, fără efecte.

Principiul spune, nu întreba (**TDA**, *Tell, Don't Ask*) încurajează trimiterea de mesaje obiectelor pentru a-și face treaba, nu extragerea stării lor pentru a decide în locul lor, pentru menținerea încapsulării și a coeziunii. Logica rămâne acolo unde sunt datele. Se aplică astfel: în loc să citești starea și să decizi în exterior, invoci direct operația potrivită, de exemplu `comutator.comuta()`, iar regula rămâne într-un singur loc, lângă date, reducând dublările de logică.

Principiul accesului uniform (**UAP**, *Uniform Access Principle*) susține că un client nu ar trebui să distingă între o proprietate calculată și una stocată, pentru libertatea de a schimba implementarea fără a afecta codul client [4]. Se aplică expunând operații consistente pentru date, de exemplu `state()` poate citi dintr-un câmp sau o poate calcula. Semnătura rămâne aceeași, iar clienții nu sunt afectați de schimbare.

1.2.3. Ilustrare scurtă a unor principii OO

Principiile de mai sus devin mai clare atunci când sunt privite pe un exemplu mic. Presupunem că aplicația trebuie să lucreze cu un cont și să anunțe undeva operațiile importante. O variantă slabă ar fi ca aceeași clasă să gestioneze soldul, să scrie mesaje în consolă și să decidă formatul raportării. Aceasta ar încălca **principiul responsabilității unice**.

```
public final class AccountBadExample {
    private double balance;

    public void deposit(double amount) {
        if (amount <= 0) {
            System.out.println("Invalid amount");
            return;
        }

        balance += amount;
        System.out.println("Deposit: " + amount);
    }

    public double balance() {
        return balance;
    }
}
```

Clasa de mai sus are cel puțin două motive de schimbare: regulile contului și modul de raportare. Schimbarea ieșirii din consolă cu scriere într-un jurnal de diagnostic (fișier *log*) sau trimitere către un serviciu, ar forța modificarea aceleiași clase care gestionează soldul. O variantă mai bună **separă responsabilitățile**.

```
interface AccountReporter {
    void depositAccepted(double amount);
    void depositRejected(double amount);
}

final class ConsoleAccountReporter implements AccountReporter {
    public void depositAccepted(double amount) {
        System.out.println("Deposit accepted: " + amount);
    }

    public void depositRejected(double amount) {
        System.out.println("Deposit rejected: " + amount);
    }
}
public final class AccountWithReporter {
    private double balance;
    private final AccountReporter reporter;

    public AccountWithReporter(double ib, AccountReporter ar) {
        this.balance = Math.max(0, ib);
        this.reporter = ar;
    }
    public boolean deposit(double amount) {
        if (amount <= 0) {
```

```
            reporter.depositRejected(amount);
            return false;
        }
        balance += amount;
        reporter.depositAccepted(amount);
        return true;
    }
    public double balance() {
        return balance;
    }
}
```

Exemplul anterior **ilustrează simultan mai multe principii**. `AccountWithReporter` are o responsabilitate principală, gestionează soldul. Raportarea este mutată într-o interfață separată, `AccountReporter`. Clasa contului nu depinde de `ConsoleAccountReporter`, ci de o abstracție, ceea ce aplică **programarea spre interfețe** și **inversiunea dependențelor**. În testare, putem furniza o implementare falsă a raportorului, fără să folosim consola.

```
final class FakeAccountReporter implements AccountReporter {
    boolean accepted;
    boolean rejected;

    public void depositAccepted(double amount) {
        accepted = true;
    }
    public void depositRejected(double amount) {
        rejected = true;
    }
}
```

O altă idee utilă este **separarea comandă-interogare**. Metoda `deposit()` este o comandă, deoarece modifică starea contului. Metoda `balance()` este o interogare, deoarece doar citește starea. Când metodele sunt separate astfel, codul devine mai previzibil, iar testele sunt mai ușor de scris.

```
FakeAccountReporter reporter = new FakeAccountReporter();
AccountWithReporter account = new AccountWithReporter(100.0, reporter);

boolean result = account.deposit(50.0);

assert result;
assert account.balance() == 150.0;
assert reporter.accepted;
```

Important este că testul verifică efectele observabile: rezultatul operației, soldul și notificarea raportorului. Nu este nevoie să cunoască detaliile interne ale clasei. Aceasta continuă aceeași idee introdusă la modularizarea structurată, dar într-o formă specifică orientării spre obiecte.

1.2.4. Compunerea și agregarea, înaintea moștenirii

Compunerea (*composition*) înseamnă să construiești funcționalitate prin includerea altor obiecte ca părți, clasa le deține și le orchestrează, iar colaborarea se face prin **interfață** și **delegare**. Un comutator cu temporizator configurabil poate păstra în interior un obiect care implementează politica de temporizare și îi poate cere să își facă treaba. Astfel regula rămâne locală, se poate înlocui la configurare cu o altă politică, iar restul codului nu se schimbă. Această abordare păstrează încapsularea, permite evoluția independentă a componentelor și susține testarea, deoarece componenta internă poate fi substituită în probe cu o dublură care respectă aceeași interfață.

Agregarea (*aggregation*) exprimă o relație mai slabă, obiectele colaboratoare nu sunt neapărat deținute de clasă și pot exista independent. Un controler care primește din exterior referințe la mai multe comutatoare și le coordonează fără a le gestiona ciclul de viață folosește agregare. Beneficiul este decuplarea, controlerul cunoaște doar operațiile, nu decide despre durată de viață sau detalii interne, ceea ce simplifică schimbările și permite compunerea scenariilor la nivel de arhitectură.

Moștenirea (*inheritance*) este utilă când există o relație clară de tip "**este un**" (*is a*), iar tipul de bază oferă un contract stabil care se specializează, de exemplu un comutator inteligent care extinde comutatorul de bază și adaugă raportare sau reguli de siguranță. Totuși, moștenirea impune ierarhii și poate introduce fragilitatea clasei de bază, adică schimbări aparent locale în superclasă pot afecta subclasele în moduri greu de anticipat. De aceea, pentru variații de politică și colaboratori, **compunerea** este de regulă preferată, deoarece exprimă explicit relația "**are un**" (*has a*), menține granițe clare între roluri și ușurează extensibilitatea fără a atinge codul stabil [11].

Un **criteriu practic** este întrebarea dacă se vrea schimbarea a “cum se face ceva” sau a “tipului de obiect”. Pentru schimbarea unei politici, de exemplu strategia de temporizare sau de audit, este potrivită compunerea cu o **interfață** a politicii. Se configurează alt obiect care implementează aceeași **interfață** și nu se editează clasa principală. Pentru un tip nou cu identitate proprie, care trebuie să fie acceptat oriunde este așteptat tipul de bază, moștenirea este potrivită, cu condiția respectării substituției Liskov și a contractelor publice stabilite.

În limbajul vizual **UML** (care nu face obiectul acestei lucrări) diferența se notează și grafic. Compunerea (romb negru în UML) este un întreg cu părți pe care le deține, ciclul de viață al părților urmează întregul. Agregarea (romb alb în UML) este o asociere relaxată între obiecte care pot exista separat [12]. În proiectare, alegerea compunere sau agregare se face după cine deține responsabilitatea pentru starea internă și cine controlează ciclul de viață, criterii care influențează direct testarea, depanarea și modul de evoluție a componentelor.

Pentru a face concrete diferențele și a arăta de ce compunerea este adesea prima opțiune când se variază o regulă, urmează un **exemplu** care pune față în față compunerea și moștenirea pe aceeași cerință, întârzierea unei comutări.

```
// variatia de comportament, interfata minimalista
interface BeforeToggle {
    void run();
}
// doua implementari foarte simple
final class Noop implements BeforeToggle {
    public void run() { System.out.println("noop"); }
}
final class Beep implements BeforeToggle {
    public void run() { System.out.println("beep"); }
}
```

```java
// COMPUNERE, comutatorul primeste variatia prin constructor
final class SwitchWithPolicy {
    private boolean on;
    private final BeforeToggle fn;

    SwitchWithPolicy(BeforeToggle fn) { this.fn = fn; }

    public void toggle() {
        fn.run();        // delegare catre componenta variabila
        on = !on;
        System.out.println("state: " + on);
    }
}

// MOSTENIRE, comportamentul se adauga prin override (suprascriere a codului)
class BasicSwitch {
    protected boolean on;
    public void toggle() {
        on = !on;
        System.out.println("state: " + on);
    }
}

final class BeepingSwitch extends BasicSwitch {
    @Override
    public void toggle() {
        System.out.println("beep");
        super.toggle();  // pastrez logica de baza
    }
}

// demo minimal, doar pentru a vedea ordinea mesajelor
class Demo {
    public static void main(String[] args) {
        SwitchWithPolicy a = new SwitchWithPolicy(new Noop());
        SwitchWithPolicy b = new SwitchWithPolicy(new Beep());
        a.toggle(); // noop, apoi state: true
        b.toggle(); // beep, apoi state: true
        BasicSwitch s1 = new BasicSwitch();
        BeepingSwitch s2 = new BeepingSwitch();
        s1.toggle(); // state: true
        s2.toggle(); // beep, apoi state: true
    }
}
```

Exemplul face vizibil locul unde se introduce variația de comportament. În **compunere** obiectul principal primește prin constructor o politică `BeforeToggle`, delegă execuția printr-o interfață și își păstrează logica de bază neschimbată. Astfel se poate schimba ulterior doar obiectul injectat, fără a edita clasa, ceea ce respectă programarea spre interfețe și principiul deschis la extindere, închis la modificare. În **moștenire** variația se adaugă prin rescrierea metodei în subclasă, soluția rămâne utilă când se dorește un tip nou cu aceeași identitate de tip, dar depinde de ierarhie și de stabilitatea superclasei.

Din perspectiva testării **compunerea** permite dubluri simple care implementează interfața `BeforeToggle` și verifică efectul observabil, fără a porni infrastructura reală. Tot în **compunere**, pentru depanare, punctul de variație este localizat în componenta injectată, iar controlul ciclului de viață și al configurației rămâne în afara clasei principale, ceea ce scurtează investigațiile. În **moștenire**, aceste puncte de variație sunt răspândite în ierarhie, iar depanarea depinde de stabilitatea superclasei.

1.2.5. Impactul orientării spre obiecte pentru depanare și testare

Orientarea spre obiecte **simplifică testarea** atunci când se programează spre interfețe și când variațiile de comportament se introduc prin compunere. În exemplul din 1.2.3, clasa comutator primește prin constructor o politică `BeforeToggle`. În test, această politică poate fi înlocuită cu o dublură simplă, o dublură de test sau un ***mock*** (**obiect simulat**), care notează ce s-a întâmplat sau expune un efect observabil [5]. Astfel, testul verifică rezultatul și ordinea apelurilor fără să depindă de detalii interne. Avantajul practic este izolarea, componenta variabilă este separată printr-o **interfață**, iar comutatorul rămâne ușor de verificat în scenarii pozitive și negative.

Exemplul cu **`AccountReporter`** arată aceeași idee pe un caz mic: dependența variabilă este exprimată printr-o interfață, iar în test poate fi înlocuită cu o implementare falsă. Astfel, testul verifică logica obiectului principal fără să depindă de consolă, fișiere, rețea sau alte detalii externe.

Încapsularea reduce suprafața unde pot apărea erori, starea rămâne ascunsă, iar validările se fac la interfața obiectului. Asta înseamnă mesaje de eroare coerente și puncte clare de refuz, ceea ce scade timpul până la diagnostic. În aceeași logică, principiul responsabilității unice ajută depanarea, clasele mici, coezive, cu rol bine definit, produc rapoarte de eroare mai clare și trasee de execuție mai scurte. Când apare o abatere, locul probabil este una dintre puținele clase candidate, nu un tip monolit cu roluri amestecate.

Polimorfismul permite reutilizarea testelor pe implementări diferite ale aceleiași interfețe. O suită de teste scrisă pentru `Comutator` se poate rula neschimbată atât pe varianta cu bip sonor, cât și pe varianta fără efect, atâta vreme cât ambele respectă contractul public. Acesta este efectul direct al principiului substituției Liskov, aceeași interfață, aceleași așteptări, implementări diferite, aceleași teste. În practică, această proprietate accelerează evoluția, se poate adăuga o nouă implementare și se pot rula imediat testele existente pentru a confirma compatibilitatea.

Compunerea facilitează și diagnosticarea incidentelor în producție. Punctul de variație este localizat într-o componentă introdusă prin injectarea dependențelor, astfel mesajele de diagnostic și măsurătorile relevante pot fi colectate la nivelul acelei interfețe, fără a încărca inutil clasa principală. O problemă provenită din politica de temporizare poate fi investigată prin activarea unei implementări instrumentate doar pentru acea politică, fără modificarea logicii comutatorului. Acest mod de lucru aliniază testarea, depanarea și operarea, aceeași graniță arhitecturală funcționează în toate etapele.

Moștenirea rămâne utilă, dar este mai sensibilă în testare și depanare, deoarece schimbările din superclasă pot afecta subclasele în moduri greu de anticipat. Testele trebuie să acopere atât contractul de bază, cât și specializările, iar diagnosticul necesită uneori urmărirea lanțului de apeluri în ierarhie. De aceea, pentru variații de politică sau colaboratori, compunerea este preferată, deoarece exprimă clar relația are un, păstrează granițe curate și permite dubluri de test locale.

O regulă practică este **să se testeze prin efecte observabile, nu prin inspectarea stării interne**. Interfața publică devine contractul testabil, iar tot ce este intern poate evolua liber. Aceasta reduce fragilitatea testelor și susține refactorizările. În plus, separarea comandă interogare simplifică aserțiile, metodele de comandă au efecte verificabile, metodele de interogare nu schimbă starea, ceea ce produce teste mai clare și mai robuste.

1.2.6. Stiluri de dezvoltare a codului

În practică, programatorii nu construiesc întotdeauna codul în aceeași ordine. Uneori **se pornește de la implementare**, apoi se extrage interfața și se adaugă testele**.** Alteori **se scrie mai întâi un program** `main` care exprimă felul dorit de utilizare, apoi se completează clasele necesare.

În stilul *test-first*, specific *test-driven development* (TDD), se scrie mai întâi un test care exprimă comportamentul așteptat, apoi se implementează codul minim care face testul să treacă [13]. **În stilul *interface-first*, se definește mai întâi contractul public**, iar implementările apar ulterior.

Aceste stiluri nu se exclud. Ele exprimă **priorități** diferite. **Implementarea înainte** este rapidă când problema este exploratorie, dar riscă să producă interfețe neclare. **Testul înainte** clarifică așteptările și cazurile limită, dar cere disciplină. **Interfața înainte** este utilă când mai multe echipe sau componente trebuie să colaboreze, deoarece fixează contractul înainte ca detaliile interne să fie cunoscute. Un **exemplu simplu** poate porni de la un comutator.

```
interface Switch {
    void toggle();
    boolean isOn();
}
```

În stilul *interface-first*, această interfață apare înaintea implementării. Ea spune ce se poate face cu un comutator, nu cum este realizat intern.

```
final class SimpleSwitch implements Switch {
    private boolean on;
    public void toggle() {  on = !on;
    }
    public boolean isOn() {
        return on;
    }
}
```

În stilul *main-first*, se scrie mai întâi codul care arată cum vrem să folosim obiectul.

```
public class Demo {
    public static void main(String[] args) {
        Switch sw = new SimpleSwitch();
        sw.toggle();
        System.out.println(sw.isOn());
    }
}
```

În stilul *test-first*, se exprimă mai întâi comportamentul așteptat. Exemplul folosește JUnit, *framework* standard pentru teste unitare Java [14].

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
class SimpleSwitchTest {
    @Test
    void toggleChangesState() {
        Switch sw = new SimpleSwitch();
        sw.toggle();
        assertTrue(sw.isOn());
    }
}
```

Același exemplu arată că **proiectarea nu este doar o problemă de sintaxă**. Ordinea în care scriem codul influențează forma interfețelor, claritatea contractului și ușurința testării. **Pentru cod didactic și pentru sisteme mici**, *`main`-first* ajută la înțelegerea utilizării. **Pentru biblioteci, componente sau servicii**, *interface-first* este adesea mai potrivit. **Pentru logică de domeniu și reguli cu multe cazuri limită**, *test-first* reduce riscul de regresii.

1.2.7. Influența programării funcționale asupra limbajelor moderne

Deși firul principal al acestui capitol urmărește evoluția de la programarea structurată la orientarea spre obiecte, componente și servicii, merită amintită și influența programării funcționale. **Programarea funcțională** pune accent pe **funcții pure, imutabilitate, compunerea funcțiilor** și **reducerea efectelor laterale** [15]. Într-o funcție pură, același set de intrări produce întotdeauna același rezultat, fără modificarea unei stări externe.

Aceste idei au influențat puternic limbajele moderne, inclusiv limbaje folosite frecvent în stil OO, precum Java, C#, JavaScript sau Kotlin**. Expresiile *lambda*, operațiile de tip *map*, *filter*, *reduce*, fluxurile** și **colecțiile imutabile** (*immutable*) sunt exemple de concepte funcționale integrate în programarea curentă. În Java, pachetul `java.util.stream` documentează explicit operații funcționale de tip *filter-map-reduce* peste colecții [16].

Pentru arhitectură, valoarea practică este clară: mai puțină stare modificabilă înseamnă cod mai ușor de testat, mai puține efecte laterale și o comportare mai previzibilă în contexte concurente sau asincrone. Programarea funcțională nu înlocuiește orientarea spre obiecte în această carte, dar completează aceeași preocupare centrală: controlul complexității prin reducerea dependențelor ascunse și prin separarea clară a responsabilităților.

În exemplul următor, nu se modifică o variabilă globală și nu se controlează manual o buclă. Codul descrie transformarea colecției: se păstrează valorile pare, apoi se calculează suma. Pentru cazuri simple, o buclă clasică este la fel de corectă. Avantajul stilului funcțional apare mai ales când transformările sunt compuse clar și când se evită starea partajată modificabilă.

```
import java.util.List;

public class Demo {
    public static void main(String[] args) {
        List<Integer> values = List.of(1, 2, 3, 4, 5);

        int sumOfEvenValues = values.stream()
            .filter(x -> x % 2 == 0)
            .mapToInt(x -> x)
            .sum();
        System.out.println(sumOfEvenValues);
    }
}
```

Programarea funcțională adaugă încă o perspectivă asupra modularizării: comportamentul poate fi exprimat prin funcții compuse, datele pot fi tratate ca valori imutabile, iar efectele laterale pot fi limitate la zone controlate. În practică, aplicațiile moderne combină frecvent stiluri diferite: cod structurat pentru algoritmi simpli, obiecte pentru modelarea domeniului, concepte funcționale pentru transformări de date și *framework*-uri pentru organizarea fluxului general. După aceste paradigme de bază, următorul pas este recunoașterea unor **forme recurente de proiectare**. Aceste forme, cunoscute ca *pattern*-uri de proiectare, oferă soluții consacrate pentru probleme care apar des atunci când obiectele și componentele trebuie să colaboreze.

1.3. *Pattern*-uri de proiectare OO

După ce programarea structurată a adus un nivel de reutilizare, prin blocuri și funcții, iar orientarea spre obiecte a adăugat încă două, prin clase și moștenire, *pattern*-urile de proiectare au introdus un nivel suplimentar: **reutilizarea soluțiilor**, nu doar a codului, ceea ce a marcat o nouă etapă în felul în care se scrie software. Un ***pattern* de proiectare** (**DP**, *design pattern*) surprinde o **problemă recurentă** și oferă o **schemă de soluție** care poate fi aplicată în contexte diferite, fără a fixa o implementare anume, astfel că echipele capătă un **vocabular comun** și pot proiecta mai repede și mai sigur [11].

Cartea clasică a autorilor cunoscuți ca **GoF** (*Group of four* - Gamma, Helm, Johnson și Vlissides) prezintă 23 de *pattern*-uri, suficiente cât să umple singure o carte [11]. Aici facem doar o trecere de ansamblu, pentru a înțelege ideea de probleme recurente în software și soluțiile lor consacrate ca bune practici. Între timp au apărut sute de alte *pattern*-uri, câteva zeci fiind folosite intensiv, iar dintre cele 23 GoF numai o parte sunt utilizate sistematic în proiectele moderne. Începem cu cele structurale, deoarece sunt cele mai intuitive raportat la modelarea OO, continuăm cu cele comportamentale și încheiem cu cele creaționale, iar în interiorul fiecărei familii ordinea este aproximativ după frecvența de utilizare în practică.

1.3.1. *Pattern*-uri de proiectare structurale (GoF)

Pattern-urile de proiectare **structurale** descriu **felul în care obiectele și clasele pot fi compuse pentru a obține forme mai flexibile de colaborare**. Ele nu se concentrează pe un algoritm anume, ci pe organizarea relațiilor dintre părți. De aceea sunt foarte utile pentru a înțelege cum pot fi conectate componente existente, cum poate fi ascuns un subsistem complex sau cum se poate controla accesul la o resursă.

În practică, *pattern*-urile structurale apar frecvent la granițele dintre module, biblioteci, servicii sau subsisteme. *Adapter* face compatibile interfețe diferite, *Proxy* controlează accesul la aceeași interfață, *Facade* simplifică utilizarea unui subsistem complex, iar *Decorator* adaugă responsabilități fără a modifica obiectul de bază. Ideea comună este că structura colaborării poate fi schimbată fără ca toate clasele implicate să fie rescrise.

Pattern-ul de proiectare ***Adapter*** permite **colaborarea între două părți care au interfețe diferite**. În loc să se modifice clientul sau furnizorul, se introduce un adaptor care traduce apelurile dintr-o formă în alta. Este util când un serviciu extern are un contract pe care nu îl putem schimba, iar sistemul existent așteaptă alt contract. Câștigul este reutilizarea unei implementări existente fără refactorizări masive. Atenția trebuie acordată lanțurilor de adaptoare, deoarece mai mulți pași de transformare pot ascunde nepotriviri de modele și pot complica depanarea. Când apare nealiniere conceptuală, este mai sănătos să se negocieze un contract comun decât să se acumuleze adaptoare. Un adaptor este util atunci când avem deja o clasă utilă, dar interfața ei nu se potrivește cu interfața așteptată de restul aplicației. În loc să modificăm clasa existentă, introducem un obiect intermediar care traduce apelurile.

Pattern-ul *Adapter* poate fi privit din două perspective complementare. Prima este **perspectiva structurală**: clientul cunoaște interfața așteptată, adaptorul implementează această interfață și traduce apelul către clasa existentă. Figura 1.1 arată rolurile principale ale *pattern*-ului. Clientul depinde de interfața așteptată, nu de clasa existentă. `Adapter` implementează interfața cunoscută de client și conține logica de transformare. `Adaptee` este clasa deja existentă, utilă, dar cu o interfață incompatibilă.

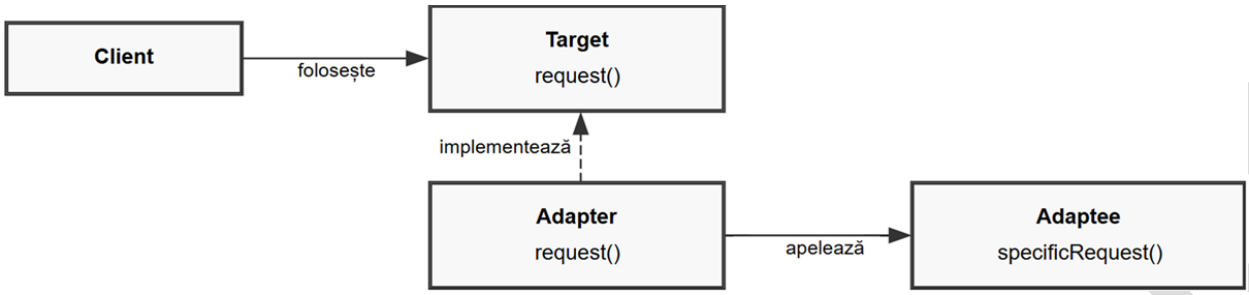


Figura 1.1. Structura *pattern*-ului *Adapter* (adaptarea unei interfețe la alta)

A doua este **perspectiva comportamentală**: la execuție, clientul apelează adaptorul ca și cum ar apela interfața normală, iar adaptorul transformă apelul și îl transmite către obiectul adaptat. Figura 1.2 arată aceeași idee la execuție. Clientul trimite un apel normal către `Adapter`, de exemplu `request()`. `Adapter` traduce acest apel în forma cerută de `Adaptee`, de exemplu `specificRequest()`, apoi întoarce rezultatul către client. Astfel, codul client rămâne stabil, iar clasa existentă poate fi reutilizată fără modificare.

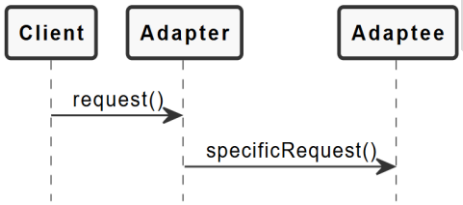


Figura 1.2. Interacțiunea în *pattern*-ul *Adapter* (traducerea apelului către obiectul adaptat)

Exemplul următor transpune această structură într-un **caz concret**. Aplicația cunoaște interfața `MessageSender`, dar biblioteca existentă oferă deja clasa `LegacyEmailClient`, cu o metodă publică diferită. Adaptorul face legătura dintre interfețe, fără să modifice clasa existentă.

```
interface MessageSender {
    void send(String to, String text);
}
final class LegacyEmailClient {
    public void sendEmail(String address, String body) {
        System.out.println("Email to " + address + ": " + body);
    }
}
final class EmailAdapter implements MessageSender {
    private final LegacyEmailClient client;
    EmailAdapter(LegacyEmailClient client) {
        this.client = client;
    }

    public void send(String to, String text) {
        client.sendEmail(to, text);
    }
}
public class Demo {
    public static void main(String[] args) {
        MessageSender sender = new EmailAdapter(new LegacyEmailClient());
        sender.send("student@example.com", "Hello!");
    }
}
```

Clientul lucrează cu `MessageSender`, adică **interfața de care are nevoie aplicația**. Clasa veche `LegacyEmailClient` nu este modificată. `EmailAdapter` face traducerea dintre cele două lumi. Avantajul este că putem integra cod existent, biblioteci externe sau servicii cu interfețe diferite fără să rescriem logica principală.

Adapter este **util mai ales la granițe**: între aplicație și o bibliotecă veche, între un domeniu intern și un API extern, între o componentă nouă și una deja existentă. Totuși, dacă apar prea multe adaptoare în lanț, acesta poate fi un semn că modelele conceptuale nu se mai potrivesc și că trebuie renegociat contractul dintre părți. Un alt *pattern* structural apropiat ca formă, dar diferit ca intenție, este *Proxy*. Și aici apare un obiect intermediar, însă scopul nu este adaptarea unei interfețe la alta, ci controlul accesului la aceeași interfață.

Pattern-ul de proiectare ***Proxy*** introduce **un obiect reprezentant în fața țintei pentru a controla accesul**. Clientul folosește același contract, însă apelul trece prin reprezentant. În acest punct intermediar se pot face verificări de securitate, de exemplu controlul drepturilor înainte de a apela ținta. Se poate aplica și cache, adică memorarea rezultatelor recente pentru a evita recalculări costisitoare sau apeluri de rețea repetate. Se poate folosi *lazy loading*, adică amânarea creării sau încărcării resurselor grele până în momentul în care chiar sunt necesare. De asemenea se pot adăuga măsurători, de exemplu înregistrarea timpilor de execuție sau a numărului de apeluri. Avantajul este că se adaugă capabilități transversale fără a modifica clasa reală. Riscul este ascunderea costurilor de rețea sau a latențelor în spatele unei interfețe aparent locale, ceea ce poate surprinde clienții. De aceea contractul ar trebui să seteze așteptări realiste despre timpi de răspuns și erori. În practică apar multe situații de tip *proxy*, de la acces la fișiere la servicii la distanță, de aceea modelarea prin *Proxy* este foarte folosită.

Proxy **seamănă** la prima vedere cu *Adapter*, deoarece introduce un obiect **intermediar**. Diferența importantă este că *Proxy* **păstrează aceeași interfață** ca obiectul real, în timp ce *Adapter* **schimbă interfața** pentru a o potrivi cu așteptările clientului. *Proxy* nu traduce neapărat un contract în altul, ci controlează accesul la același contract. Figura 1.3 arată structura, iar Figura 1.4 arată interacțiunea dintre client, proxy și obiectul real.

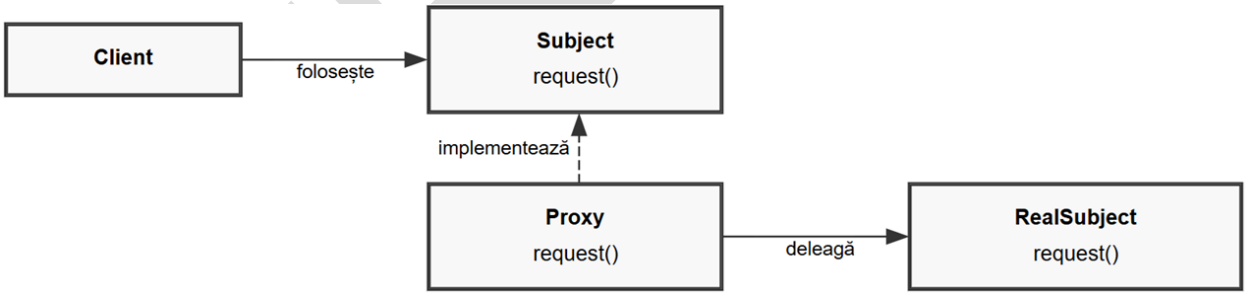


Figura 1.3. Structura *pattern*-ului *Proxy* (controlul accesului la aceeași interfață / reprezentant la distanță)

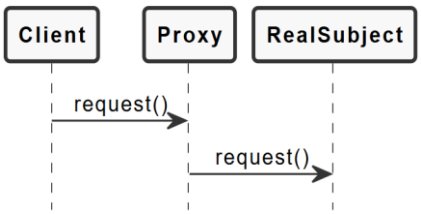


Figura 1.4. Interacțiunea în *pattern*-ul *Proxy* (apel prin reprezentant către obiectul real)

În execuție, clientul apelează *proxy*-ul prin **aceeași interfață**, iar *proxy*-ul poate adăuga verificări, scriere în jurnal sau *caching* înainte de a delega către obiectul real. Exemplul următor folosește un *proxy* pentru a scrie mesaje de diagnostic în jurnal în jurul unui serviciu, fără a modifica serviciul real.

```
interface ReportService {
    String report();
}
final class RealReportService implements ReportService {
    public String report() {
        return "Report content";
    }
}
final class LoggingReportProxy implements ReportService {
    private final ReportService target;
    LoggingReportProxy(ReportService target) {
        this.target = target;
    }
    public String report() {
        System.out.println("Before report");
        String result = target.report();
        System.out.println("After report");
        return result;
    }
}
public class Demo {
    public static void main(String[] args) {
        ReportService service = new LoggingReportProxy(new RealReportService());
        System.out.println(service.report());
    }
}
```

Clientul folosește tot interfața `ReportService`. Nu știe dacă în spate se află direct `RealReportService` sau un *proxy*. Putem adăuga mesaje de diagnostic, verificare de drepturi, cache sau măsurători fără să schimbăm codul clientului și fără să modificăm clasa reală.

Comparația cu *Adapter* este importantă. *Adapter* face **compatibile** două interfețe diferite. *Proxy* păstrează interfața, dar **interceptează apelul** pentru a adăuga un comportament suplimentar. De aceea *Proxy* este foarte folosit în *framework*-uri, în accesul la servicii la distanță, în mecanisme de securitate și în implementări de injectări de dependențe.

Pattern-ul de proiectare ***Composite*** **permite tratarea arborescentă uniformă a obiectelor individuale și a ansamblurilor lor**. Frunzele și compozițiile expun aceeași interfață, astfel clientul poate comanda o singură entitate sau un întreg arbore fără cod separat pentru fiecare caz. Este potrivit pentru ierarhii parte întreg, de exemplu meniuri, scene grafice sau topologii de dispozitive. Se obține simplificarea codului client. Probleme apar când logica specifică frunzelor este împinsă în rădăcina compoziției, ceea ce scade coeziunea. O proiectare bună păstrează comportamentele specifice acolo unde apar și evită dependențele încrucișate între niveluri.

Pattern-ul de proiectare ***Facade*** oferă **o intrare simplă peste un subsistem complex**. Clientul primește câteva operații bine alese, iar detaliile, ordinea pașilor și dependențele interne rămân ascunse. Acest lucru reduce numărul de dependențe vizibile și scade curba de învățare pentru utilizatorii subsistemului. Este utilă când un nucleu complicat trebuie expus restului aplicației printr-un *frontend* clar. Riscul este ca fațada să adune responsabilități diverse și să devină o clasă greu de întreținut. Dacă se observă această tendință, se revine la principiul responsabilității unice și se extrag fațade mai mici, orientate pe scenarii.

Pattern-ul de proiectare ***Decorator* atașează dinamic responsabilități suplimentare unui obiect existent, fără a crea noi subclase ale lui**. Un decorator înfășoară obiectul și interceptează apelurile, adăugând comportamente punctuale, de exemplu scriere în jurnal, măsurarea duratei apelului sau reîncercare. Mai multe decorații pot fi înlănțuite pentru a compune capabilități. Avantajul este flexibilitatea, deoarece se pot activa sau dezactiva comportamente la configurare. Limita apare când lanțul de decorații devine lung și ordinea lor devine greu de urmărit. În astfel de cazuri se documentează ordinea și se folosesc nume explicite pentru decorații.

Pattern-ul de proiectare ***Bridge* separă o abstracție de implementarea ei astfel încât ambele să poată evolua independent**. În loc să se creeze o ierarhie mare care combină mai multe dimensiuni de variație, se definesc două ierarhii mai mici, una pentru abstracții și alta pentru implementări, apoi se compun la execuție (*runtime*). Exemplul clasic este separarea formei de desenare, forma variază independent de modul de randare. Câștigul este evitarea exploziei de subclase și combinarea controlată a variantelor. Costul este complexitatea inițială, deoarece trebuie înțeleasă și menținută relația dintre cele două axe.

Pattern-ul de proiectare ***Flyweight* economisește memorie partajând starea intrinsecă între multe obiecte fine**. Atributele care sunt identice la multe instanțe se păstrează o singură dată și se referă la ele din mai multe locuri. Starea extrinsecă, care diferă de la un caz la altul, se furnizează din exterior la apel. Acest model este util când există milioane de obiecte similare, de exemplu noduri într-o structură repetitivă sau caractere într-un editor. Beneficiul este reducerea presiunii pe memorie și îmbunătățirea localizării cache-ului. Provocarea este gestionarea atentă a stării externe pentru a evita erori subtile și pentru a menține interfața clară pentru clienți.

1.3.2. *Pattern*-uri de proiectare comportamentale (GoF)

Pattern-urile de proiectare **comportamentale** descriu **felul în care obiectele colaborează pentru a realiza un comportament**. *Pattern*-urile structurale se concentrează pe forma relațiilor dintre clase și obiecte, pe când *pattern*-urile comportamentale pun accent pe comunicare, delegare, notificare, alegerea algoritmului, parcurgerea colecțiilor sau controlul pașilor de execuție. Ele ajută la separarea deciziilor de implementări concrete și fac mai ușor de modificat modul în care obiectele reacționează unele la altele.

În practică, aceste *pattern*-uri **apar foarte des în** *framework*-uri, interfețe grafice, fluxuri de lucru, conducte de procesare (*pipelines*), mecanisme de evenimente și sisteme asincrone. *Observer* exprimă notificarea schimbărilor, *Command* transformă o acțiune într-un obiect, *Strategy* permite înlocuirea unui algoritm, *Iterator* ascunde structura internă a unei colecții, iar *State* mută comportamentul dependent de stare în obiecte dedicate. Toate au aceeași idee de fond: comportamentul devine mai flexibil atunci când colaborarea este exprimată prin interfețe și roluri clare, nu prin apeluri rigide și condiționale răspândite în cod.

Pattern-ul de proiectare ***Observer* decuplează emițătorul unor evenimente de consumatorii acestora printr-un mecanism de abonare**. Emitentul notifică evenimente, observatorii reacționează fără ca părțile să se cunoască direct. Avantajul este propagarea schimbărilor fără dependențe rigide. Grija principală privește ordinea, consistența și dezabonarea pentru a evita scurgerile de memorie. În practică este folosit pentru UI reactiv, actualizări de stare în clienți, ***webhooks*** (apeluri HTTP trimise de un sistem către altul, când are loc un eveniment), notificări în sisteme distribuite. Figura 1.5 arată structura mecanismului, iar Figura 1.6 arată ordinea *register-notify-update*.

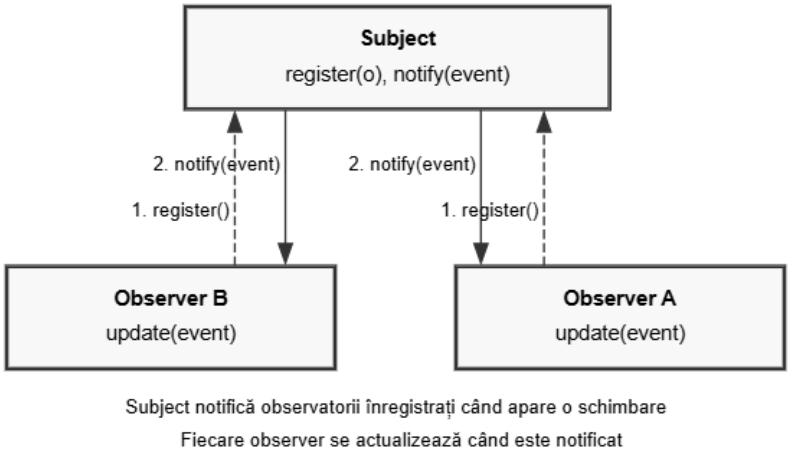


Figura 1.5. Structura *pattern*-ului *Observer* (mecanism *register-notify*)

Pattern-ul *Observer* poate fi numit intuitiv și **mecanism *register-notify***: observatorii se înregistrează la un subiect. Subiectul îi notifică atunci când apare o schimbare, iar fiecare observator își actualizează starea sau reacționează local. Această denumire ajută deoarece descrie direct pașii principali: cineva se abonează, apoi este anunțat.

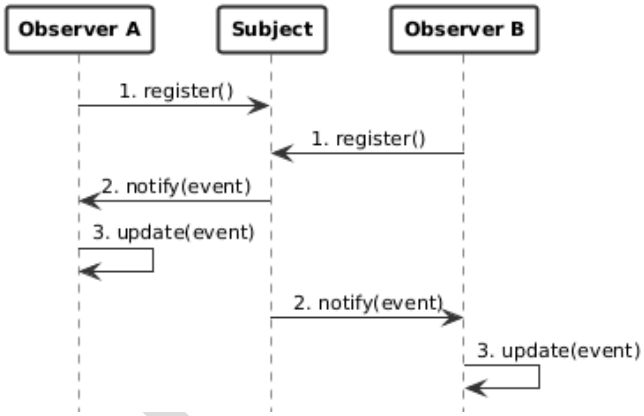


Figura 1.6. Interacțiunea în *pattern*-ul *Observer* (*register*, *notify* și *update*)

Un **exemplu simplu** poate porni de la un **comutator**. Când starea se schimbă, comutatorul notifică observatorii înregistrați. Observatorii pot afișa mesajul, pot actualiza o interfață grafică, pot trimite evenimentul mai departe sau pot scrie în jurnal.

```
import java.util.ArrayList;
import java.util.List;

interface SwitchObserver {
    void stateChanged(boolean on);
}
final class SwitchSubject {
    private boolean on;
    private final List<SwitchObserver> observers = new ArrayList<>();
    public void register(SwitchObserver observer) {
        observers.add(observer);
    }
    public void unregister(SwitchObserver observer) {
```

```
        observers.remove(observer);
    }
    public void toggle() {
        on = !on;
        notifyObservers();
    }
    public boolean isOn() {
        return on;
    }
    private void notifyObservers() {
        for (SwitchObserver observer : observers) {
            observer.stateChanged(on);
        }
    }
}
```

Implementarea unui observator poate fi foarte simplă:

```
final class ConsoleSwitchObserver implements SwitchObserver {
    public void stateChanged(boolean on) {
        System.out.println("Switch state changed: " + on);
    }
}
```

Utilizarea *pattern*-ului *Observer*:

```
public class Demo {
    public static void main(String[] args) {
        SwitchSubject sw = new SwitchSubject();
        sw.register(new ConsoleSwitchObserver());
        sw.toggle();
        sw.toggle();
    }
}
```

În acest exemplu, `SwitchSubject` nu cunoaște detaliile observatorului. El știe doar că observatorul respectă interfața `SwitchObserver`. Astfel, se poate adăuga ulterior un observator pentru interfața grafică, unul pentru jurnalizare sau unul pentru trimiterea unui eveniment către alt sistem, fără modificarea clasei `SwitchSubject`.

Avantajul principal este **decuplarea**. Subiectul emite schimbarea, iar observatorii reacționează independent. Totuși, apar și câteva riscuri practice. Dacă observatorii sunt notificați într-o ordine importantă, această ordine trebuie documentată. Dacă observatorii nu se dezabonează atunci când nu mai sunt necesari, pot apărea scurgeri de memorie. Dacă un observator execută cod lent, notificarea tuturor celorlalți poate fi întârziată.

Pattern-ul de proiectare ***Command* încapsulează o acțiune într-un obiect**. Comanda descrie ce trebuie făcut și cu ce date, poate fi pusă într-o coadă, retransmisă dacă eșuează, salvată pe disc pentru rulare ulterioară, ținută în jurnal pentru urmărire. Invocatorul nu are nevoie să cunoască implementarea efectivă. Este util în cozi de lucru, programatoare de sarcini, funcții de undo și în fluxuri unde trebuie garantat că acțiunea chiar rulează. În **servere web**, cererile se transformă în obiecte acțiune care stau în coadă până primesc resurse, apoi sunt executate una câte una. În **Android**, *Intent*-urile joacă un rol similar, descriu acțiunea și datele, sistemul decide când și unde o rulează, iar mecanisme precum *JobScheduler* sau *WorkManager* programează comenzi pentru mai târziu, cu reîncercări și condiții de rulare [17], [18]. Pentru acțiuni foarte simple adăugarea acestui strat poate complica inutil codul fără beneficii reale.

Pattern-ul *Command* **transformă o acțiune într-un obiect**. Această idee este importantă deoarece o acțiune devine astfel transmisibilă, memorabilă și executabilă mai târziu. În loc ca un invocator să cunoască direct metoda concretă care trebuie apelată, el primește o comandă și o execută. Figura 1.7 arată structura *pattern*-ului, iar Figura 1.8 arată interacțiunea dintre invocator, comandă și receptor.

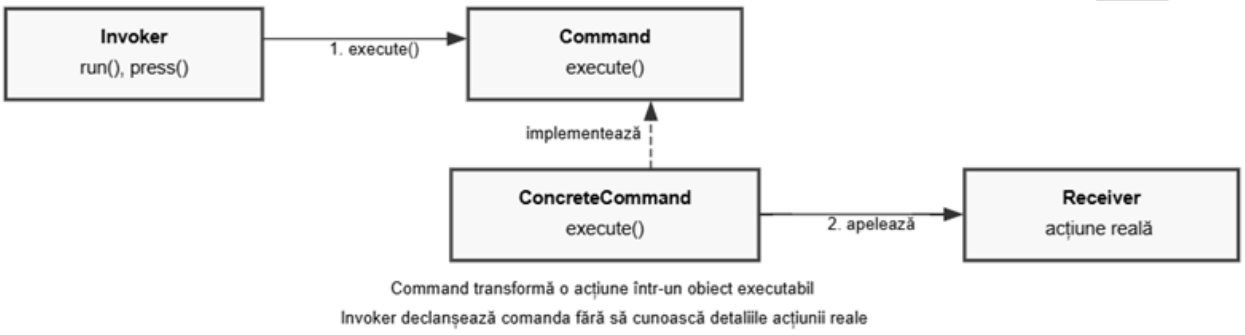


Figura 1.7. Structura *pattern*-ului *Command* (acțiune încapsulată într-un obiect)

Diagrama de interacțiune a *pattern*-ului *Command* evidențiază faptul că invocatorul execută o comandă, iar comanda transmite acțiunea concretă către receptor.

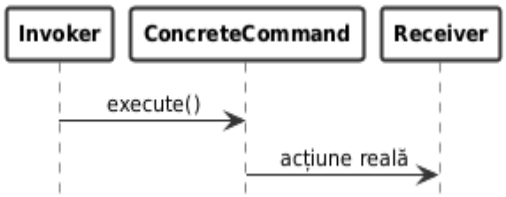


Figura 1.8. Interacțiunea în *pattern*-ul *Command* (invocare, execuție și acțiune reală)

Un exemplu simplu poate privi din nou comutatorul. Comanda va încapsula acțiunea de comutare.

```
interface Command {
    void execute();
}

final class LightSwitch {
    private boolean on;

    public void toggle() {
        on = !on;
        System.out.println("Light is on: " + on);
    }
    public boolean isOn() {
        return on;
    }
}

final class ToggleCommand implements Command {
    private final LightSwitch lightSwitch;

    ToggleCommand(LightSwitch lightSwitch) {
```

```
        this.lightSwitch = lightSwitch;
    }
    public void execute() {
        lightSwitch.toggle();
    }
}
```

Invokerul nu știe ce face concret comanda. El știe doar că poate apela `execute`.

```
final class Button {
    private final Command command;

    Button(Command command) {
        this.command = command;
    }
    public void press() {
        command.execute();
    }
}
```

Utilizarea *pattern*-ului *Command* devine:

```
public class Demo {
    public static void main(String[] args) {
        LightSwitch lightSwitch = new LightSwitch();
        Command command = new ToggleCommand(lightSwitch);
        Button button = new Button(command);
        button.press();
        button.press();
    }
}
```

În acest exemplu, `Button` nu depinde de `LightSwitch`. Butonul poate executa orice comandă, nu doar comutarea unei lumini. Aceasta este puterea *pattern*-ului: acțiunea este separată de obiectul care o declanșează. Se pot introduce comenzi pentru salvare, ștergere, trimitere mesaj, reîncercare, anulare sau procesare asincronă, fără schimbarea invocatorului.

În *framework*-uri web, ideea apare atunci când o cerere HTTP este transformată într-o acțiune sau într-un *handler*. În ***Struts***, clasele de tip ***Action*** au jucat acest rol, primind cererea și executând o operație a aplicației. În ***Spring MVC***, un controler sau o metodă de controler poate fi privită ca punctul în care cererea este mapată către o acțiune. În ***Android***, un ***Intent*** are un rol apropiat: descrie o intenție de acțiune, împreună cu datele necesare, iar sistemul decide componenta care o poate trata. **Nu este același cod ca în exemplul GoF clasic**, dar **ideea de acțiune descrisă, transmisă și executată ulterior** este foarte apropiată.

Command este **util mai ales când acțiunile trebuie puse într-o coadă, executate mai târziu, repetate în caz de eșec, salvate în jurnal** sau **anulate**. Dacă acțiunea este foarte simplă și nu avem nevoie de aceste beneficii, introducerea unei clase de comandă poate fi inutilă. Ca multe *pattern*-uri, *Command* este valoros atunci când rezolvă o problemă reală de decuplare, programare sau control al execuției.

Observer și *Command* sunt printre cele mai importante *pattern*-uri comportamentale pentru arhitecturile moderne. *Observer* pregătește înțelegerea sistemelor bazate pe evenimente, iar *Command* pregătește înțelegerea cozilor de lucru, a *handler*-elor, a mecanismelor de reîncercare (*retry*) și a execuției amânate. Celelalte *pattern*-uri comportamentale completează aceeași idee generală: **separarea deciziilor, a pașilor de execuție și a colaborărilor dintre obiecte**.

Pattern-ul de proiectare ***Iterator*** **oferă o modalitate uniformă de a parcurge elementele unei colecții fără a expune structura internă**. Clientul scrie același cod de traversare pentru liste, mulțimi sau arbori. Beneficiul este lizibilitatea și decuplarea, iar considerațiile practice includ concurența și durata de viață a iteratorului. În lumea reală este peste tot, de la colecțiile din limbaje până la citirea fluxurilor mari de date.

Pattern-ul de proiectare ***Visitor*** **separă operațiile de structurile de date**. Se pot adăuga comportamente noi fără a modifica tipurile vizitate, cu prețul configurării inițiale a dublului dispatch. Este potrivit când ierarhia de elemente este stabilă, iar operațiile se schimbă des. În practică apare la traversarea arborilor de sintaxă, generare de rapoarte, validări multiple peste aceeași structură.

Pattern-ul de proiectare ***Template Method*** **fixează scheletul unui algoritm într-o superclasă, iar pașii variabili sunt rescriși în subclase**. Se pot standardiza etapele, păstrând flexibilitatea locală acolo unde este nevoie. Este potrivit când ordinea pașilor este stabilă, dar implementarea unor pași diferă. Alternativa modernă este *Strategy*, mai ales când se dorește evitarea ierarhiilor adânci. În practică apare în conducte de procesare (*pipelines*) cu pași bine stabiliți.

Pattern-ul de proiectare ***State*** **mută comportamentul dependent de stare în obiecte distincte de stare**. Contextul deleagă către starea curentă, iar tranzițiile schimbă obiectul de stare. Se elimină condiționalele repetate, codul devine clar pentru automatul finit pe care îl modelează. Este folosit în protocoale, fluxuri de autentificare, mașini de vândut sau orice logică cu stări bine definite și tranziții controlate.

Pattern-ul de proiectare ***Mediator*** **centralizează colaborarea dintre obiecte pentru a reduce cuplarea laterală**. Participanții comunică prin mediator în loc să se refere unii pe alții. Graficul de dependențe devine mai simplu, iar schimbările locale nu se propagă în rețea. În aplicații se întâlnește în sisteme de chat, panouri de control, coordonare între componente UI sau orchestrarea interacțiunilor dintr-un modul complex.

Pattern-ul de proiectare ***Strategy*** **înlocuiește condiționalele ramificate cu politici schimbabile**. Clientul depinde de o interfață, algoritmul concret se alege la configurare sau la rulare. Beneficiul este că se menține clasa stabilă, iar variația se mută într-un obiect dedicat. În testare se pot folosi dubluri pentru politică. În lumea reală apare la selectarea metodelor de plată, a strategiilor de reîncercare sau a regulilor de tarifare, unde regula se schimbă fără a atinge clasa care orchestrează.

Pattern-ul de proiectare ***Chain of Responsibility*** **aliniază mai multe *handler*-e care pot procesa o cerere sau o pot pasa mai departe**. Fiecare verigă este simplă și focalizată, iar ordinea se poate configura. Avantajul este separarea preocupărilor, diagnosticul devine observabil dacă se loghează cine a preluat cererea și unde s-a oprit lanțul. Util în validări succesive, procesarea de evenimente, *middleware*-uri HTTP sau conducte de procesare (*pipelines*) de filtrare.

Pattern-ul de proiectare ***Memento*** **captează starea internă a unui obiect pentru a o putea restaura ulterior fără a rupe încapsularea**. Este baza pentru undo sau checkpoint. Costurile țin de memorie și de numărul de memento-uri păstrate. În practică apare în editoare, aplicații cu pași multipli și în motorul de jocuri pentru salvări rapide.

Pattern-ul de proiectare ***Interpreter*** **definește o reprezentare pentru o gramatică simplă și un interpret pentru propoziții din acel limbaj**. Este util în motoare de reguli sau mini limbaje specifice domeniului, unde expresiile trebuie parcurse și evaluate. Astăzi este folosit mai rar direct, fiind înlocuit adesea de generatoare sau motoare dedicate, totuși rămâne o tehnică utilă în soluții restrânse și controlate.

1.3.3. *Pattern*-uri de proiectare creaționale (GoF)

Pattern-urile de proiectare **creaționale** descriu **felul în care sunt create obiectele și cum poate fi separat codul care folosește un obiect de codul care decide ce obiect concret trebuie instanțiat**. La prima vedere, crearea unui obiect pare o problemă simplă, deoarece în multe limbaje folosim direct un constructor. În aplicații reale, însă, alegerea implementării concrete, inițializarea dependențelor, controlul ciclului de viață și configurarea obiectului pot deveni aspecte importante de arhitectură.

În practică, *pattern*-urile creaționale **apar atunci când obiectele nu mai pot fi create simplu**, direct și peste tot cu new. *Factory Method* și *Abstract Factory* ascund alegerea implementării concrete, Builder ajută la construirea obiectelor cu mai multe opțiuni, *Prototype* pornește de la copierea unui obiect existent, iar *Singleton* controlează existența unei singure instanțe. Ideea comună este că instanțierea devine o responsabilitate separată, astfel încât restul aplicației să depindă mai mult de contracte și mai puțin de clase concrete.

Pattern-ul de proiectare ***Singleton*** **asigură existența unei singure instanțe și un punct global de acces**. Este tentant pentru resurse partajate, de exemplu registrul de configurare sau un cache. Totuși, introduce cuplare ascunsă și îngreunează testarea, deoarece logica devine dependentă de un obiect global greu de înlocuit în probe. De aceea, în practica modernă *Singleton* este adesea considerat un *anti-pattern* când este folosit ca acces global direct la stare sau servicii. Alternativa recomandată este controlul ciclului de viață printr-un container de injecție a dependențelor, expunerea prin interfețe și configurarea clară a domeniului de viață: aplicație, sesiune sau cerere.

Un **exemplu simplu** de *Singleton* este un registru de configurare. Constructorul este privat, iar accesul se face printr-o metodă statică. Utilizarea este directă:

```
public final class AppConfig {
    private static final AppConfig INSTANCE = new AppConfig();
    private String environment = "dev";

    private AppConfig() {
    }
    public static AppConfig getInstance() {
        return INSTANCE;
    }
    public String environment() {
        return environment;
    }
    public void setEnvironment(String environment) {
        this.environment = environment;
    }
}

public class Demo {
    public static void main(String[] args) {
        AppConfig config = AppConfig.getInstance();

        System.out.println(config.environment());
        config.setEnvironment("prod");
        System.out.println(AppConfig.getInstance().environment());
    }
}
```

Exemplul arată partea atractivă a *pattern*-ului: accesul este simplu, iar aplicația are **o singură instanță** a configurării. Totuși, aceeași simplitate ascunde și riscul principal. Orice cod poate ajunge la `AppConfig.getInstance()`, deci dependența nu mai este vizibilă în constructor sau în semnătura metodei. În timp, *Singleton* poate deveni echivalentul orientat spre obiecte al unei variabile globale.

Problema devine mai clară în testare. Un test care modifică mediul în `prod` poate face ca alt test să pornească cu aceeași valoare, deși se așteaptă la `dev`.

```
AppConfig.getInstance().setEnvironment("prod");

/* un alt test poate vedea tot "prod", desi astepta valoarea initiala */
String env = AppConfig.getInstance().environment();
```

De aceea, în aplicațiile moderne se preferă adesea injectarea dependențelor. În loc ca o clasă să caute singură *Singleton*-ul global, ea **primește configurarea explicit, prin constructor**.

```
interface Config {
    String environment();
}

final class DefaultConfig implements Config {
    private final String environment;

    DefaultConfig(String environment) {
        this.environment = environment;
    }
    public String environment() {
        return environment;
    }
}

final class ReportService {
    private final Config config;

    ReportService(Config config) {
        this.config = config;
    }
    public String reportHeader() {
        return "Environment: " + config.environment();
    }
}
```

În această variantă, **dependența este vizibilă**. `ReportService` nu știe dacă primește o configurare reală, una de test sau una încărcată dintr-un fișier. Ciclul de viață poate fi controlat de codul de pornire al aplicației sau de un container de DI (injectare de dependențe). Astfel se păstrează ideea de instanță controlată, dar fără acces global ascuns.

Legătura cu discuția despre **variabile globale** din **programarea structurată** este directă. În ambele cazuri, starea comună pare comodă la început, dar poate produce dependențe ascunse, efecte laterale și teste fragile. Diferența este că *Singleton* folosește mecanisme OO, constructor privat și metodă statică, dar problema arhitecturală poate rămâne aceeași: cine controlează starea, cine o poate modifica și cât de vizibilă este dependența pentru cititorul codului. Figura 1.9 sintetizează contrastul: în varianta *Singleton*, dependența este obținută prin acces global, în timp ce în varianta cu injectare de dependență (*Dependency Injection*), dependența este primită explicit și poate fi înlocuită.

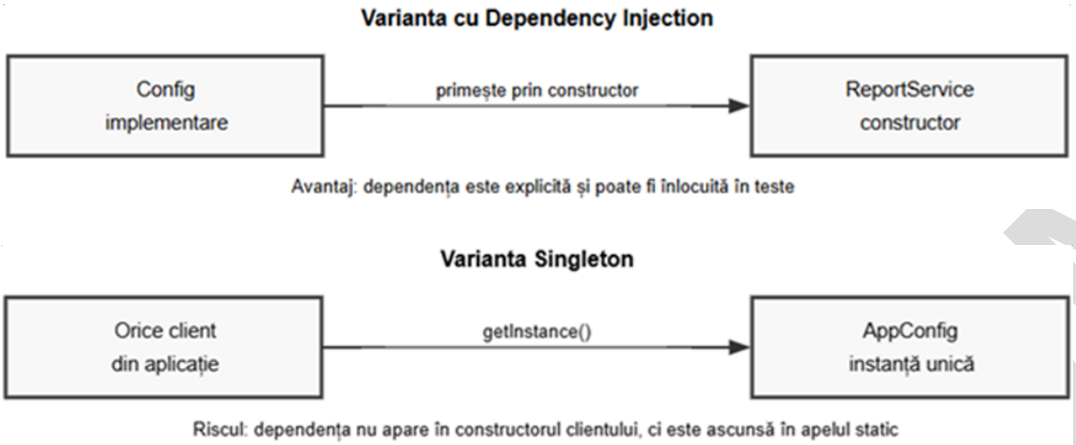


Figura 1.9. Singleton față de injectare de dependență (acces global față de dependență explicită)

Pattern-ul de proiectare ***Factory Method*** **are ca scop decuplarea codului client de detaliile instanțierii**. În loc ca un client să creeze direct un obiect concret, deleagă decizia unei metode fabrică, care returnează o instanță ce respectă o interfață. Beneficiul este flexibilitatea, alegerea implementării se poate face din configurație, la execuție sau în funcție de context, iar testarea devine mai simplă deoarece se pot furniza implementări dublură. În lumea reală apare la alegerea driverului pentru o resursă, fie sistem de stocare, fie transport de mesaje, fie codec media, clientul vorbește cu interfața, fabrica decide varianta concretă. Capcana obișnuită este fabricarea „cosmetică", doar mutarea instrucțiunii de creare într-un alt loc, fără a ascunde o regulă reală de selecție.

Pattern-ul de proiectare ***Abstract Factory*** **extinde ideea precedentă la familii de produse compatibile**. O fabrică abstractă expune metode pentru a crea mai multe obiecte care trebuie să funcționeze coerent împreună, de exemplu un set de componente UI sau o suită de adaptoare pentru același furnizor. Avantajul este coerența, clientul primește obiecte care se „potrivesc între ele" fără a cunoaște clasele concrete. În practică este util când se schimbă furnizorul unei platforme, de exemplu trecerea de la un broker de mesaje la altul, setul de producători și consumatori vine din aceeași fabrică. Costul este complexitatea suplimentară, iar capcana este proiectarea unei familii artificiale când, de fapt, variația poate fi exprimată mai simplu cu *Factory Method* sau compunere.

Pattern-ul de proiectare ***Builder*** **separă pașii de construire de obiectul final și face vizibilă intenția de inițializare**. În locul unui constructor cu mulți parametri, builderul permite setarea treptată a opțiunilor, apoi o operație `build()` produce o instanță imutabilă. Beneficiile sunt claritatea la citire, reducerea erorilor de poziționare a parametrilor și posibilitatea de a oferi valori implicite sensibile. În lumea reală se folosește pentru configurări bogate, conexiuni la baze de date, clienți HTTP sau obiecte domeniu cu multe opțiuni. Capcana este un *Builder* supra-proiectat (inutil), prin crearea unui *Builder* pentru obiecte triviale, caz în care sunt suficiente un constructor bine ales sau metode statice numite clar.

Pattern-ul de proiectare ***Prototype*** **creează obiecte noi prin clonarea unui exemplar prototip**. Este util când inițializarea este costisitoare, de exemplu încărcarea unui șablon mare sau calcularea unei structuri inițiale, clonarea replică rapid starea de pornire. În practică se întâlnește la generarea de documente pe baza unui șablon sau la instanțierea de entități cu configurații predefinite. Grija principală este definirea corectă a clonării pentru obiecte compuse, profundă sau superficială, precum și menținerea invariabilelor după copiere. Dacă clonarea devine dificilă, de multe ori o "fabrică" explicită sau un *builder* oferă control mai limpede.

1.3.4. Alte familii de *pattern*-uri de proiectare

Pattern-urile GoF rămân un **vocabular** de bază pentru proiectarea orientată spre obiecte. Pe măsură ce aplicațiile au devenit distribuite, stratificate și integrate cu infrastructuri diverse, au apărut probleme recurente care nu mai țineau doar de colaborarea dintre obiecte, care au condus la alte familii de *pattern*-uri, aplicate la niveluri mai apropiate de *middleware*, rețea, aplicații *enterprise*, integrare, persistență, mesagerie și cloud. Aceste familii nu contrazic *pattern*-urile GoF, ci extind aceeași idee: numesc probleme întâlnite des și propun forme de organizare care s-au dovedit utile în practică.

În lumea **CORBA** (*Common Object Request Broker Architecture*) și a obiectelor distribuite, accentul a fost pe concurență, rețea și transparență de locație. Aici apar *pattern*-uri de evenimente și I/O precum *Reactor* și *Proactor* care structurează buclele de evenimente și separă demultiplexarea de tratarea efectivă, utile când se gestionează mii de conexiuni. La nivel de comunicare și configurare apar *Acceptor Connector* pentru inițierea conexiunilor și refolosirea logicii de accept, respectiv *Component Configurator* pentru încărcarea dinamică a componentelor la rulare [19].

În CORBA în sine, masele de tipuri generate de IDL au dus frecvent la *Bridge* între abstracția domeniului și invocarea *remote*, la *Proxy* gestionat de ORB și la *Adapter* pentru maparea între interfețe stabilite și implementări existente. Beneficiul acestor *pattern*-uri este că fac explicită granița dintre evenimente, concurență și logică de domeniu, astfel testarea și diagnoza se concentrează pe punctele unde se schimbă contextul de execuție.

În zona **Java EE**, azi **Jakarta EE**, pe măsură ce aplicațiile au crescut, s-au cristalizat *pattern*-uri care ordonează straturile aplicației și reduc cuplarea. *Front Controller* centralizează intrările web, rutează și aplică politici comune, autentificare sau localizare. *Intercepting Filter* intercalează filtre transversale, de exemplu jurnalizare, limitare de rată (*rate limiting*) sau conversii, fără a amesteca logica în fiecare controler. Pentru a decupla codul de acces la resurse s-au folosit *Service Locator* și *Business Delegate*, primul oferă un punct de rezolvare pentru resurse și servicii, al doilea ascunde detaliile unui serviciu la distanță în spatele unei interfețe locale. În stratul de servicii, *Session Facade* agregă operații mai fine într-un contract coerent pentru clienți, reduce traficul și oferă un loc clar pentru tranzacții. Pentru persistență, *Data Access Object* și *Transfer Object* au izolat accesul la date și au redus numărul de traversări la distanță prin obiecte de transfer coezive, astăzi ideea se regăsește în *Repository* și *DTO* în aplicațiile moderne. În plus, *Composite View* și *View Helper* au disciplinat partea de prezentare, separând compoziția vizuală de logica de pregătire a datelor [20].

Aceste familii au **același obiectiv** ca GoF**, să ofere nume și forme pentru probleme recurente, dar la un nivel mai aproape de infrastructură, de protocoale și de stratificarea aplicației**. Ele rămân relevante chiar dacă tehnologiile s-au schimbat, deoarece principiile persistă, separarea preocupărilor, granițe explicite, contracte clare. De exemplu, un *Session Facade* devine azi un API consolidat în jurul unor microservicii, un *Service Locator* este înlocuit de injectare de dependențe, iar *Reactor* trăiește în platforme reactive. Pentru un începător, valoarea practică vine din a vedea cum *pattern*-urile se combină, filtre transversale, fațade pentru operații, obiecte de acces la date, și cum reduc complexitatea locală și cresc testabilitatea pe granițe bine definite.

O familie foarte importantă este cea a ***pattern*-urilor de integrare *enterprise***. Aici accentul nu mai cade pe o singură aplicație, ci pe schimbul de mesaje între aplicații. Apar *pattern*-uri precum *Message Channel, Message Router, Message Translator, Splitter, Aggregator* sau *Dead-Letter Channel*. Ele descriu probleme foarte concrete: cum transportăm un mesaj, cum alegem

destinația, cum transformăm formatul, cum împărțim un mesaj mare în mesaje mai mici, cum recompunem rezultatele și ce facem cu mesajele care nu pot fi procesate [21].

În **aplicațiile *cloud-native*** au apărut alte *pattern*-uri, legate de operarea serviciilor independente. *API Gateway* oferă o intrare comună către mai multe servicii. *Service Discovery* permite găsirea serviciilor la rulare. *Circuit Breaker* oprește temporar apelurile către un serviciu care eșuează repetat, pentru a evita propagarea problemelor [22]. *Sidecar* adaugă funcții auxiliare, de exemplu monitorizare, diagnostic sau comunicație securizată, pe lângă serviciul principal [23].

Aceste *pattern*-uri **nu înlocuiesc principiile** OO **sau *pattern*-urile** GoF, ci **le completează** la nivel operațional și distribuit. Unele ajută la proiectarea claselor, altele la organizarea straturilor unei aplicații, altele la comunicarea dintre aplicații sau operarea serviciilor în cloud. Tehnologiile se schimbă, dar principiile rămân stabile: separarea responsabilităților, granițe explicite, contracte clare, dependențe controlate și comportament observabil.

O vedere sintetică este utilă. Figura 1.10 grupează aceste familii după nivelul la care acționează, de la obiecte la cloud.

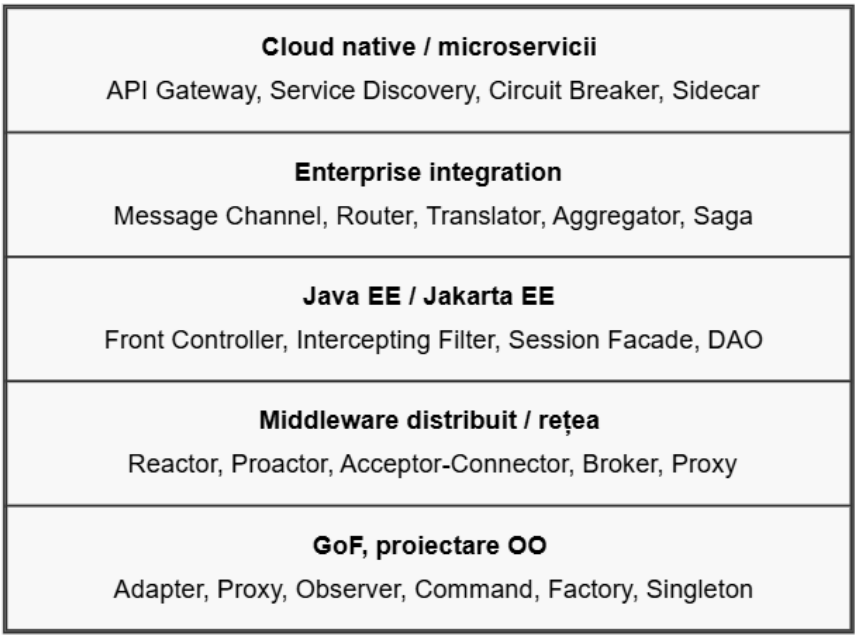


Figura 1.10. Familii de *pattern*-uri software (de la obiecte la cloud)

Se observă o **evoluție a nivelului de abstractizare**. *Pattern*-urile GoF lucrează în primul rând cu obiecte și clase. *Pattern*-urile de *middleware* adaugă preocupări de rețea, concurență și execuție distribuită. *Pattern*-urile Java EE/Jakarta EE organizează aplicații enterprise stratificate. *Pattern*-urile de integrare *enterprise* se concentrează pe mesaje, transformări, rute și procese distribuite. *Pattern*-urile *cloud-native* apar în jurul serviciilor operate independent, al rezilienței, al descoperirii serviciilor și al infrastructurii containerizate.

Pentru un **programator aflat la început, valoarea practică** nu este memorarea tuturor denumirilor. Important este să recunoască situațiile: când avem nevoie să adaptăm o interfață, când trebuie să controlăm accesul, când o cerere trebuie transformată într-o comandă, când mai multe servicii comunică prin mesaje, când un serviciu trebuie protejat de eșecul altuia. Numele *pattern*-ului vine după înțelegerea problemei, nu înaintea ei.

1.4. Orientarea spre componente

După ce programarea structurată a adus claritate prin blocuri și module, iar orientarea spre obiecte a unit datele cu operațiile într-un tip coerent, orientarea spre componente face pasul la un **nivel superior de organizare**. O **componentă** este o **unitate de livrare și de evoluție**, are **granițe clare** și se tratează ca o **piesă schimbabilă**, cu *contract* public stabil, *implementare* ascunsă și *versiune* explicită [24]. Ideea continuă ascunderea informației și modularizarea discutate la programarea structurată, dar le aplică asupra unor „pachete" care pot fi dezvoltate, testate, distribuite și înlocuite independent. Practic, componenta devine moneda de schimb între echipe, instrumentul prin care se controlează dependențele, se stabilește compatibilitatea și se planifică migrațiile. În jurul acestor piese apar deseori un „model de componente" și un „container" care oferă ciclul de viață, injectarea dependențelor, descoperirea și configurarea la rulare, iar pentru ecosisteme mari se adaugă registru, semnare și politici de publicare.

1.4.1. Conceptul de componentă software

O componentă se recunoaște prin trei elemente nedespărțite. Mai întâi **contractul** public, adică ceea ce consumatorii pot vedea și folosi, **interfețe** și **API-uri** (*Application Programming Interface*) documentate, tipuri de date expuse, erori posibile și promisiuni non-funcționale, de exemplu limite, latențe, reguli de compatibilitate. Apoi **implementarea**, tot ceea ce realizează efectiv comportamentul, cod, algoritmi, optimizări, structuri de date, care rămân ascunse în spatele contractului și se pot schimba fără să rupă consumatorii, atâta vreme cât contractul rămâne valabil. În fine **versiunea**, un identificator care fixează starea publică a contractului și permite managementul schimbării, de regulă prin *semantic versioning* (*major*, *minor*, *patch*), unde creșterea majoră semnalează ruperea compatibilității, cea minoră adaugă capabilități compatibile înapoi, iar *patch* corectează defecte fără a schimba contractul [25].

Privită din exterior, **o componentă ar trebui să fie ușor de folosit fără a-i cunoaște detaliile interne**. Consumatorul depinde de contract, nu de implementare. Producătorul componentei poate schimba algoritmii, structurile interne sau optimizările, atâta vreme cât păstrează contractul public. Figura 1.11 sintetizează această separare dintre contractul vizibil, implementarea ascunsă și versiunea publică a componentei. Figura 1.12 arată utilizarea componentei prin contract.

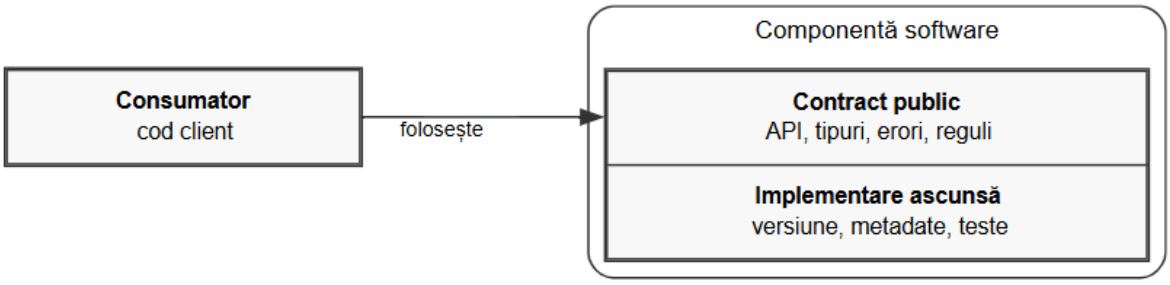


Figura 1.11. Componenta software (contract public, implementare ascunsă, versiune)

Cele două diagrame pun accent pe separarea dintre utilizare și implementare. **Consumatorul cunoaște** API-ul, tipurile expuse și regulile de compatibilitate. **Implementarea** rămâne internă componentei și **poate evolua controlat**, iar versiunea indică starea publică a contractului și ajută la gestionarea schimbărilor.

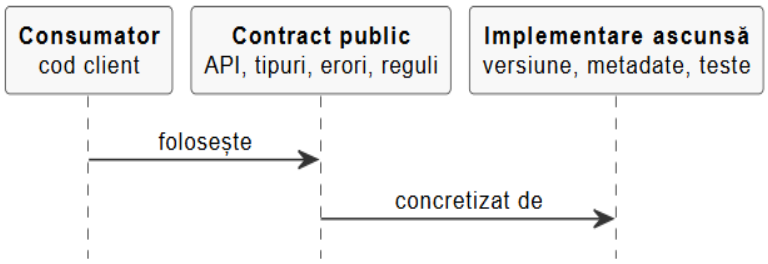


Figura 1.12. Interacțiunea cu o componentă software (utilizare prin contract public)

Din perspectiva utilizatorului, **componenta se importă și se folosește prin contract**, fără cunoașterea detaliilor interne. Din perspectiva producătorului, *componenta se construiește, se testează și se livrează ca unitate*, de exemplu **JAR** pentru o bibliotecă, pachet ***npm*** pentru *frontend*, sau chiar un microserviciu pentru granițe mai groase, ideea de componentă rămâne aceeași, contract stabil, implementare ascunsă, versiune controlată.

Când două componente colaborează, **dependența se exprimă la nivel de contract, nu la nivel de detaliu intern**, astfel **migrațiile devin previzibile**, se poate declara „acceptă versiuni 1.x” sau „acceptă versiunea 2.3 și mai noi (compatibile înapoi)”, iar sistemele de *build* pot verifica compatibilitatea de binar și de surse.

Un contract sănătos separă API de SPI (*Service Provider Interface*). **API** descrie cum este folosită componenta de către clienți, **SPI** descrie cum pot furnizorii *pluggable* (care pot fi conectați sau înlocuiți la configurare, fără a schimba codul existent) să extindă componenta cu implementări proprii, de exemplu un driver de stocare sau un codec. Această separare permite ecosistemelor să crească prin *plug-in*, fără a modifica nucleul.

La execuție (*runtime*), un *container* sau un mediu de rulare (*runtime*) de componente poate oferi ciclul de viață, injectare de dependențe, descoperirea prin nume sau prin clasificări și conectarea la evenimente, astfel **componenta rămâne concentrată pe responsabilitatea ei principală**, iar „infrastructura” se ocupă de fire, conexiuni, configurare, monitorizare și diagnostic.

Diferența dintre **API** și **SPI** poate fi înțeleasă ca diferența dintre **felul în care componenta este folosită** și **felul în care poate fi extinsă**. API-ul este orientat spre consumatori, SPI-ul este orientat spre furnizorii de implementări. Figura 1.13 arată structura acestei separări, iar Figura 1.14 arată interacțiunea dintre client, componentă și furnizorul unei extensii.

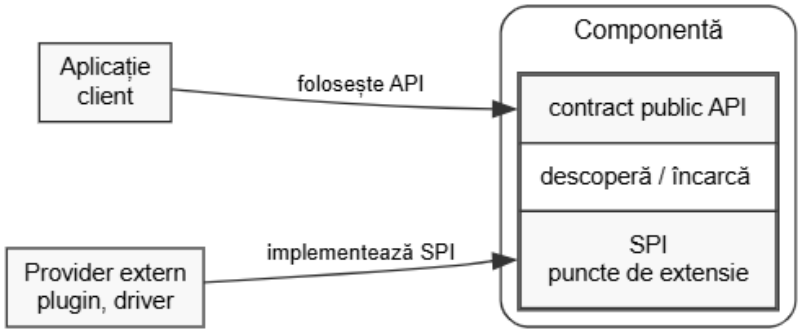


Figura 1.13. Structura relației API și SPI într-o componentă (utilizare față de extensie)

Prin API, aplicația client **folosește** componenta. Prin SPI, furnizori externi **pot adăuga** implementări noi, de exemplu un driver, un codec, un conector sau o politică de stocare. Această separare permite extensibilitate fără modificarea nucleului componentei.

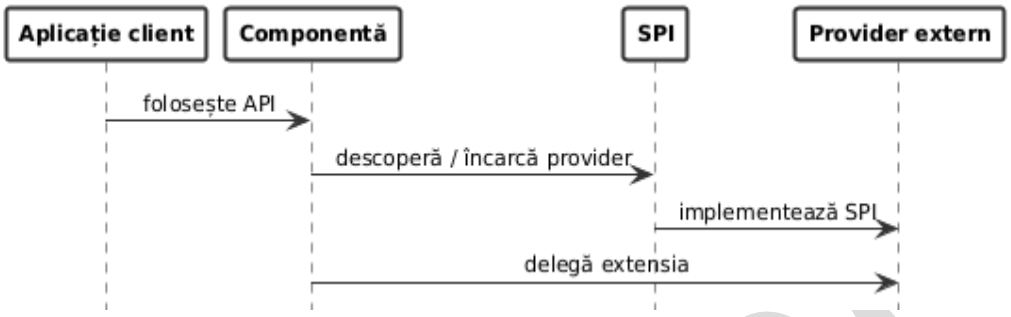


Figura 1.14. Interacțiunea în relația API și SPI (apel prin API, extensie prin SPI)

Pentru consumatori contează în primul rând **compatibilitatea înapoi** a contractului. Adăugarea unei operații noi este, de obicei, compatibilă, schimbarea semnăturii unei operații existente sau eliminarea ei nu este, de aceea schimbările de contract cer versiuni majore, iar proiectarea contractului se face cu grijă pentru a permite extensii viitoare. În plus, politica de erori și regulile non-funcționale sunt parte a contractului, dacă un apel care era local devine la distanță printr-un *proxy*, contractul ar trebui să reflecte așteptările realiste, timpi de răspuns și erori de rețea, altfel consumatorii sunt surprinși de costuri invizibile.

În fine, o componentă bună vine însoțită de **metadate** și de artefacte de calitate, descrierea versiunii, dependențele declarate, *changelog*, ghid de migrare, și un set minim de teste de contract care poate fi rulat de consumatori, pentru a confirma că integrarea este asumată. În acest fel, orientarea spre componente **nu este doar o formă de ambalare**, ci o **disciplină de arhitectură** care face posibilă evoluția controlată a sistemelor mari, cu echipe multe și ritm de livrare susținut.

Pentru exemple istorice de model de componente și container, proprietăți, evenimente și cicluri de viață, sunt utile specificațiile *JavaBeans* și *Jakarta Enterprise Beans* [26], [28].

1.4.2. *JavaBeans*, proprietăți, evenimente, introspecție

Modelul ***JavaBeans* a standardizat** ideea de **componentă ușoară în Java**, o clasă cu constructor fără parametri, proprietăți accesibile prin metode de acces cu nume convenționale, `getX()`, `setX()`, sau `isX()` pentru `boolean`, evenimente livrate prin mecanismul de ascultători, și introspecție, adică posibilitatea de a descoperi la rulare proprietățile și evenimentele expuse [26]. Scopul este simplu, unelte, biblioteci și *framework*-uri pot manipula componente fără a cunoaște implementarea, doar urmând convențiile. În practică, un *JavaBean* expune proprietăți clare, anunță schimbările prin evenimente și se lasă inspectat de unelte sau de cod generic.

Mai jos este un **exemplu compact**. Prima clasă este un *bean* simplu, un comutator cu o proprietate `on`. Respectă convențiile pentru metodele de acces, expune metode de înregistrare a ascultătorilor și notifică atunci când starea se schimbă. A doua parte arată un mic program care folosește *bean*-ul, ascultă evenimentul și face introspecție pentru a lista proprietățile.

```
import java.beans.*;

// un bean simplu, cu proprietate "on" si suport de evenimente
public class SwitchBean {
    private boolean on;
```

```
    private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);

    public SwitchBean() { } // constructor fara parametri, tipic JavaBeans

    // metode de acces standard pentru proprietatea "on"
    public boolean isOn() { return on; }
    public void setOn(boolean newValue) {
        boolean old = this.on;
        this.on = newValue;
        // notific evenimentul de schimbare a proprietatii
        pcs.firePropertyChange("on", old, newValue);
    }

    // inregistrare si dezabonare de la evenimente
    public void addPropertyChangeListener(PropertyChangeListener l)
        { pcs.addPropertyChangeListener(l); }
    public void removePropertyChangeListener(PropertyChangeListener l)
        { pcs.removePropertyChangeListener(l); }
}

import java.beans.*;

public class DemoBeans {
    public static void main(String[] args) throws Exception {
        // 1) folosire proprietate si ascultare evenimente
        SwitchBean sw = new SwitchBean();
        sw.addPropertyChangeListener(evt -> {
            if ("on".equals(evt.getPropertyName())) {
                System.out.println("on changed: " + evt.getOldValue() + " -> "
                + evt.getNewValue());
            }
        });
        sw.setOn(true);   // declanseaza eveniment
        sw.setOn(false);  // declanseaza eveniment

        // 2) introspectie, listarea proprietatilor descoperite din conventii
        BeanInfo info = Introspector.getBeanInfo(SwitchBean.class);
        for (PropertyDescriptor pd : info.getPropertyDescriptors()) {
            System.out.println("property: " + pd.getName() + " type: "
            + pd.getPropertyType());
        }
    }
}
```

Proprietatea `on` este vizibilă pentru unelte deoarece numele metodelor respectă convențiile *JavaBeans*, `isOn()` și `setOn()`. Când `setOn` schimbă starea, trimite un eveniment de tip `PropertyChangeEvent` prin `PropertyChangeSupport`, astfel codul care s-a abonat primește notificări coerente și nu are nevoie să sondeze periodic starea. **Introspecția** folosește `Introspector.getBeanInfo()`, care parcurge reflecția și convențiile pentru a identifica proprietățile, evenimentele și metodele, apoi returnează descrieri care pot fi folosite de unelte grafice sau de cadre de lucru [27]. În acest fel, *JavaBeans* **oferă un contract minim, clar și util**, proprietăți prin metode de acces standard, evenimente pentru schimbări, introspecție pentru descoperire, iar implementarea internă rămâne ascunsă și liberă să evolueze.

Exemplul arată **cum ideea de *Observer* devine o convenție de componentă**: schimbarea unei proprietăți nu este doar o modificare internă, ci un eveniment observabil de către codul care s-a abonat. În acest fel, *JavaBeans* leagă încapsularea OO de un model simplu de componentă, în care proprietățile, evenimentele și introspecția formează contractul utilizabil de unelte și *framework*-uri.

1.4.3. EJB (*Enterprise JavaBeans*)

Enterprise JavaBeans (**EJB**) a fost **primul model de componente *enterprise* larg adoptat în ecosistemul Java**, el a încercat să **standardizeze în mod riguros separarea** dintre logica de *business* și infrastructură. Ideea centrală a fost că **dezvoltatorii scriu componente declarative**, iar **containerul EJB se ocupă de ciclul de viață, tranzacții, securitate, concurență și configurare** [28].

EJB a dus **ascunderea informației și modularizarea la nivel *enterprise***, printr-un model clar de componente, descriptor de lansare în producție (*deployment descriptor*), contracte bine definite și un *runtime* puternic.

Figura 1.15 arată structura unei componente EJB în container.

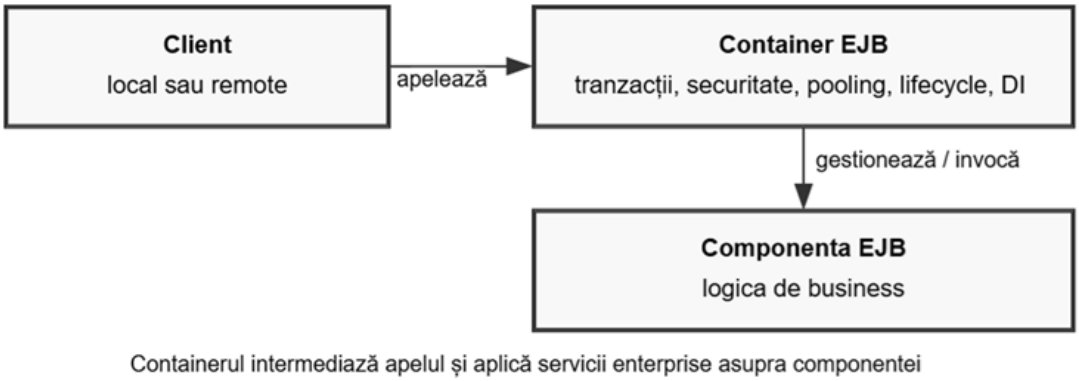


Figura 1.15. Structura unei componente EJB în container (logică de *business* și servicii *enterprise*)

Figura 1.16 prezintă interacțiunea dintre client, container și componentă.

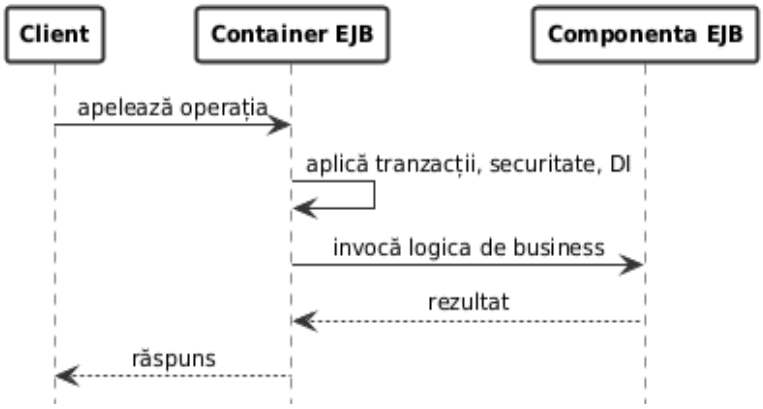


Figura 1.16. Interacțiunea cu o componentă EJB în container (apelul clientului prin serviciile containerului)

Figurile evidențiază aceeași separare a responsabilităților din două perspective. **Componenta** conține logica de *business*, iar **containerul** aplică serviciile *enterprise* din jurul ei: tranzacții, securitate, ciclul de viață, concurență și injectare de dependențe. Clientul nu apelează direct infrastructura, ci folosește componenta prin mecanismele oferite de container. Astfel, **preocupările transversale sunt tratate uniform**, iar **codul componentei rămâne concentrat pe regulile aplicației**.

Session Beans au reprezentat componentele care **executau logica de *business* a aplicației**. În varianta *Stateless* containerul gestionează un grup (*pool*) de instanțe care servesc cereri independente, se maximizează scalarea și se minimizează starea. În varianta *Stateful* componenta păstrează context între apeluri pentru un client, util pentru conversații scurte și coerente, de exemplu un mic asistent de achiziție. În ambele cazuri containerul atașează tranzacții declarative, de exemplu *Required* sau *RequiresNew*, aplică securitate declarativă pe metode și face injectarea dependențelor. Lecția de arhitectură este separarea clară a logicii de *business* de preocupările transversale, prin politici declarative și un *runtime* care le aplică uniform.

În EJB 3, forma de bază a unui *Session Bean* a devenit mult mai simplă decât în versiunile timpurii. **Un exemplu minimal de componentă *Stateless*** poate arăta astfel:

```
import jakarta.ejb.Stateless;
@Stateless
public class AccountService {
    public boolean transfer(long fromId, long toId, double amount) {
        if (amount <= 0 || fromId == toId) {
            return false;
        }
        // aici ar fi apelata logica de debitare si creditare
        return true;
    }
}
```

În acest exemplu, `AccountService` exprimă logica de *business*, iar adnotarea `@Stateless` spune containerului că **instanțele nu păstrează stare conversațională între apeluri**. Containerul poate gestiona grupul (*pool*) de instanțe, ciclul de viață și tranzacțiile, fără ca aceste detalii să fie scrise explicit în fiecare metodă. Dacă vrem să separăm și mai clar contractul de implementare, putem introduce o **interfață locală**. Interfața descrie operațiile disponibile pentru clienții din aceeași aplicație sau din același container, fără să expună detaliile clasei concrete.

```
import jakarta.ejb.Local;

@Local
public interface AccountOperations {
    boolean transfer(long fromId, long toId, double amount);
}
```

Adnotarea **`@Local`** arată că interfața este destinată apelurilor locale, în interiorul aplicației *enterprise*. Clientul poate depinde de `AccountOperations`, dar nu de clasa concretă `AccountService`. Implementarea vine separat și este gestionată de container.

```
import jakarta.ejb.Stateless;

@Stateless
public class AccountService implements AccountOperations {
    public boolean transfer(long fromId, long toId, double amount) {
        if (amount <= 0 || fromId == toId) {
            return false;
        }
        return true;
    }
}
```

În această variantă, `AccountOperations` este contractul, iar `AccountService` este implementarea. Ideea este aceeași ca în exemplele anterioare cu interfețe și injectare de dependențe, dar aplicată la **nivel *enterprise***. Clientul lucrează cu interfața, containerul creează și gestionează componenta, iar infrastructura poate aplica servicii precum tranzacții, securitate etc. Astfel, componenta nu mai este doar o clasă instanțiată manual, ci o unitate controlată de *runtime*.

Entity Beans, în versiunile timpurii **EJB 2.x**, au încercat să modeleze **persistența la nivel de componentă**. S-au definit două stiluri, *Bean Managed Persistence* în care componenta scrie manual SQL, respectiv *Container Managed Persistence* în care containerul genera implicit accesul la date. În practică, modelul a devenit greu, cu cicluri de viață complexe și portabilitate dificilă. Lecția învățată a dus la apariția **JPA** (*Java Persistence API*), un **standard separat pentru maparea obiect relațională**, care a simplificat dramatic persistența. În proiectarea modernă, persistența este o preocupare dedicată, separată de componenta de *business*, iar EJB a renunțat treptat la rolul de a o controla direct.

Message-Driven Beans au adus **integrarea asincronă** în modelul de componente. Un **MDB** se abonează la o **coadă** sau un **topic JMS**, primește mesaje și rulează logică de procesare în cadrul containerului, cu aceleași garanții de tranzacții și securitate ca un *Session Bean*. Acest model a încurajat **decuplarea prin mesaje**, reîncercări și procesare concurentă controlate. Pentru sisteme scalabile, se potrivesc un canal de mesaje clar definit și o componentă dedicată care procesează asincron, nu doar apeluri sincrone de tip **RPC**.

Un element definitoriu al EJB a fost **contractul clar** dintre componentă și container. În versiunile timpurii, acest contract era exprimat prin interfețe locale și la distanță, *home* și *remote*, plus descriptor de lansare în producție (*deployment descriptor*). În **EJB 3.x**, modelul s-a simplificat prin **stil POJO** (*Plain Old Java Object*) și **adnotări**. Dezvoltatorul declara prin metadate ce are nevoie, tranzacții, securitate, timere, cicluri de viață, iar containerul furniza aceste servicii. Lecția rămâne importantă: politicile declarative și contractele clare reduc erorile sistematice și fac depanarea mai predictibilă.

Totuși, modelul clasic EJB a venit și cu **costuri** reale: servere de aplicații grele, configurări verbose în versiunile timpurii, cicluri lente de livrare și portabilitate variabilă între furnizori. Comunitatea a răspuns prin simplificare. EJB 3.x a introdus **adnotări, injectare de dependențe, interfețe mai simple** și integrare naturală cu **JPA**. În paralel, *framework*-uri mai ușoare precum Spring au popularizat abordarea POJO. Direcția generală a ecosistemului a devenit clară: infrastructura trebuie să fie puternică, dar cât mai puțin invazivă pentru codul aplicației. Privind înapoi, EJB a sedimentat câteva idei care rămân valide: politici declarative pentru tranzacții și securitate, separarea componentelor *stateless* de cele *stateful*, standardizarea contractelor la granițe și integrarea prin mesaje pentru decuplare. În arhitecturile actuale, aceste idei apar în ***Jakarta EE*, JPA, *MicroProfile*, containere ușoare, microservicii, interceptori, filtre** și **mesagerie**. Forma s-a schimbat, dar principiul de bază a rămas: logica de *business* trebuie să rămână cât mai curată, iar preocupările transversale trebuie declarate și aplicate consecvent de *runtime*.

1.5. *Framework*-uri și inversarea controlului (IoC)

În ingineria software bazată pe obiecte sau componente s-a arătat în mod repetat, prin studii industriale și experiență acumulată, că **riscul de eșec al proiectelor scade** pe măsură ce **arhitectura este mai bună, explicită și consecventă**. *Framework*-urile apar tocmai din această nevoie, ele oferă „**prefabricate arhitecturale**”, adică un **schelet de aplicație validat**, care deține controlul execuției și oferă puncte clare de extensie.

Spre deosebire de o bibliotecă pe care o apelează codul scris de programator, **într-un *framework* codul scris de programator este chemat la momentul potrivit de cadrul de execuție**, ceea ce înseamnă inversarea controlului (IoC) [29]. Rezultatul este un **nou nivel de reutilizare**. Nu doar clase și funcții ca în OO și nu doar pachete livrabile ca în orientarea spre componente, ci și **reutilizarea de fluxuri, convenții și politici transversale**, ceea ce reduce variația accidentală, accelerează livrarea și crește șansele de reușită ale proiectului.

1.5.1 Prefabricate arhitecturale plus extensie prin componente

Un *framework* **oferă un traseu gata făcut pentru lucruri greu de făcut corect de fiecare dată**: inițializare, configurare, gestionarea erorilor, rutare de cereri, serializare, tranzacții, securitate, observabilitate. În loc să fie reinventat „scheletul", este extins prin componentele programatorului. Beneficiile sunt clare: consistență între proiecte, timp mai scurt până la prima versiune, facilități transversale aplicate uniform, de exemplu autentificare, jurnalizare și colectare de metrici, și o zonă de testare mai mică în codul aplicației. În acest context, jurnalizarea înseamnă scrierea unor mesaje de diagnostic într-un *log* (jurnal), iar metricile sunt valori măsurabile, cum ar fi numărul de cereri, durata unui apel sau rata de erori. Jurnalizarea, metricile și urmărirea parcursului cererilor formează baza observabilității, adică posibilitatea de a înțelege ce face sistemul în timpul execuției.

Figura 1.17 sintetizează diferența: **biblioteca este subordonată codului programatorului**, pe când ***framework*-ul cheamă codul programatorului în „cârlige" (*hooks*) bine definite**, de exemplu un controler web, un *handler* de evenimente, un interceptor.

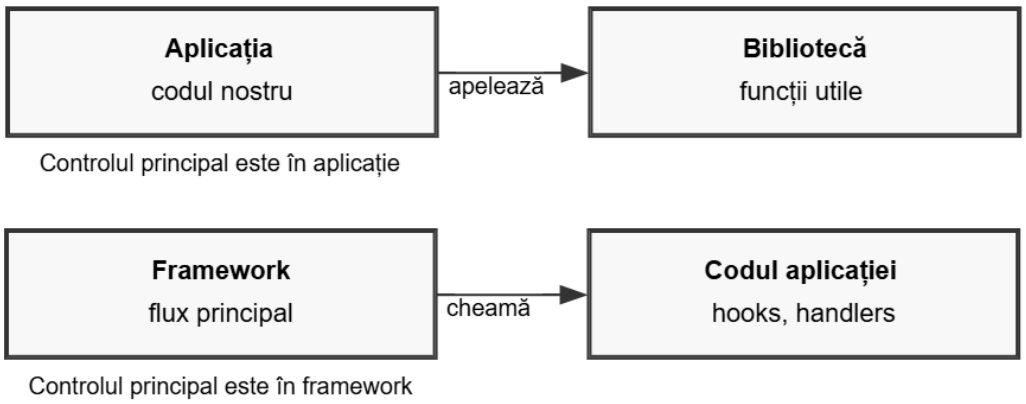


Figura 1.17. Bibliotecă față de *framework* (apel direct față de inversarea controlului)

În cazul unei biblioteci, aplicația decide când apelează funcțiile. În cazul unui *framework*, **fluxul principal este deja organizat**, iar **codul aplicației este chemat în puncte de extensie**, de exemplu controlere, *handler*-e, interceptori sau metode ale ciclului de viață.

Extensia se face în două feluri. Prin **compunere**, adaugi obiecte care implementează interfețele cerute de *framework*, strategie de autentificare, convertor de mesaje, validator, avantajul fiind decuplarea și testarea ușoară. Prin **moștenire**, specializezi clase de bază oferite de *framework*, avantajul este integrarea rapidă în fluxul standard, dar ierarhiile pot deveni rigide, iar schimbările în clasa de bază te pot afecta. Ca regulă, compunerea și programarea spre interfețe sunt preferate pentru variații de politică, moștenirea rămâne utilă pentru a completa schelete concepute anume pentru a fi extinse.

În **Android**, de exemplu, *framework*-ul deține „bucla principală" și apelează componentele scrise de programator la momente bine definite [30].

```
// MainActivity.java
public class MainActivity extends AppCompatActivity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState); // framework-ul cheama onCreate()
        setContentView(R.layout.activity_main);
        // programatorul completeaza logica
    }
}
```

Android stabilește pașii universali (inițializare, restaurare stare, oprire etc), iar programatorul extinde scheletul în „cârligele" (*hooks*) potrivite. Beneficiul este un traseu validat arhitectural, inițializare și oprire coerente, erori transversale tratate uniform.

Același mecanism apare și în ***framework*-uri web**, ***framework*-uri de testare** sau ***framework*-uri *enterprise***. Programatorul nu scrie bucla principală a serverului web, ci metode care răspund la cereri. Nu scrie motorul de execuție al testelor, ci metode marcate ca teste. Nu scrie manual toate regulile de tranzacționare, ci declară unde se aplică ele. În toate aceste cazuri, *framework*-ul oferă traseul general, iar aplicația completează punctele variabile.

1.5.2 Inversarea controlului (IoC) și injectarea de dependențe (DI)

Inversarea controlului (IoC) înseamnă că fluxul principal aparține *framework*-ului, care creează obiecte, le leagă între ele și le apelează. Tehnica practică este **injectarea de dependențe** (DI), prin care containerul creează dependențele și le injectează acolo unde sunt declarate. Există trei forme uzuale, **prin constructor** (cea mai clară și testabilă), **prin metodă *setter*** (utilă pentru dependențe opționale), **prin interfață specifică *framework*-ului** (de evitat în codul de domeniu deoarece introduce cuplare) [29]. DI crește testabilitatea, într-un test se injectează dubluri ale dependențelor fără a schimba codul.

Un **exemplu minimal de DI** se poate construi pornind de la un serviciu care are nevoie să trimită notificări. **Varianta mai slabă** este ca **serviciul să își creeze singur dependența concretă**.

```
final class EmailNotifier {
    void send(String text) {
        System.out.println("Email: " + text);
    }
}
final class OrderServiceBad {
    private final EmailNotifier notifier = new EmailNotifier();
    void placeOrder() {
        notifier.send("Order placed");
    }
}
```

În această variantă, `OrderServiceBad` este legat direct de `EmailNotifier`. Dacă vrem să schimbăm mecanismul de notificare, de exemplu SMS, broker de mesaje sau dublură de test, trebuie să modificăm clasa. **Cu DI, dependența devine explicită** și este **primită din exterior**.

```
interface Notifier {
    void send(String text);
}
final class EmailNotifier implements Notifier {
    public void send(String text) { System.out.println("Email: " + text);
    }
```

```
}
final class OrderService {
    private final Notifier notifier;
    OrderService(Notifier notifier) {
        this.notifier = notifier;
    }
    void placeOrder() {
        notifier.send("Order placed");
    }
}
```

Utilizarea poate fi ca în exemplul simplu care urmează:

```
public class Demo {
    public static void main(String[] args) {
        Notifier notifier = new EmailNotifier();
        OrderService service = new OrderService(notifier);
        service.placeOrder();
    }
}
```

Într-un *framework* cu DI, crearea și legarea obiectelor ar fi făcute de container. Important este însă principiul: **`OrderService` nu mai depinde de o clasă concretă, ci de interfața `Notifier`. În test, putem injecta o implementare falsă**.

```
final class FakeNotifier implements Notifier {
    boolean called;
    public void send(String text) {
        called = true;
    }
}
```

Această formă face dependențele vizibile în constructor, simplifică testarea și reduce cuplarea. Este aceeași idee pe care am întâlnit-o la programarea spre interfețe, doar că acum este susținută de **un *framework* sau** de **un container** care **poate crea și conecta obiectele automat**. Figura 1.18 compară varianta în care dependența este creată direct în clasă cu varianta în care ea este primită explicit din exterior.

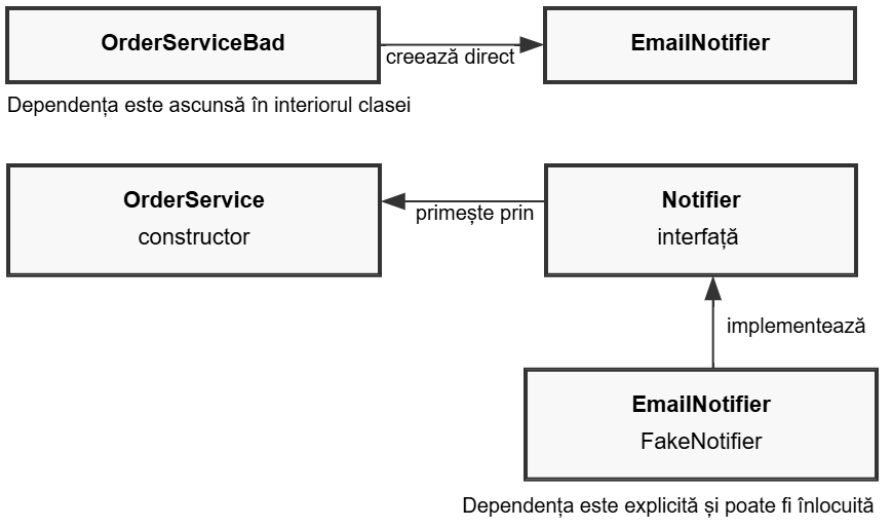


Figura 1.18. *Dependency Injection* (dependență creată direct față de dependență primită explicit)

Diferența esențială este vizibilitatea dependenței. În varianta **fără DI, clasa ascunde în interior decizia de creare** a colaboratorului. În varianta **cu DI, colaboratorul este primit prin constructor sau printr-un mecanism controlat de *framework***, ceea ce **permite înlocuirea** lui **în teste** sau **la configurare**.

Reluând exemplul de **Android**, acesta este **IoC în practică**: programatorul nu scrie cod care cheamă ciclul de viață, ci *framework*-ul cheamă metodele scrise de programator la momentele potrivite.

```
// MainActivity.java
public class MainActivity extends AppCompatActivity {
    private TextView txtInfo;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);   // framework-ul apeleaza onCreate
        setContentView(R.layout.activity_main); // logica scrisa de programator
        txtInfo = findViewById(R.id.txtInfo);   // leaga codul Java
        txtInfo.setText("onCreate()");          // de elementul din XML
    }
}
```

Codurile XML asociate:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:padding="16dp"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/txtInfo"
        android:text="Ready"
        android:textSize="18sp"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <Button
        android:id="@+id/btnAction"
        android:text="Action"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

Fișierul **XML** descrie **structura interfeței grafice**, nu fluxul de execuție. Prin `setContentView(R.layout.activity_main)`, ***framework*-ul Android** încarcă această descriere, creează obiectele vizuale corespunzătoare și le atașează activității. Identificatorul `@+id/txtInfo` devine accesibil în cod prin `R.id.txtInfo`, iar `findViewById()` permite legarea codului Java de elementul definit în XML. Astfel, programatorul completează punctele de extensie oferite de *framework*, iar acesta controlează momentul încărcării, creării și afișării interfeței.

IoC și DI nu sunt valoroase doar pentru că reduc numărul de instrucțiuni `new`. Valoarea lor reală este arhitecturală: fac dependențele vizibile, separă politica de creare a obiectelor de logica de *business* și permit înlocuirea implementărilor fără modificarea codului client. Folosite cu măsură, ele duc la aplicații mai testabile și mai ușor de evoluat. Folosite excesiv, pot ascunde fluxul real al aplicației în configurări greu de urmărit. Ca și în cazul EJB, lecția rămâne aceeași: infrastructura trebuie să ajute codul, nu să îl îngreuneze.

1.6. MVC (*Model–View–Controller*) și familia sa

După obiecte, componente și *framework*-uri, s-a impus **separarea** arhitecturală clară a responsabilităților între **date**, **prezentare** și **controlul fluxului**. *Model--View--Controller* (MVC) este un *pattern* architectural mai avansat, care organizează aplicația astfel: ***Model*** păstrează starea și regulile de domeniu, ***View*** se ocupă de afișare, ***Controller*** primește intrări și coordonează pașii [31]. Beneficiul este reducerea cuplării și a complexității. Testele pentru *Model* rămân independente de cele pentru interfață, iar schimbările de UI nu ating regulile de domeniu.

În practică, familia include variații pentru platforme moderne, MVC pentru web cu routing și șabloane, MVP cu un prezentator care împinge datele către un *View* pasiv, MVVM cu un *ViewModel* care expune stare observabilă, fiecare parte are un rol clar, granițele sunt explicite, iar evoluția se face local, fără efecte neintenționate.

1.6.1. MVC și separarea responsabilităților

***Model–View–Controller* organizează aplicația în trei părți cu roluri distincte**, *Model* reține starea și regulile de domeniu, *View* prezintă informația pentru utilizator, *Controller* primește intrări și coordonează pașii. Ideea centrală este că prezentarea, semnificația datelor și coordonarea interacțiunii rămân separate. Astfel apare o hartă clară a sistemului, ceea ce reduce cuplarea și face codul mai ușor de schimbat și testat.

Model este partea unde stau datele și logica de domeniu. Modelul știe să aplice reguli, de exemplu să comute starea unui comutator sau să valideze o comandă, dar nu știe nimic despre butoane, ferestre sau HTML. Dacă apare o regulă nouă, ea se scrie în *Model*, nu în ecran. Avantajul este dublu, același *Model* poate fi folosit cu mai multe interfețe, iar testele de corectitudine se pot scrie fără UI, direct pe metodele modelului.

View este partea unde se decide cum se afișează starea modelului. *View* primește date deja pregătite și le prezintă, text, tabel, grafic. *View* nu aplică reguli de domeniu și nu modifică modelul direct, rolul ei este să arate clar ceea ce i se dă. Când se schimbă designul, culorile sau șablonul, se lucrează în *View*, modelul rămâne neatins. De aceea echipele pot lucra în paralel, un grup ajustează prezentarea, altul ajustează regulile.

Controller este partea unde se traduc intrările utilizatorului în acțiuni asupra modelului și se alege ce *View* se afișează. *Controller*-ul citește un click, o tastă sau o cerere HTTP, invocă operația potrivită în *Model*, apoi cere *View*-ului să prezinte rezultatul. *Controller*-ul nu conține reguli de domeniu, ci doar coordonează pașii, validează parametrii la intrare și gestionează erorile la ieșire. Dacă apare o nouă rută sau un nou buton, se adaugă logică în *Controller*, nu în *Model*.

Fluxul informațiilor este simplu: intrarea ajunge la *Controller*, *Controller*-ul validează parametrii, cere modelului să aplice regula, primește starea actualizată, apoi solicită *View*-ului să o afișeze. Această ordine simplă permite localizarea problemelor. Dacă rezultatul este greșit, se verifică întâi regula din *Model*. Dacă nu se vede nimic, se verifică *View*. Dacă se declanșează acțiunea greșită, se verifică *Controller*-ul. Fiecare pas are o responsabilitate clară, ceea ce scurtează depanarea.

Figura 1.19 arată structura modelului MVC, adică separarea statică a responsabilităților între cele trei roluri, *View*, *Controller* și *Model*. Figura 1.20 completează această perspectivă prin dimensiunea dinamică: interacțiunea dintre *View*, *Controller* și *Model*, ordinea în care o intrare este preluată, transformată într-o acțiune asupra modelului și apoi reflectată în afișare.

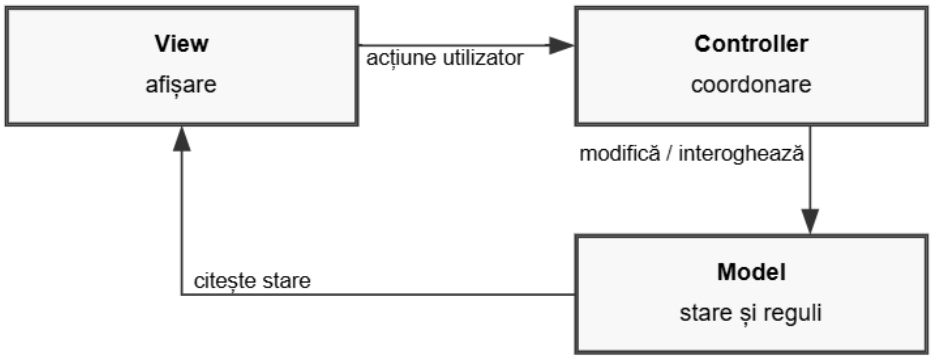


Figura 1.19. Structura *pattern*-ului *Model–View–Controller* (separarea afișării, coordonării și regulilor)

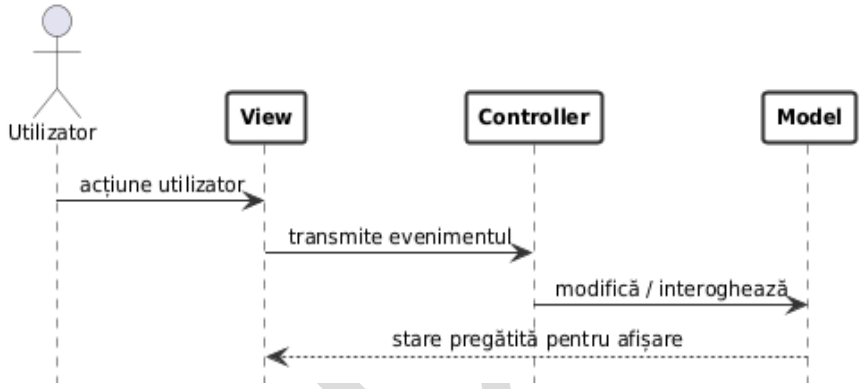


Figura 1.20. Interacțiunea în *pattern*-ul *Model–View–Controller* (fluxul dintre *View*, *Controller* și *Model*)

Putem vedea **MVC ilustrat pe un exemplu Java** foarte simplu. Modelul conține starea și regulile, *View*-ul afișează starea, iar *Controller*-ul primește comenzi și decide ce operații se aplică asupra modelului.

Modelul nu știe nimic despre consolă, ferestre, butoane sau HTTP. El știe doar regulile domeniului: cum se schimbă starea comutatorului.

```
// Model
final class SwitchModel {
    private boolean on;

    boolean isOn() {
        return on;
    }
    void turnOn() {
        on = true;
    }
    void turnOff() {
        on = false;
    }
    void toggle() {
        on = !on;
    }
}
```

***View*-ul** nu modifică Modelul. Primește o valoare și o afișează. Într-o aplicație reală, acest *View* ar putea fi înlocuit cu o interfață grafică, o pagină web sau o reprezentare JSON.

```
// View
final class SwitchView {
    void render(boolean on) {
        System.out.println("Stare comutator: " + (on ? "ON" : "OFF"));
    }
}
```

***Controller*-ul** traduce comanda primită în operații asupra modelului și apoi cere *View*-ului să afișeze rezultatul. El nu conține regula internă a comutării, ci doar coordonează pașii.

```
// Controller
final class SwitchController {
    private final SwitchModel model;
    private final SwitchView view;

    SwitchController(SwitchModel model, SwitchView view) {
        this.model = model;
        this.view = view;
    }

    void handle(String command) {
        if ("on".equals(command)) {
            model.turnOn();
        } else if ("off".equals(command)) {
            model.turnOff();
        } else if ("toggle".equals(command)) {
            model.toggle();
        }
        view.render(model.isOn());
    }
}
```

În programul `Demo`, cele trei roluri MVC sunt conectate manual. Se creează mai întâi modelul, care păstrează starea comutatorului, apoi *View*-ul, care știe să afișeze starea, și *Controller*-ul, care primește referințe către ambele. Apelurile `controller.handle("on")` și `controller.handle("toggle")` simulează două acțiuni ale utilizatorului. *Controller*-ul interpretează comanda, modifică modelul și cere *View*-ului să afișeze noua stare. Astfel se vede clar că `main()` nu lucrează direct nici cu starea internă a modelului, nici cu detaliile de afișare, ci pornește colaborarea dintre cele trei părți.

```
// Demo
public class Demo {
    public static void main(String[] args) {
        SwitchModel model = new SwitchModel();
        SwitchView view = new SwitchView();
        SwitchController controller = new SwitchController(model, view);
        controller.handle("on");
        controller.handle("toggle");
    }
}
```

Modelul se testează prin ***unit tests***, fără UI, se verifică direct metodele care schimbă starea și regulile. ***Controller*-ul** se testează prin **teste care simulează intrări**, se verifică ce metode din Model sunt invocate și ce View este ales. **View-ul** se testează prin **verificarea ieșirii pentru date date**, de exemplu randarea unui șablon cu un model simplu. Pentru început, chiar și un exemplu pe consolă ilustrează bine aceste granițe, un *Model* cu metode clare, un *View* care tipărește starea, un *Controller* care interpretează comenzi text, exact stilul minimal deja folosit.

Apar **probleme** când logica de domeniu este strecurată în *View*, de exemplu prin calcule în șabloane, deoarece testarea și refolosirea devin mai dificile. La fel, un *Controller* care decide reguli în locul modelului împrăștie responsabilitățile, iar modelul care știe de butoane sau rute creează cuplare inutilă. Codul care se schimbă la modificarea aspectului aparține *View*-ului. Codul care se schimbă la modificarea regulii de *business* aparține modelului. Codul care se schimbă la modificarea modului de interacțiune, de exemplu o rută nouă sau un buton nou, aparține *Controller*-ului.

Separarea clară oferă libertate de mișcare, UI se poate înlocui fără a atinge logica de domeniu, regulile se pot schimba fără a afecta prezentarea, iar coordonarea intrărilor rămâne lizibilă. În proiecte reale, această disciplină este cea care face diferența între un cod care poate fi extins în siguranță și unul în care fiecare schimbare rupe ceva în altă parte.

1.6.2. MVC pentru web, cerere-răspuns, șabloane, rute

În aplicațiile web, aceleași roluri rămân valabile, dar intrarea este o **cerere HTTP** (*request*), iar ieșirea este un **răspuns HTTP** (*response*). *Controller*-ul este o metodă mapată pe o **rută** (URL) care primește cererea HTTP, citește parametrii din cerere, validează, apelează Modelul și alege un **șablon** (*template*) pentru afișare. Modelul conține regulile de domeniu și nu știe nimic despre HTTP, iar *View*-ul (șablonul) se ocupă doar de prezentare, fără logică de domeniu [32].

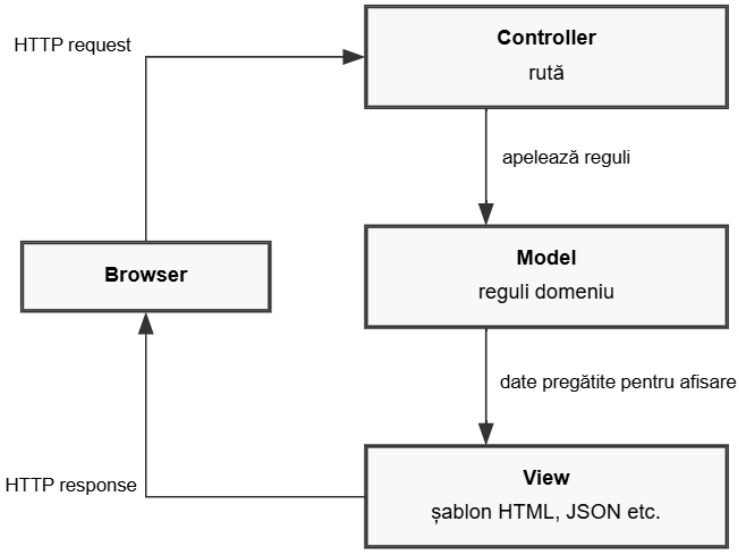


Figura 1.21. MVC pentru web (*request, controller, model, view* și *response*)

Figura 1.21 ne arată **cum se adaptează MVC la web**. Browserul trimite o cerere HTTP, ruta o duce către *Controller*, *Controller*-ul cheamă modelul sau un serviciu de domeniu, iar rezultatul este randat printr-un *View* sau transmis ca JSON. În aplicațiile moderne, *View*-ul poate fi pe server, de exemplu JSP, sau pe client, într-un *framework frontend*.

Pentru exemplificare, codurile unui **MVC web minimal** cu ***Java Servlets*** și **JSP** [33], [34]:

- **Model** (regulile domeniului):

```
// SwitchModel.java
public class SwitchModel {
    private boolean on;

    public boolean isOn() { return on; }
    public void turnOn()  { on = true; }
    public void turnOff() { on = false; }
    public void toggle()  { on = !on; }
}
```

- **Controller** (rutele, mecanismul cerere-răspuns):

```
// SwitchController.java
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import jakarta.servlet.annotation.*;

@WebServlet("/switch")                                  // ruta: /switch
public class SwitchController extends HttpServlet {
    private final SwitchModel model = new SwitchModel(); // un model simplu
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, java.io.IOException {

        // 1) citire parametrul "cmd" din cerere
        //   (rutare simpla in interiorul controlerului)
        String cmd = req.getParameter("cmd");

        // 2) validare si aplicare regula in Model
        if ("on".equals(cmd))      model.turnOn();
        else if ("off".equals(cmd)) model.turnOff();
        else if ("toggle".equals(cmd)) model.toggle();
        else if (cmd != null) req.setAttribute("msg", "comanda: " + cmd);

        // 3) pregatire date pentru View
        req.setAttribute("state", model.isOn() ? "ON" : "OFF");

        // 4) alegere si trimitere catre sablon (View)
        RequestDispatcher rd =
                req.getRequestDispatcher("/WEB-INF/views/switch.jsp");
        rd.forward(req, resp);
    }
}
```

- **View** (șablonul JSP, fără logică a domeniului):

```
<!-- /WEB-INF/views/switch.jsp -->
<!DOCTYPE html>
<html>
<head><meta charset="UTF-8"><title>Switch</title></head>
<body>
  <h1>Stare: ${state}</h1>
  <p style="color: #c00">${msg}</p>
  <ul>
    <li><a href="/app/switch?cmd=on">Porneste</a></li>
    <li><a href="/app/switch?cmd=off">Opreste</a></li>
    <li><a href="/app/switch?cmd=toggle">Comuta</a></li>
  </ul>
</body>
</html>
```

Ruta `/switch` duce mereu în `Controller`, acesta citește `cmd`, invocă **o singură** regulă în *Model* (fără HTML sau redări), apoi pasează două date simple către *View*, `state` și, opțional, `msg`. Șablonul afișează starea și oferă linkuri, fără să conțină reguli de domeniu. **Testarea** devine locală, Modelul se testează prin *unit tests*, *Controller*-ul se testează cu cereri simulate către `/switch?cmd=on`, iar șablonul se verifică prin randarea valorilor primite.

Observații practice:

- în exemplu, rutarea este făcută prin adnotarea `@WebServlet("/switch");` în *framework*-uri web, rutele se declară similar, iar *Controller*-ul rămâne translatorul dintre HTTP și *Model*;
- *Controller*-ul verifică parametrii și semnalează erori prin mesaje clare, nu amestecă regulile de domeniu în *View*;
- dacă se schimbă UI, se editează doar șablonul; dacă se schimbă regula, se atinge doar modelul; dacă apare o rută nouă, se adaugă un *Controller* sau o nouă ramură controlată.

1.6.3. Alternative la MVC

După ce MVC a fixat granițele, în aplicațiile cu interfețe bogate au apărut **variații** care organizează mai bine logica de prezentare. Scopul este același, modelul rămâne locul regulilor de domeniu, iar prezentarea și coordonarea fluxului sunt structurate astfel încât schimbările de UI să nu atingă regulile și invers [35].

MVP (*Model, View, Presenter*): *View* este cât mai simplu, afișează date și trimite evenimente de interacțiune. *Presenter* primește evenimentele de la *View*, cheamă modelul, transformă rezultatele în date pregătite pentru afișare și le împinge din nou spre *View*. *View* nu conține logică de prezentare semnificativă, doar invocă *Presenter* și desenează. Avantajul este testarea ușoară a Presenterului, deoarece are intrări și ieșiri clare.

MVVM (*Model, View, ViewModel*): *ViewModel* expune stare observabilă, de exemplu câmpuri reactive care se actualizează când se schimbă modelul sau când utilizatorul face o acțiune. *View* se leagă declarativ de această stare și se actualizează automat. Spre deosebire de MVP, în MVVM nu se împing explicit datele către *View*, ci *View* se abonează la starea *ViewModel*-ului. Câștigul este un flux de date previzibil și mai puțin cod de conectare.

***Frontend* modern, evenimente și stare:** În aplicațiile de client moderne, componentizarea UI și starea reactivă sunt norma, componentele declară ce afișează în funcție de o stare, evenimentele produc intenții, iar starea este actualizată prin reguli clare. Ideea relevantă pentru începători este aceeași ca în MVP și MVVM, logica de prezentare stă lângă *ViewModel* sau *Presenter*, logica de domeniu rămâne în *Model*, iar *View* doar reflectă starea.

BFF (*Backend for Frontend*): Când același *Model* este folosit de mai multe interfețe, de exemplu web, mobil, panou intern, este util un strat de server dedicat fiecărui *frontend*. BFF adaptează API-ul la nevoile acelui client, combină date din mai multe surse, ascunde pașii tehnici și simplifică *ViewModel*-ul sau *Presenter*-ul. Beneficiul este un *frontend* mai subțire și teste mai simple la nivelul UI [36].

Familia MVC arată **cum se organizează interiorul unei aplicații** pentru a **separa regulile, prezentarea** și **controlul fluxului**. Următorul pas este separarea aplicațiilor sau a capabilităților între ele. Când granițele nu mai sunt doar clase, pachete sau șabloane, ci procese care comunică prin rețea, discuția trece spre servicii, SOA și microservicii.

1.6.4. Arhitectura hexagonală, *Ports and Adapters*

O evoluție importantă a separării responsabilităților este arhitectura hexagonală, cunoscută și ca *Ports and Adapters*. Ideea principală este ca logica de domeniu să fie plasată în centru, iar interacțiunile cu exteriorul să fie conectate prin porturi și adaptoare. Exteriorul poate însemna interfață web, bază de date, sistem de fișiere, broker de mesaje, serviciu REST sau aplicație terță [37].

Un **port** este un **contract** prin care **domeniul comunică spre exterior sau este apelat din exterior**. Un **adaptor** este **implementarea concretă care leagă acel port de o tehnologie**. De exemplu, domeniul poate avea nevoie de un port `OrderRepository`, iar adaptorul concret poate salva comenzile într-o bază de date relațională, într-un fișier sau într-un serviciu la distanță. Domeniul nu ar trebui să depindă direct de SQL, HTTP sau de un anumit *framework*. Figura 1.22 sintetizează această separare dintre domeniu, porturi și adaptoare.

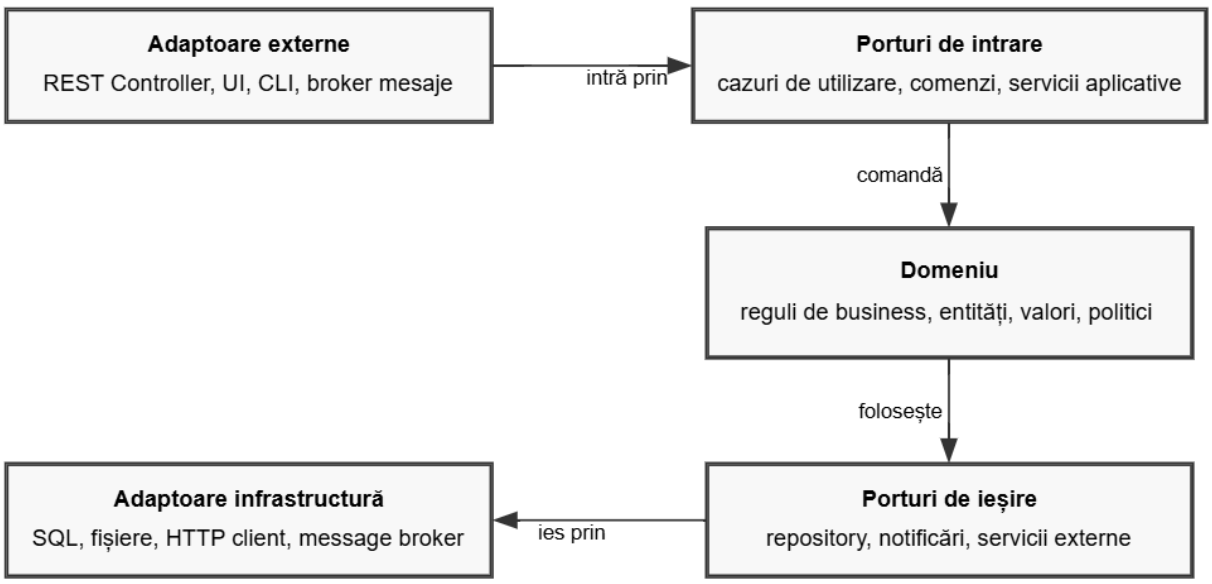


Figura 1.22. Arhitectura hexagonală (domeniu central, porturi și adaptoare)

Această arhitectură este o continuare naturală a ideilor discutate anterior: programare spre interfețe, injectare de dependențe, *Adapter* și separarea responsabilităților. Ea va fi utilă și în studiile de caz, unde aceeași logică poate fi folosită într-un monolit modular, expusă prin REST sau conectată la evenimente.

1.7. De la componente la servicii

După clase, obiecte, componente și *framework*-uri, următorul pas natural este **expunerea capabilităților prin servicii**, adică prin **contracte clare, independente de limbajul intern** sau **de procesul în care rulează** [38]. Ideea practică este simplă, în loc să legăm direct biblioteci sau să împachetăm totul într-o aplicație unică, definim ce oferă fiecare piesă printr-un contract stabil, cum se cheamă, ce date primește și ce întoarce, apoi lăsăm execuția și evoluția fiecărui serviciu să fie autonome. Asta deschide calea pentru echipe independente, versiuni diferite care conviețuiesc, scalare separată și integrare cu sisteme externe.

Trecerea de la componente la servicii nu înseamnă abandonarea ideilor anterioare, ci mutarea lor la o graniță mai largă. O componentă este, de obicei, folosită în interiorul aceleiași aplicații sau al aceluiași *runtime*. **Un serviciu expune o capabilitate printr-un contract accesibil prin rețea.** Clientul nu mai leagă o bibliotecă sau o clasă locală, ci trimite o cerere către un **punct final** (***endpoint***, adică **adresa prin care o operație sau o resursă este accesată** prin HTTP), primește un răspuns și trebuie să țină cont de latență, indisponibilitate, erori de rețea, versionare și securitate. Aceeași disciplină rămâne valabilă: contract public, implementare ascunsă, responsabilitate clară și evoluție controlată. Diferența este că granița devine mai costisitoare. Un **apel local** poate fi rapid și sigur din punctul de vedere al transportului. **Un apel de serviciu** poate eșua din motive care nu au legătură cu logica de *business*: conexiune întreruptă, depășirea timpului de așteptare, autentificare eșuată, format incompatibil sau serviciu indisponibil. De aceea, serviciile trebuie proiectate cu mai multă atenție la contracte, erori și observabilitate. Figura 1.23 compară apelul local către o componentă cu apelul prin rețea către un serviciu.

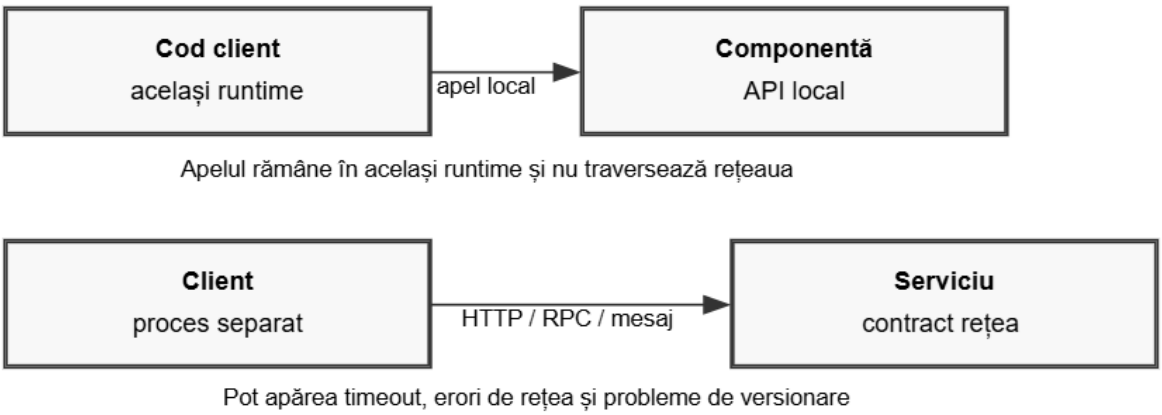


Figura 1.23. Componentă față de serviciu (apel local față de contract prin rețea)

Componenta locală este folosită printr-un apel în același proces sau *runtime*. Serviciul este folosit printr-un protocol, deci contractul trebuie să includă nu doar operațiile, ci și formatul mesajelor, codurile de eroare, regulile de securitate și comportamentul la eșec.

1.7.1. Arhitecturile orientate spre servicii (SOA)

Service-Oriented Architecture (SOA) este un **stil arhitectural** în care **capabilitățile** unui sistem **se expun ca servicii cu contracte bine definite, invocate prin mesaje**. Accentul este pe contract, nu pe implementare, serviciul este o cutie neagră care respectă un set de reguli la interfață, operații, mesaje, politici, erori, versiuni [38]. În practică, contractul poate fi descris explicit, de exemplu WSDL pentru SOAP sau specificații *OpenAPI* pentru REST [39], [40]. Ideea este aceeași, consumatorul serviciului depinde de contract, furnizorul serviciului poate evolua atâta timp cât nu rupe contractul.

Contractul unui serviciu spune ce operații există, ce formate de mesaje se acceptă, ce precondiții sunt necesare, ce răspunsuri se întorc, ce erori pot apărea și ce politici se aplică, de exemplu autentificare, limitare de rată (*rate limiting*), depășirea timpului de așteptare. Un contract clar permite testare la nivel de interfață, contracte verificate de consumatori și verificarea compatibilității la versiuni, de exemplu adăugarea unui câmp opțional în răspuns fără a rupe clienții.

Integrarea între servicii se poate face în două moduri complementare. **Orchestrarea** înseamnă un element central care conduce pașii ca un dirijor, un serviciu sau motor de procese care apelează pe rând celelalte servicii, decide ordinea, tratează erori, retransmisiile. Este ușor de urmărit și de controlat, dar introduce un punct central care trebuie operat cu grijă. **Coregrafia** înseamnă reguli locale și mesaje între participanți fără un dirijor central, fiecare serviciu știe când să emită și ce să facă atunci când primește un eveniment. Scalează mai bine și reduce dependența de un nod unic, dar cere disciplină în definirea contractelor de evenimente și în observabilitate pentru a reconstitui fluxul. Practic, procese critice cu pași clari se modelează prin orchestrare, fluxuri distribuite cu variații frecvente se modelează prin coregrafie.

Ca să fie găsite și folosite, serviciile se publică într-un **registru de servicii**, unde ele sunt descrise și unde se publică metadate, *endpoint*-uri, versiuni, politici. În SOA clasic exista **UDDI** (*Universal Description, Discovery and Integration*), un registru standard pentru publicarea și descoperirea serviciilor. În arhitecturile moderne rolul este îndeplinit de cataloage de API și registre de descoperire a serviciilor. Guvernanța la nivel de companie înseamnă reguli și procese pentru a menține portofoliul de servicii coerent, versionare, compatibilitate, securitate, convenții de nume, politici de retragere, monitorizare și acorduri de nivel de serviciu [38]. În această zonă intră și **ESB** (*Enterprise Service Bus*), componentă de integrare care intermediază mesaje, aplică transformări și politici. ESB poate ajuta la standardizare și vizibilitate, însă nu ar trebui să devină locul unde se pune logica de *business*, altfel apare un monolit distribuit greu de schimbat [41].

Figura 1.24 sintetizează **ideea clasică SOA**. Consumatorii nu ar trebui să depindă de detalii interne ale serviciilor, ci de contracte publicate. Registrul ajută la descoperirea sau documentarea serviciilor, iar infrastructura de integrare, de exemplu un ESB, poate aplica rutare, transformări, securitate și monitorizare. În practică, implementările variază, dar ideea rămâne aceeași: capabilitățile aplicației sunt expuse ca servicii guvernate prin contracte.

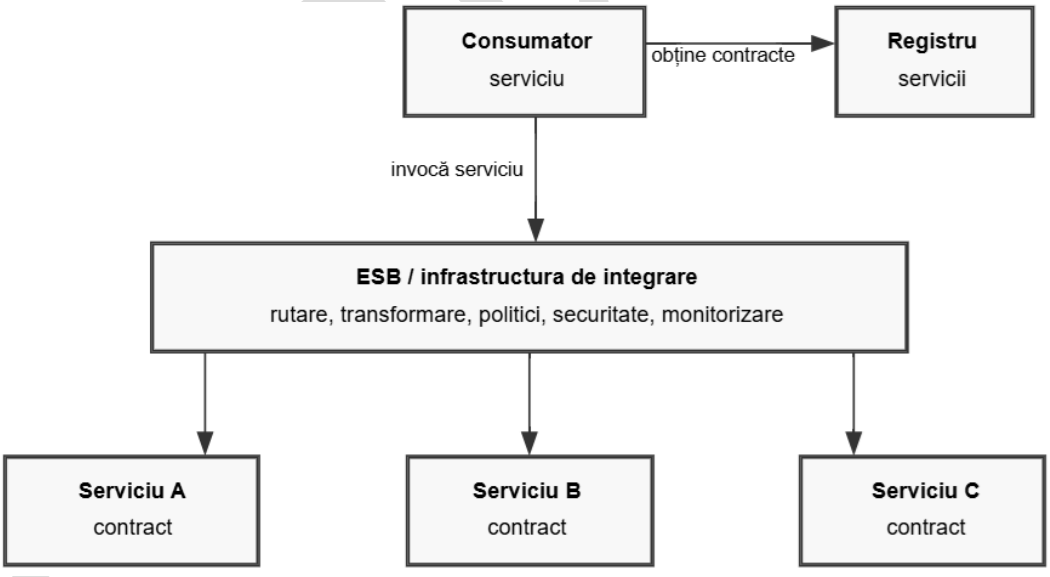


Figura 1.24. SOA (relația dintre consumatori, servicii, registru și infrastructura de integrare)

SOA a fost importantă deoarece a mutat atenția de la obiecte distribuite la capabilități de *business* expuse prin contracte. Un serviciu nu este doar o metodă apelată prin rețea, ci o funcție a organizației sau a sistemului, de exemplu verificare client, procesare plată, emitere factură sau rezervare stoc. Această perspectivă ajută la definirea unor granițe stabile.

Exemplu concret: un serviciu `Plăți` expune operațiile `initierePlata()` și `confirmaPlata()`, contractul precizează câmpurile, regulile de validare și codurile de eroare. Un proces `Comandă` orchestrează pașii, după ce primește comanda, verifică stocul, inițiază plata, rezervă transportul și confirmă clientul, toate prin contracte. În paralel, serviciul `Evenimente` plăți emite un mesaj `PlataConfirmata`, iar alte servicii se coregrafiază în jurul lui, de exemplu un serviciu de fidelizare care adaugă puncte fără a fi chemat explicit de orchestrator. Contractele rămân singurele dependențe, implementările pot evolua separat.

SOA nu înseamnă ESB peste tot. ESB este un instrument, nu locul logicii de *business*. Contractele care omit detalii critice, de exemplu timpi de răspuns sau tipuri de erori, surprind consumatorii, aceste aspecte trebuind declarate. Orchestrarea oferă control, dar poate aduce riscuri, coregrafia oferă elasticitate, dar cere observabilitate și contracte de evenimente bine definite. Acordurile de nivel de serviciu, adică ținte măsurabile pentru disponibilitate, latență și rate de eroare, trebuie urmărite explicit, altfel nu se poate garanta comportamentul promis către clienți.

1.7.2. Microservicii

Microserviciile sunt un stil arhitectural în care **aplicația este construită din servicii mici și autonome**, fiecare **cu un scop clar**, care **pot fi dezvoltate, livrate și operate independent** [22]. Fiecare serviciu are propriul cod, propriul ciclu de livrare și propriile măsurători de sănătate. Granițele acestor servicii nu se aleg întâmplător, ele urmează limitele naturale ale domeniului, astfel regulile care se schimbă împreună stau împreună.

Figura următoare sintetizează **o organizare frecventă a unei aplicații bazate pe microservicii**. Clientul nu accesează direct fiecare serviciu, ci intră printr-un *API Gateway*, care oferă un punct comun de intrare și poate prelua responsabilități precum rutarea, autentificarea sau aplicarea unor politici de acces [22]. În spatele acestuia, serviciile sunt separate și autonome, fiecare având propria responsabilitate și, de regulă, propria bază de date.

Figura 1.25 ilustrează această combinație dintre **intrare unificată, servicii autonome** și **integrare prin evenimente**.

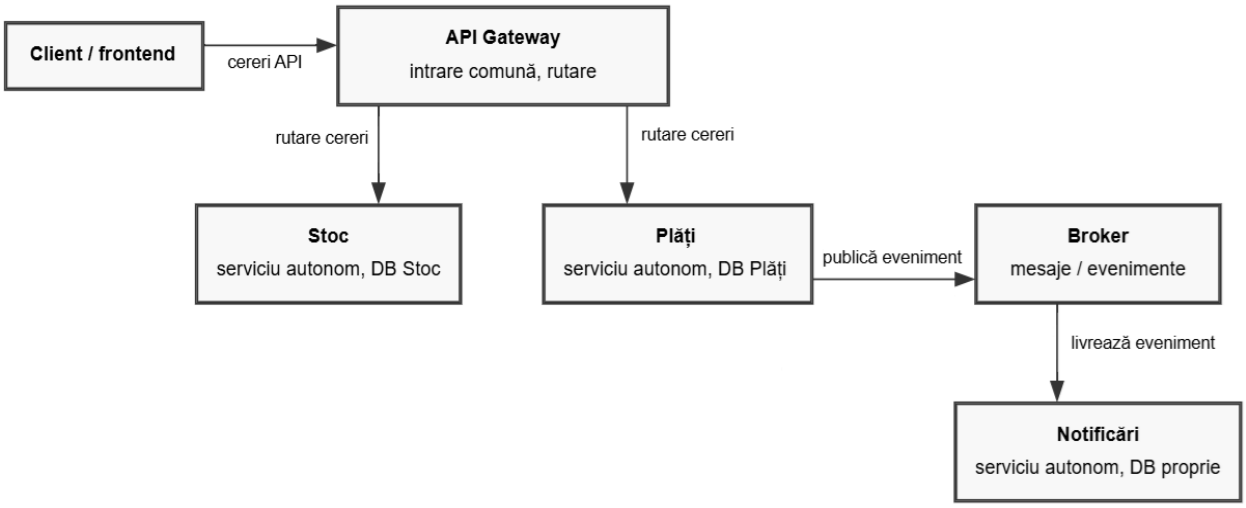


Figura 1.25. Microservicii (*API Gateway*, servicii autonome cu baze de date proprii, integrare prin evenimente)

În figură, serviciile ***Stoc*** și ***Plăți*** sunt apelate prin gateway, ceea ce ilustrează integrarea sincronă de tip cerere-răspuns. În schimb, legătura dintre ***Plăți***, ***Broker*** și ***Notificări*** sugerează o integrare asincronă, bazată pe publicarea și consumul de evenimente. Astfel, un serviciu poate continua fluxul de procesare fără să cunoască direct toate celelalte servicii care vor reacționa ulterior. În plus, fiecare serviciu își deține propriile date, ceea ce reduce cuplarea la nivel de persistență, dar cere contracte clare și o bună observabilitate la nivelul întregului sistem.

Diagrama nu trebuie citită ca o rețetă fixă, ci ca o schemă didactică. Nu orice sistem cu microservicii are neapărat un *API Gateway* sau un broker de mesaje în aceeași formă. Totuși, ea surprinde câteva idei definitorii: **separarea pe servicii mici**, **autonomie la nivel de date**, **combinația** dintre **apeluri sincrone** și **comunicare asincronă**, precum și **nevoia de observabilitate** pentru operarea întregului ansamblu.

Față de o aplicație stratificată clasică, diferența principală nu este doar împărțirea codului pe straturi, ci mutarea granițelor la nivel de procese, contracte și responsabilități operaționale. Într-un microserviciu, codul, datele, contractul public și monitorizarea formează o unitate care trebuie proiectată, livrată și operată coerent. Câștigul este autonomia, dar costul este complexitatea distribuită.

Microserviciile pot fi înțelese ca **o reinterpretare mai ușoară și mai operațională a unor idei SOA**. Ambele pun accent pe servicii și contracte, dar diferă prin accent. SOA clasică a folosit adesea infrastructuri centrale de integrare, contracte formale și guvernanță puternică. Microserviciile preferă servicii mai mici, autonome, livrate independent, cu infrastructură mai descentralizată și responsabilitate mai mare la nivelul echipei care deține serviciul. Această autonomie vine însă cu preț: monitorizare, versionare, compatibilitate, testare end-to-end și diagnostic distribuit mai dificile.

Pentru alegerea granițelor, merită amintit pe scurt *Domain-Driven Design* (DDD), abordarea formulată de Eric Evans, care recomandă modelarea software-ului pornind de la limbajul și regulile domeniului, nu de la tabele sau tehnologii. DDD delimitează zone coerente numite *bounded context*, în interiorul cărora termenii au același sens și regulile se schimbă împreună [42]. Într-un magazin online, ***Plăți***, ***Catalog*** și ***Comenzi*** pot fi astfel de contexte, fiecare cu propriul model pentru cuvinte aparent identice, de exemplu „preț" sau „stare". Ideal, microserviciile se aliniază acestor contexte sau unor părți bine delimitate din ele, iar traducerea dintre contexte se face prin contracte explicite. Astfel se evită modelul unic, greu de schimbat, iar echipele pot evolua independent fără să își impună reciproc detalii interne.

O consecință practică este că fiecare microserviciu ar trebui să dețină propriile date, baze de date separate sau cel puțin scheme separate, fără acces direct din alte servicii. Colaborarea se face prin API-uri și mesaje, nu prin tabele partajate. Beneficiul este independența reală: un serviciu poate schimba schema internă fără a rupe alte servicii. Costul este că actualizările care ating mai multe servicii nu se mai fac, de regulă, printr-o tranzacție globală, ci prin comenzi compensatorii, evenimente și consistență eventuală.

Comunicarea poate fi sincronă, de tip **cerere-răspuns** (*request-response*), utilă pentru interogări imediate, **sau asincronă, prin evenimente**, potrivită pentru propagarea schimbărilor fără cuplare directă. În practică, cele două stiluri se combină: unele citiri sau validări sunt sincrone, iar actualizările și integrările pot fi propagate prin evenimente. Important este ca semnificația contractelor să fie clară: ce se garantează, când se garantează, ce coduri de eroare apar și ce se întâmplă dacă un serviciu nu răspunde. În apelurile sincrone apar frecvent depășirea timpului de așteptare și reîncercări controlate, pentru ca o eroare locală să nu se propage necontrolat în întregul sistem [22].

Într-un **monolit**, diagnosticul se poate face de obicei urmărind jurnalul aplicației într-un singur loc. În microservicii, aceeași cerere poate trece prin mai multe procese, deci este nevoie ca informațiile de diagnostic să poată fi corelate. **Observabilitatea** este capacitatea de a înțelege ce se întâmplă într-un sistem aflat în execuție. Ea se bazează pe trei tipuri de informații: înregistrări de jurnal, metrici și urmărirea parcursului cererii prin mai multe servicii. Înregistrările de jurnal arată ce s-a întâmplat, metricile arată cât de des, cât de repede sau cu câte erori se întâmplă, iar urmărirea parcursului arată prin ce servicii a trecut cererea. Pentru aceasta, trebuie păstrat același identificator de corelație al apelurilor, astfel încât parcursul unei cereri să poată fi reconstruit [43].

Microserviciile se schimbă independent, deci contractele trebuie să rămână compatibile în timp. Variantele noi ar trebui să adauge câmpuri opționale, nu să schimbe semnificații existente, iar contractele importante pot fi verificate prin teste. Când un contract vechi devine depășit, se marchează ca depreciat și se menține o perioadă, pentru a permite migrarea consumatorilor.

Un criteriu practic este următorul: nu orice clasă, modul sau componentă trebuie transformată într-un microserviciu. **O graniță de microserviciu merită introdusă atunci când există un motiv real de autonomie**: ritm de livrare diferit, scalare diferită, echipă responsabilă separat, model de date distinct sau cerințe operaționale diferite. Dacă aceste motive nu există, un monolit modular poate fi mai simplu, mai rapid și mai ușor de testat. De aceea, microserviciile nu trebuie văzute ca final obligatoriu al evoluției, ci ca opțiune arhitecturală utilă în anumite condiții.

O direcție importantă în cloud este ***serverless***. În acest stil, unitatea de livrare poate deveni o funcție sau un *handler* declanșat de un eveniment, iar infrastructura de rulare este gestionată de platforma cloud. Dezvoltatorul scrie codul care reacționează la o cerere HTTP, la un mesaj, la încărcarea unui fișier sau la o modificare într-o bază de date, iar platforma se ocupă de pornirea instanțelor, scalare, izolare și execuție [44]. *Serverless* nu elimină arhitectura, ci mută unele responsabilități către platformă. Aplicația trebuie în continuare proiectată cu granițe clare, contracte explicite, tratarea erorilor, observabilitate și securitate. Multe probleme devin chiar mai importante, deoarece execuția este distribuită, efemeră și declanșată de evenimente. Câștigul este elasticitatea și reducerea administrării infrastructurii. Costul este dependența mai mare de platformă, dificultatea testării locale și nevoia de observabilitate foarte bună.

1.7.3. Exemplu de microserviciu REST cu JSON

Pentru a face vizibilă comunicarea dintre un client și un microserviciu, folosim un exemplu minim: un serviciu„***Plăți***" cu o operație de inițiere și un client foarte simplu care îl apelează prin HTTP cu JSON. Scopul nu este să acoperim toate cazurile reale, ci să vedem cum arată un schimb simplu de mesaje JSON, cum se semnalează erorile și cum pot fi adăugate două detalii utile pentru robustețe: o cheie de idempotentă pentru reîncercări sigure și un identificator de corelație pentru diagnostic [45], [46], [47]. După acest exemplu, cititorul poate recunoaște în proiecte reale aceleași elemente și le poate extinde controlat.

Exemplul următor nu urmărește să acopere toate detaliile unei aplicații REST reale. Scopul lui este să arate forma minimă a unui apel HTTP cu JSON: o adresă, o metodă, date trimise în cerere și un răspuns interpretabil. Detaliile despre REST, versionare și compatibilitate sunt discutate separat în secțiunea 2.7.2.

Înainte de a analiza codul, merită privit mai întâi **fluxul conceptual al exemplului**. Figura 1.26 descrie pașii principali ai unui microserviciu REST: primirea unei cereri HTTP cu JSON, validarea datelor de intrare, verificarea idempotentei, reutilizarea unui rezultat existent sau crearea unuia nou și întoarcerea unui răspuns coerent. Ea evidențiază trei situații importante: cerere invalidă, cerere repetată și cerere nouă.

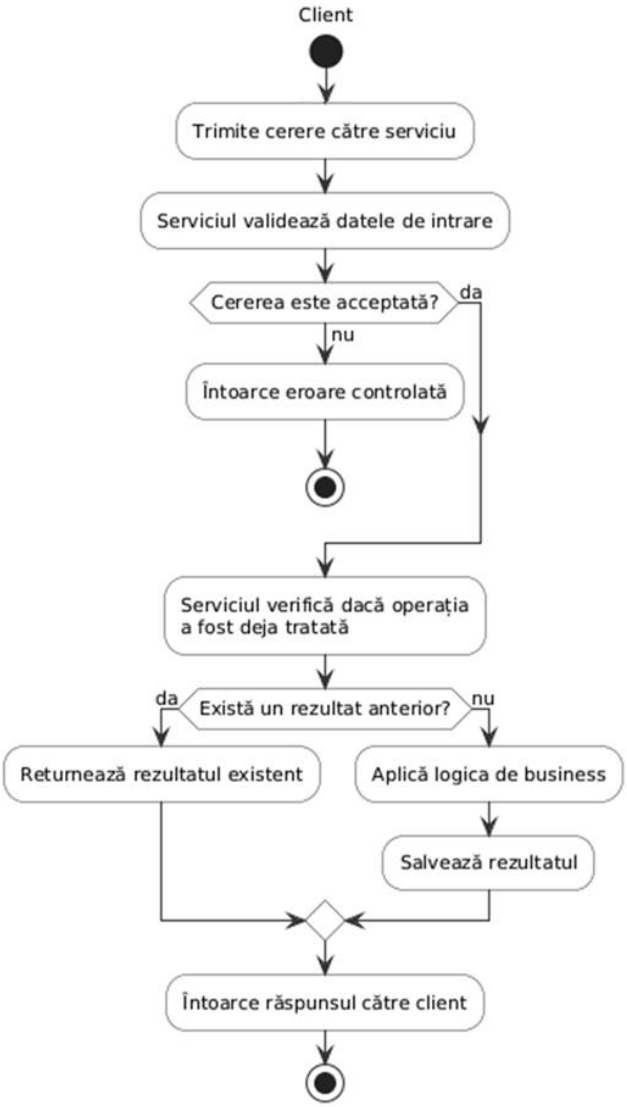


Figura 1.26. Flux conceptual pentru inițierea unei plăți printr-un microserviciu REST, de la cerere la răspuns

Codul **Java** al unui **mini-serviciu REST** `Payments`, folosind *Spring Boot* [48]:

```
// PaymentController.java
@RestController
@RequestMapping("/v1/payments")
public class PaymentController {
    // in-memory store doar pentru demo: idempotencyKey -> Payment
    private final ConcurrentHashMap<String, Payment> store =
                                        new ConcurrentHashMap<>();
    private final AtomicLong seq = new AtomicLong(1);

    @PostMapping
    public ResponseEntity<PaymentResponse> initiatePayment(
         @RequestBody InitiatePaymentRequest req,
      @RequestHeader(value = "Idempotency-Key", required = false) String idemKey,
      @RequestHeader(value = "X-Correlation-Id", required = false) String corrId)
      {
        // guard: correlation id (genereaza daca lipseste)
        if (corrId == null || corrId.isBlank()) {
```

```
            corrId = UUID.randomUUID().toString();
        }
        // validare input minimal
        if (req == null || req.amount() == null || req.amount().value() <= 0) {
            // model de eroare simplu pentru demo
            return ResponseEntity.unprocessableEntity()
                    .header("X-Correlation-Id", corrId)
                    .body(new PaymentResponse(null, "invalid_amount"));
        }
        // idempotency: daca avem cheie si exista deja, returnam acelasi rezultat
        if (idemKey != null && store.containsKey(idemKey)) {
            Payment existing = store.get(idemKey);
            return ResponseEntity.status(HttpStatus.CREATED)
                    .header("Location", "/v1/payments/" + existing.id())
                    .header("X-Correlation-Id", corrId)
                    .body(new PaymentResponse(existing.id(), existing.status()));
        }
        // creare plata noua
        String id = "pay_" + seq.getAndIncrement();
        Payment p = new Payment(id, "pending");
        if (idemKey != null) {
            store.put(idemKey, p);
        }
        return ResponseEntity.status(HttpStatus.CREATED)
                .header("Location", "/v1/payments/" + id)
                .header("X-Correlation-Id", corrId)
                .body(new PaymentResponse(id, "pending"));
    }
    // DTO-uri minimaliste (Java 17+ records)
    public record InitiatePaymentRequest(String orderId, Money amount, String
method) {}
    public record Money(String currency, double value) {}
    public record PaymentResponse(String paymentId, String status) {}
    public record Payment(String id, String status) {}
}
```

Controllerul expune `POST /v1/payments` și aplică la vedere câteva reguli esențiale pentru microservicii: versionarea explicită în cale, validarea intrării la interfață, idempotenta prin antetul „`Idempotency-Key`", includerea „`X-Correlation-Id`" în răspuns și indicarea resursei create prin antetul „`Location`". Validarea respinge rapid valori invalide, ceea ce mută erorile aproape de sursă și simplifică testarea. Idempotenta permite clientului să repete aceeași cerere fără efecte secundare nedorite, important când apar depășirea timpului de așteptare sau reîncercări automate [50]. Identificatorul de corelație face posibilă urmărirea parcursului cap-la-cap al unei cereri, deoarece fiecare componentă poate scrie același identificator în jurnalul ei de diagnostic. Modelul de răspuns este intenționat minimalist, câmpurile sunt puține și stabile, astfel contractul poate evolua ulterior adăugând opționale, fără a rupe clienții existenți.

Codul **JavaScript** al unui **apel simplu către serviciu** (*React*) [49]:

```
// PaymentButton.jsx
import { useState } from "react";

export default function PaymentButton() {
  const [result, setResult] = useState(null);
  const [error, setError] = useState(null);

  const startPayment = async () => {
    const correlationId = crypto.randomUUID(); // id pentru urmarirea cererii
    const idemKey = crypto.randomUUID();       // idempotency key pentru retry
```

```
    try {
      const res = await fetch("/v1/payments", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
          "Accept": "application/json",
          "X-Correlation-Id": correlationId,
          "Idempotency-Key": idemKey
        },
        body: JSON.stringify({
          orderId: "ORD-2026-000123",
          amount: { currency: "EUR", value: 129.90 },
          method: "card"
        })
      });
      const data = await res.json();
      if (!res.ok && res.status !== 201) {
        setError({ status: res.status, data });
      } else {
        setResult({ status: res.status, data, location:
                              res.headers.get("Location") });
      }
    } catch (e) {
      setError({ status: "NETWORK", data: String(e) });
    }
  };
  return (
    <div>
      <button onClick={startPayment}>Inițiază plata</button>
      {result && (
        <pre>{JSON.stringify(result, null, 2)}</pre>
      )}
      {error && (
        <pre>{JSON.stringify(error, null, 2)}</pre>
      )}
    </div>
  );
}
```

Componenta React joacă rolul unui client simplu care ilustrează consumul unui contract REST: construiește o cerere JSON, setează din start „`X-Correlation-Id`” pentru trasabilitate, folosește o „`Idempotency-Key`” nouă pentru fiecare încercare și tratează separat răspunsul de succes și răspunsul de eroare.

Rezultatul arată ce este relevant pentru client, statusul HTTP, corpul JSON și antetul „`Location`” pentru pașii următori. Fragmentul nu introduce detalii de infrastructură sau librării suplimentare, ci rămâne la esențialul unui consumator corect: date curate la intrare, antete bine alese, verificări explicite la ieșire. Într-un proiect real, aceleași idei se generalizează în clienți reutilizabili, interceptori de rețea, politici de retransmisie și instrumente de monitorizare și diagnostic, însă structura de bază rămâne identică.

Figura 1.27 mută accentul de pe deciziile interne ale microserviciului pe **schimbul de mesaje dintre participanți**: clientul, *endpoint*-ul REST, componenta de validare, mecanismul de idempotentă și componenta care construiește răspunsul.

Astfel poate fi urmărită mai clar **ordinea comunicării**: clientul trimite cererea HTTP cu JSON și antete relevante, serviciul validează datele, verifică dacă cererea a mai fost procesată, creează sau reutilizează rezultatul și întoarce un răspuns HTTP coerent.

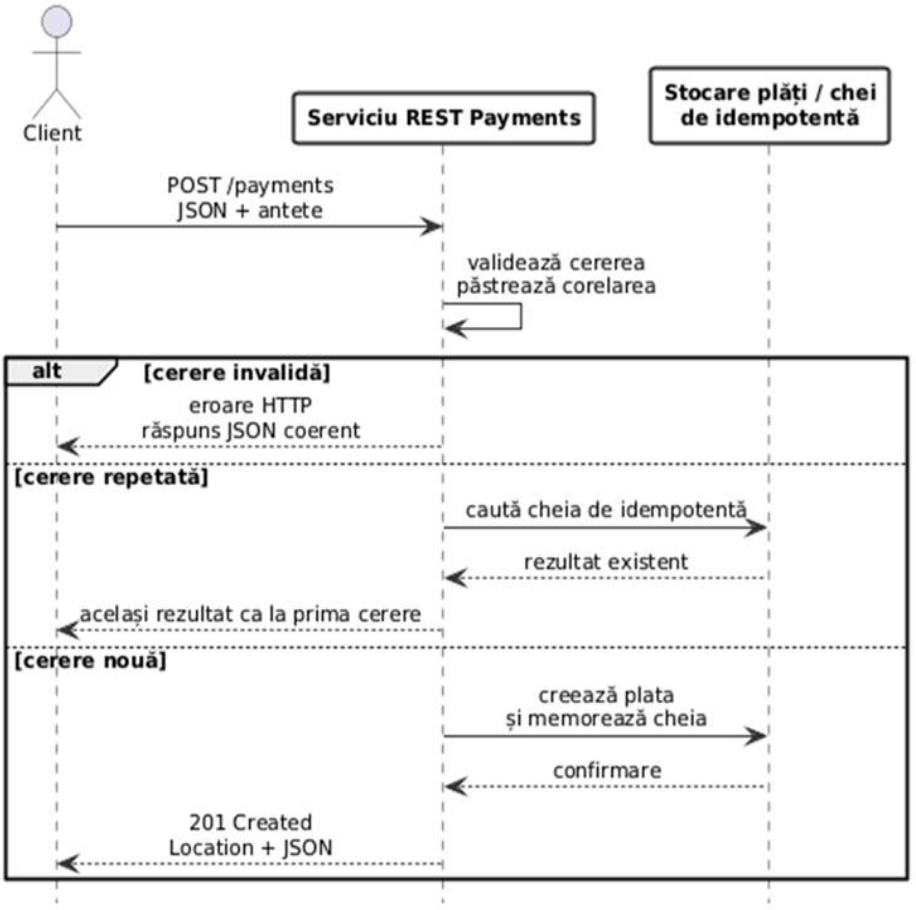


Figura 1.27. Interacțiunile pentru inițierea unei plăți printr-un microserviciu REST, de la cerere la răspuns

Din exemplu pot fi observate patru lucruri. În primul rând, **contractul** nu mai este o semnătură de metodă Java, ci combinația dintre URL, metodă HTTP, formatul cererii, formatul răspunsului și codurile de stare. În al doilea rând, apelantul nu mai primește o excepție locală obișnuită, ci trebuie să trateze **erori de transport**, **depășirea timpului de așteptare** și **răspunsuri HTTP diferite** [46]. În al treilea rând, datele trebuie **serializate**, de exemplu în JSON, ceea ce introduce problema compatibilității dintre versiuni [47]. În al patrulea rând, **monitorizarea și diagnosticul** devin esențiale: pentru a depana un apel distribuit, avem nevoie ca același identificator de cerere să apară în jurnalele componentelor implicate și avem nevoie de măsurători ale întârzierilor [43].

1.8. Concluzii și perspective

Capitolul a urmărit o **evoluție a gândirii software**, de la cod local și algoritmi simpli la servicii care comunică prin rețea. **Programarea structurată** a introdus disciplina fluxului clar, bazat pe secvență, decizie și iterație, apoi modularizarea a arătat că funcțiile și structurile de date pot fi organizate prin contracte și interfețe stabile. Chiar și în limbajul C, separarea dintre antet și fișierul sursă anticipează o idee care revine permanent în arhitectură: consumatorul trebuie să depindă de ceea ce este promis public, nu de detaliile interne.

Orientarea spre obiecte a dus această idee mai departe, reunind datele și operațiile în clase și obiecte. Încapsularea, polimorfismul, programarea spre interfețe, compunerea și injectarea de dependență au arătat cum pot fi construite părți mai flexibile și mai testabile. Principiile OO nu sunt reguli teoretice izolate, ci răspunsuri la probleme reale: clase prea mari, dependențe ascunse, stare greu de controlat, testare fragilă și modificări care se propagă neintenționat. Influența programării funcționale completează această perspectivă prin imutabilitate, funcții pure și reducerea efectelor laterale, mai ales în contexte concurente sau asincrone.

***Pattern*-urile de proiectare** au adăugat un vocabular comun pentru soluții recurente. *Adapter*, *Proxy*, *Observer*, *Command* sau *Singleton* nu trebuie memorate ca șabloane rigide, ci înțelese ca forme de colaborare între obiecte. Unele conectează interfețe incompatibile, altele controlează accesul, notifică schimbări, încapsulează acțiuni sau gestionează cicluri de viață. Alte familii de *pattern*-uri au dus aceeași idee spre *middleware*, aplicații *enterprise*, integrare și cloud. Valoarea lor practică apare atunci când ajută la clarificarea responsabilităților, nu atunci când sunt aplicate mecanic.

Orientarea spre componente a mutat atenția de la clase individuale la unități livrabile, versionabile și administrate de un *runtime*. *JavaBeans* și EJB au fost exemple istorice importante, deoarece au introdus explicit proprietăți, evenimente, introspecție, cicluri de viață, containere, tranzacții și politici declarative. *Framework*-urile și inversarea controlului (IoC) au simplificat apoi folosirea acestor idei: aplicația nu mai controlează tot fluxul, ci completează punctele de extensie oferite de infrastructură. Injectarea de dependențe (DI) face dependențele vizibile și permite înlocuirea implementărilor, ceea ce ajută direct testarea și evoluția.

MVC și variantele sale au arătat cum se separă modelul, prezentarea și controlul interacțiunilor. De aici, trecerea spre servicii devine naturală. O componentă locală este folosită în același proces sau *runtime*, în timp ce un serviciu expune o capabilitate printr-un contract de rețea. SOA a pus accent pe servicii guvernate, registre, politici și infrastructură de integrare. Microserviciile au dus mai departe ideea autonomiei, prin servicii mai mici, livrate independent, cu date proprii și observabilitate. *Serverless* continuă aceeași linie, dar mută și mai multă responsabilitate operațională către platforma cloud.

Această trecere în revistă nu epuizează toate paradigmele și stilurile arhitecturale. O lucrare mai amplă ar putea trata separat programarea logică, programarea reactivă, *Domain-Driven Design*, arhitectura hexagonală, CQRS, *DevOps*, arhitecturile *cloud-native* și modelele de securitate precum *zero trust*. Unele dintre acestea vor fi atinse indirect în capitolele următoare, mai ales prin integrare, mesagerie, evenimente, containerizare și cloud. Ele nu au fost dezvoltate aici deoarece scopul capitolului a fost să păstreze un fir didactic principal, de la modularizarea codului la servicii.

Tendințele recente, *low-code / no-code*, *AI-assisted development*, *agentic software engineering* și platformele *AI-native*, sunt importante, dar trebuie înțelese cu prudență. *Low-code* și *no-code* democratizează dezvoltarea și accelerează aplicațiile interne, însă sunt mai degrabă abordări de platformă și productivitate decât paradigme fundamentale de proiectare software [51]. AI-ul generativ și agenții software schimbă rapid modul în care se scrie, testează și modernizează codul, dar nu elimină nevoia de arhitectură [52], [53]. Dimpotrivă, codul generat automat are nevoie de contracte clare, teste, observabilitate, securitate și responsabilități bine separate. În acest sens, noile tendințe nu anulează evoluția prezentată în capitol, ci o fac și mai necesară: cu cât instrumentele produc mai repede software, cu atât devine mai important ca oamenii să înțeleagă principiile care îl fac corect, modificabil și sustenabil.

Capitolul 2. Arhitecturi de integrare pentru aplicații software

După ce am abordat, în capitolul 1, modul în care scriem și organizăm codul, de la funcții la obiecte, componente, framework-uri și servicii, **acest capitol tratează modul în care punem părțile în legătură**, pe aceeași mașină sau prin rețea. Pornind de la execuția locală, procese, fire și memorie, vom trece la comunicarea între procese pe aceeași mașină, integrarea prin baze de date, apelurile sincrone RPC și RMI, *middleware*-ul clasic, CORBA și ideea de proxy, mesageria asincronă, integrarea web prin HTTP, SOAP, REST și contracte JSON, apoi arhitecturile orientate spre evenimente, SOA, guvernanță, virtualizare, containerizare și cloud ca suport operațional pentru serviciile moderne. Ordinea urmărește, în linii mari, evoluția istorică a acestor mecanisme, dar toate rămân utile astăzi, fiecare fiind o cărămidă potrivită pentru anumite cerințe de integrare, scalare, compatibilitate și operare.

2.1. Execuție locală: procese, fire, memorie

Pe aceeași mașină de calcul, sistemul de operare rulează aplicațiile în procese. Un **proces** are **spațiul lui de adrese, resurse și descriptori de fișiere**, adică un „gard" natural care **izolează** greșelile. Dacă un proces se oprește anormal din cauza unei erori, el nu corupe memoria altuia. În interiorul unui proces pot exista unul sau mai multe fire de execuție. Firele împart aceeași memorie a procesului și pot colabora rapid, însă tocmai această partajare cere disciplină pentru a evita *race conditions* (condiții de cursă, două sau mai multe fire accesează aceeași resursă în același timp, iar rezultatul depinde de ordinea execuției) [54]. Figura 2.1 sintetizează diferența dintre izolarea proceselor și memoria comună a firelor din același proces.

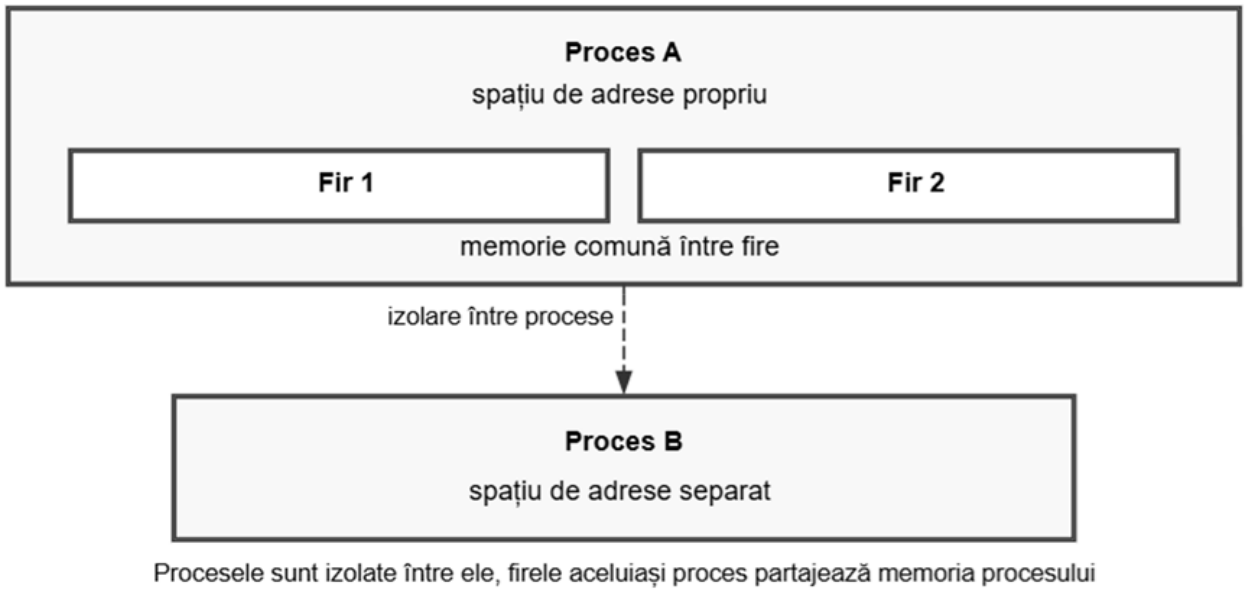


Figura 2.1. Proces și fire de execuție (izolare între procese, memorie comună între fire)

Pentru a face diferența mai concretă, putem porni de la un exemplu Unix foarte simplu. Apelul `fork()` creează un **proces copil** pornind de la procesul curent [55]. La prima vedere, copilul pare să continue execuția cu aceleași variabile, dar modificările făcute în copil nu schimbă variabilele din părinte, deoarece procesele au spații de adrese separate.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    int value = 10;
    pid_t pid = fork();
    if (pid == 0) {
        value = 20;
        printf("Child process, value = %d\n", value);
        return 0;
    }
    wait(NULL);
    printf("Parent process, value = %d\n", value);
    return 0;
}
```

În acest exemplu, procesul copil afișează valoarea 20, iar procesul părinte afișează valoarea 10. Variabila are același nume și aceeași valoare inițială în ambele procese, dar după `fork()` cele două execuții sunt separate. Modificarea din copil nu afectează memoria părintelui. Aceasta este ideea de bază a **izolării** dintre procese.

Procesele ilustrează izolarea, pe când **firele de execuție ilustrează colaborarea în interiorul aceluiași proces**. În **Java**, o confuzie frecventă este diferența dintre apelul direct al metodei `run()` și pornirea efectivă a unui fir prin `start()` [56].

```
public class ThreadDemo {
    public static void main(String[] args) {
        Thread t = new Thread(() -> {
            System.out.println("Worker: "
                    + Thread.currentThread().getName());
        });
        t.run();
        t.start();
        System.out.println("Main: " + Thread.currentThread().getName());
    }
}
```

Apelul **`run()`** nu pornește un fir nou. El execută metoda în firul curent, exact ca un apel obișnuit de metodă. Apelul **`start()`** cere *runtime*-ului Java să pornească un fir nou, iar codul asociat firului este executat separat. De aceea, pentru concurență reală se folosește `start()`, nu `run()` apelat direct.

Cele două exemple arată diferența de bază dintre izolare și partajare. Procesul oferă izolare mai bună, dar comunicarea cu alte procese cere mecanisme explicite. Firul de execuție colaborează mai ușor cu alte fire din același proces, dar partajarea memoriei cere sincronizare. Din această tensiune apar mecanismele discutate în continuare: memorie partajată, conducte (*pipes*), socketuri locale, cozi de mesaje și alte forme de IPC.

Modelul de memorie al unui proces se poate privi simplu ca patru zone, codul programului, datele statice, *heap*-ul și *stack*-ul. **Fiecare fir** are propriul *stack* pentru variabile locale și apeluri de funcții, iar toate firele văd același *heap*, zona unde sunt alocate dinamic obiectele și structurile. Consecința practică este clară, datele doar pentru citire pot fi partajate fără griji, la orice scriere pe date comune este nevoie de coordonare.

Coordonarea firelor se face cu primitive de sincronizare, un ***mutex*** serializează intrarea într-o secțiune critică, un **semafor** limitează câte fire pot intra, o **variabilă de condiție** permite așteptarea unui eveniment clar. Unde se poate, se evită partajarea. se folosesc obiecte imutabile, structuri sigure pentru acces concurent (*thread-safe*, folosite de mai multe fire fără coruperea stării interne), oferite de biblioteci standard sau modele avansate de tip *actor model* [54], [57].

Alegerea între procese și fire ține de **izolare**, **performanță** și **simplitate**. Procesele oferă granițe puternice și diagnostic mai clar pe instanță, prin jurnal, consum de resurse și stare proprie, dar comunicarea dintre procese este mai scumpă. Firele pornesc mai repede și pot schimba date în memorie cu latență mică, însă aduc responsabilitatea sincronizării și riscul de blocaje. O regulă sănătoasă este să se pornească de la un proces cu un număr controlat de fire gestionate de un grup de fire (*thread pool*), detaliile de sincronizare rămân încapsulate în spatele unor interfețe, restul codului folosește operații clare, nu blocări.

Performanța locală depinde mult de **memorie**. Pentru sarcini care solicită intens procesorul, un grup de fire (*thread pool*) evită costul de creare și distrugere repetată a firelor. Metrici precum dimensiunea cozilor, timpul mediu de așteptare pe blocări (*locks*) și rata de comutare de context (*context switch*) pot indica blocaje sau supraîncărcări. Testele de integrare caută explicit condiții de cursă (*race conditions*) și blocaje.

În ansamblu, se urmărește minimizarea partajării, validări timpurii la interfețe, încapsularea sincronizării într-un loc bine definit și măsurare înainte de optimizare. Când izolarea devine o cerință puternică sau comunicarea internă devine greu de controlat în memorie, se trece deliberat la comunicare între procese.

2.2. Comunicarea între procese pe aceeași mașină de calcul (IPC)

După ce au fost clarificate procesele și firele, următorul pas este comunicarea între procese. Comunicarea între procese, în engleză ***inter-process communication*** (IPC), înseamnă schimb de date și coordonare între programe izolate de „gardurile" sistemului de operare. Este necesară atunci când părți ale aplicației rulează în procese diferite, fie pentru izolare și stabilitate, fie pentru a folosi limbaje sau *runtime*-uri diferite, fie pentru a separa responsabilități operaționale.

Spre deosebire de fire, care partajează același spațiu de memorie, **procesele comunică prin mecanisme oferite de sistemul de operare**, memorie partajată controlată, canale de tip *pipe* (conductă) sau FIFO, socketuri locale Unix, cozi de mesaje, mapare de fișiere și tehnici cu copiere minimă sau zero între spațiile de adresă [54], [58].

Alegerea unui mecanism IPC se face în funcție de câteva criterii simple, **latență** și **debit**, **dimensiunea tipică a mesajelor**, **necesarul de ordine a livrării**, **modelul de programare** dorit, **flux de octeți sau mesaje** discrete, **ușurința de diagnostic** și **constrângerile de securitate**.

Memoria partajată oferă debit ridicat, dar cere sincronizare atentă și protocol clar pe marginea datelor comune. ***Pipes*** (conductele) și **FIFO** (cozile) simplifică fluxurile liniare, scriere la un capăt și citire la celălalt, cu blocare naturală când nu există consumator [59]. **Socketurile locale Unix** oferă o interfață familiară cu socketuri, dar rămân pe aceeași mașină, utilă când se dorește un protocol de mesaje clar și compoziție cu instrumente existente [60].

Cozile de mesaje evită blocarea simetrică a părților și pot adăuga persistență sau priorități [61]. **Maparea de fișiere în memorie**, *mmap*, facilitează împărțirea unor seturi mari de date între procese [58], iar **tehnicile *zero-copy*** reduc trecerile prin *bufferele kernel*-ului [62], [63].

Secțiunile următoare prezintă pe rând mecanismele principale, cu criterii de alegere și modele locale uzuale, producător–consumator și bucle de evenimente de tip *reactor* și *proactor*.

2.2.1. Memorie partajată, conducte și socketuri locale Unix

Acest subcapitol prezintă mecanismele de bază prin care datele circulă între procese pe aceeași mașină: memorie partajată pentru viteza maximă, conducte/cozi FIFO pentru fluxuri simple de tip „scrie–citește”, și *Unix Domain Sockets* pentru dialog bidirecțional cu API de tip socket.

Memoria partajată este o zonă de memorie pe care două sau mai multe procese o văd simultan în propriul spațiu de adrese. Avantajul este viteza, datele nu se mai copiază dintr-un proces în altul, ambele procese citesc și scriu direct în aceeași zonă. Costul este disciplina, fiindcă nu există „ordine” sau „blocare” implicită, trebuie definite reguli de acces și sincronizare, de exemplu un *mutex* și o variabilă de condiție plasate într-o zonă partajată, plus un protocol simplu, cine scrie, cine citește, cum se marchează „date disponibile”, când se semnalează „*buffer* gol” sau „*buffer* plin”. Memoria partajată este potrivită pentru volume mari de date și pentru schimburi cu latență mică, de exemplu un proces colectează mostre și altul le procesează în loturi, însă fără sincronizare corectă apar condiții de cursă (*race conditions*) și coruperea datelor.

Pentru a vedea concret ideea, putem folosi **un exemplu Unix minimal cu memorie partajată POSIX**. Un obiect de memorie partajată este creat prin `shm_open`, dimensionat cu `ftruncate`, apoi mapat în spațiul de adrese al procesului prin `mmap` [58]. După `fork()`, atât procesul părinte, cât și procesul copil văd aceeași zonă mapată [55].

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <unistd.h>

#define SHM_NAME "/demo_shm"
#define SHM_SIZE 128

int main(void) {
    int fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0600);
    ftruncate(fd, SHM_SIZE);

    char* shared = mmap(NULL, SHM_SIZE,
        PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    pid_t pid = fork();

    if (pid == 0) {
        strcpy(shared, "mesaj scris de procesul copil");
        return 0;
    }
    wait(NULL);
    printf("Parintele citeste: %s\n", shared);
```

```
    munmap(shared, SHM_SIZE);
    close(fd);
    shm_unlink(SHM_NAME);

    return 0;
}
```

În acest exemplu, **procesul copil scrie un text în zona partajată, iar procesul părinte îl citește după `wait(NULL)`.** Spre deosebire de exemplul anterior cu `fork()`, unde modificarea unei variabile locale în copil nu afecta părintele, aici cele două procese comunică printr-o zonă mapată explicit ca partajată. Apelul `wait(NULL)` este folosit doar pentru a păstra exemplul simplu: părintele așteaptă copilul înainte să citească. Într-o aplicație reală, această ordine ar trebui controlată prin mecanisme de sincronizare dedicate, nu prin presupuneri despre momentul execuției.

Exemplul arată și **limita principală** a memoriei partajate. Transferul este rapid, dar memoria partajată nu definește singură un protocol. Trebuie stabilit cine scrie, cine citește, când datele sunt valide, cum se marchează finalul unui mesaj și ce se întâmplă dacă două procese scriu simultan. De aceea, memoria partajată este potrivită pentru performanță, dar cere disciplină în proiectarea contractului dintre procese.

Exemplul anterior a folosit **maparea de fișiere în memorie** (***mmap***) pentru a mapa un obiect de memorie partajată. O variantă apropiată este maparea unui fișier obișnuit în memorie, tot prin ***mmap***. În acest caz, procesele împart o vedere comună asupra unui fișier. Toate procesele mapează același fișier, iar nucleul le oferă o vedere comună, scrierile apar în memorie și pot fi „împinse" pe disc. *mmap* este util când datele trebuie și persistate sau când dimensiunile sunt mari și nu se dorește citirea integrală în *buffere* [58]. Sistemul de operare încarcă datele la nevoie. Ca și la memoria partajată, sincronizarea accesului rămâne în sarcina aplicației, iar protocolul de acces trebuie documentat clar pentru a evita inconsistențe.

Un ***pipe*** (conductă) este un canal unidirecțional între două procese, la un capăt se scrie, la celălalt se citește. Este un flux de octeți cu ordine garantată, scrierile apar la citire în aceeași ordine, iar blocarea este naturală [59]. Dacă nu există date, citirea așteaptă. Dacă *bufferul* este plin, scrierea așteaptă. Conductele sunt foarte simple și potrivite pentru conectarea etapelor unei „linii de producție", de exemplu un proces produce linii de text, altul le filtrează.

FIFOs (numite și „*named pipes*") extind ideea la procese care nu au relație părinte-copil, canalul are un nume în sistemul de fișiere, procesele îl deschid când au nevoie [59]. Avantajul este simplitatea, nu trebuie gândit un protocol complex. Limita este modelul strict secvențial, flux de octeți fără mesaje discrete și fără adresare bidirecțională în același obiect.

Socketurile locale Unix sunt socketuri care comunică doar pe aceeași mașină, folosesc aceleași primitive ca socketurile de rețea, dar transferul se face în *kernel*. Există două moduri principale, *stream* (asemănător TCP) pentru fluxuri ordonate de octeți, și *datagram* (asemănător UDP) pentru mesaje discrete [60]. Avantajele sunt flexibilitatea, se pot folosi protocoale „cu mesaje", adresarea bidirecțională este firească, iar sistemul permite capabilități specifice. De exemplu transmiterea de *file descriptors* între procese sau autentificarea implicită a procesului prin credențiale Unix. Socketurile locale sunt adesea alegerea bună când se dorește o interfață clară, compatibilă cu un viitor protocol de rețea, dar cu performanță de mașină locală.

Figura 2.2 sintetizează **diferențele dintre cele trei mecanisme locale**: memorie partajată, conducte/cozi FIFO și socket local Unix.

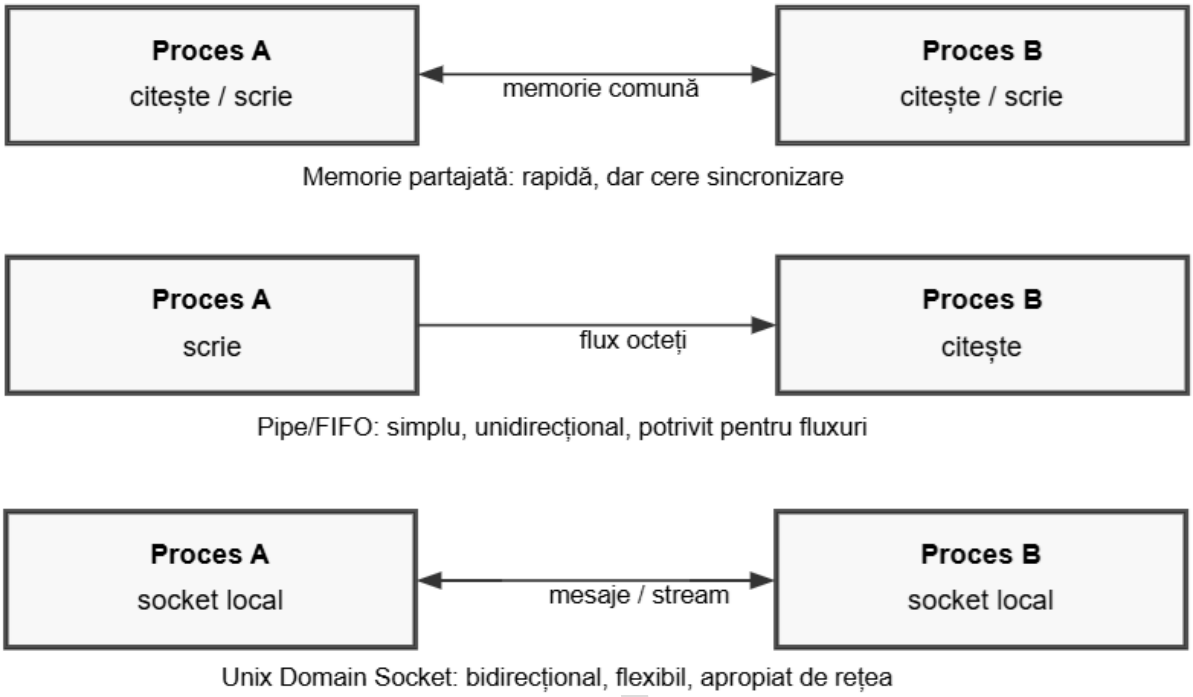


Figura 2.2. Mecanisme IPC locale (memorie partajată, conductă/FIFO și socket local Unix)

Alegerea mecanismului depinde de **debit**, **latență**, **simplitate** și **claritatea contractului**. Pentru volume mari și latență minimă se preferă memoria partajată sau *mmap*, dacă există un protocol de sincronizare simplu și bine documentat. Pentru conectarea liniară a două etape se folosesc conducte sau FIFO, deoarece sunt cele mai simple. Pentru dialog bidirecțional și contracte de mesaje se aleg socketurile locale Unix, deoarece sunt mai flexibile și mai ușor de migrat ulterior către comunicație prin rețea. În toate cazurile trebuie definit formatul datelor, cine inițiază schimbul, cum se semnalează lipsa de date și cum se raportează erorile, pentru a ușura depanarea.

Securitatea și **diagnosticul** sunt esențiale indiferent de mecanism, memoria partajată și *mmap* se protejează prin permisiuni pe obiectele *kernel* sau pe fișiere, socketurile locale și FIFO moștenesc permisiuni din sistemul de fișiere, iar pentru observabilitate este util un minim de convenții, identificatori de corelație în capetele de mesaj pe socketuri, semnături și contoare în antetul zonelor partajate pentru a detecta coruperi, .mesaje de jurnal cu marcaje de timp și dimensiuni de *buffer*.

Testele de integrare ar trebui să acopere cazurile cu "*buffer* plin", consumator indisponibil, reconectări și mesaje trunchiate, aceste situații apar frecvent în practică și se depistează mai ușor când există protocoale explicite și măsurători simple, latență, debit, lungime coadă, număr de erori la citire sau scriere.

2.2.2. Cozi de mesaje și transfer *zero-copy*

În unele situații, procesele nu trebuie doar să transmită un flux de octeți, ci să schimbe **mesaje persistate temporar și ușor de interpretat**. O coadă de mesaje introduce această structură: fiecare mesaj are o lungime, poate avea prioritate sau metadate, iar producătorul și consumatorul nu trebuie să ruleze în același ritm. Astfel, comunicarea devine mai clară decât în cazul unui flux simplu, iar procesele sunt mai bine decuplate.

Pentru volume mari de date, apare și **problema costului copierilor** între *buffere*. Tehnicile *zero-copy* urmăresc reducerea acestor copii inutile, astfel încât datele să poată fi transferate cu mai puțin consum de CPU și cu debit mai bun. Ele sunt importante mai ales în servere web, *streaming*, baze de date, brokeri de mesaje și alte sisteme unde cantitatea de date transmisă este mare.

O coadă de mesaje oferă un **mod mai structurat de comunicare** decât memoria partajată simplă. Procesele nu mai scriu direct în aceeași zonă de memorie, ci trimit mesaje printr-un mecanism gestionat de sistemul de operare. În POSIX, o coadă poate fi creată cu `mq_open`, mesajele pot fi trimise *cu* `mq_send`, iar citirea se face cu `mq_receive` [61].

```
#include <fcntl.h>
#include <mqueue.h>
#include <stdio.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

#define QUEUE_NAME "/demo_queue"
#define MAX_MSG_SIZE 128

int main(void) {
    struct mq_attr attr;
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = MAX_MSG_SIZE;
    attr.mq_curmsgs = 0;

    mqd_t queue = mq_open(QUEUE_NAME, O_CREAT | O_RDWR, 0600, &attr);

    if (queue == (mqd_t)-1) {
        perror("mq_open");
        return 1;
    }
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork");
        mq_close(queue);
        mq_unlink(QUEUE_NAME);
        return 1;
    }

    if (pid == 0) {
        const char* message = "mesaj trimis de procesul copil";
        mq_send(queue, message, strlen(message) + 1, 0);
        mq_close(queue);
        return 0;
    }

    char buffer[MAX_MSG_SIZE];
    mq_receive(queue, buffer, MAX_MSG_SIZE, NULL);

    printf("Parintele a primit: %s\n", buffer);
    wait(NULL);
    mq_close(queue);
    mq_unlink(QUEUE_NAME);
    return 0;
}
```

În exemplu, **procesul copil trimite un mesaj, iar procesul părinte îl primește din coadă**. Spre deosebire de memoria partajată, unde procesele trebuie să stabilească singure unde începe și unde se termină mesajul, coada păstrează delimitarea mesajelor. Fiecare `mq_send` introduce un mesaj distinct, iar fiecare `mq_receive` extrage un mesaj. Această proprietate simplifică protocolul dintre procese.

Coada de mesaje are însă propriile limite. Dimensiunea mesajului este limitată, numărul de mesaje din coadă este limitat, iar operațiile pot bloca atunci când coada este goală sau plină. În schimb, oferă o separare mai clară între producător și consumator: procesul care trimite mesajul nu trebuie să știe exact când va fi procesat acesta. Figura 2.3 arată **structura unei cozi de mesaje** locale, iar Figura 2.4 arată **succesiunea operațiilor de trimitere și preluare**.

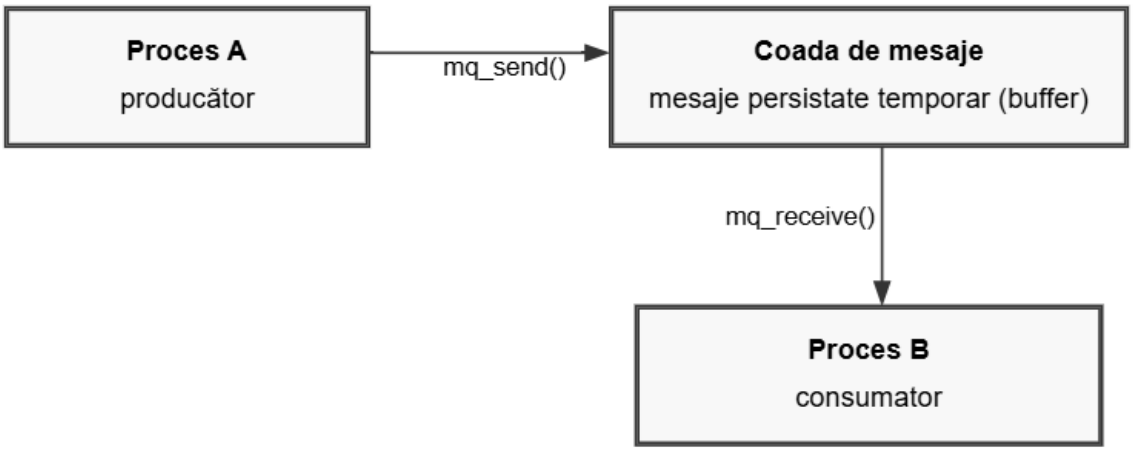


Figura 2.3. Coadă de mesaje locală (producător, coadă, consumator)

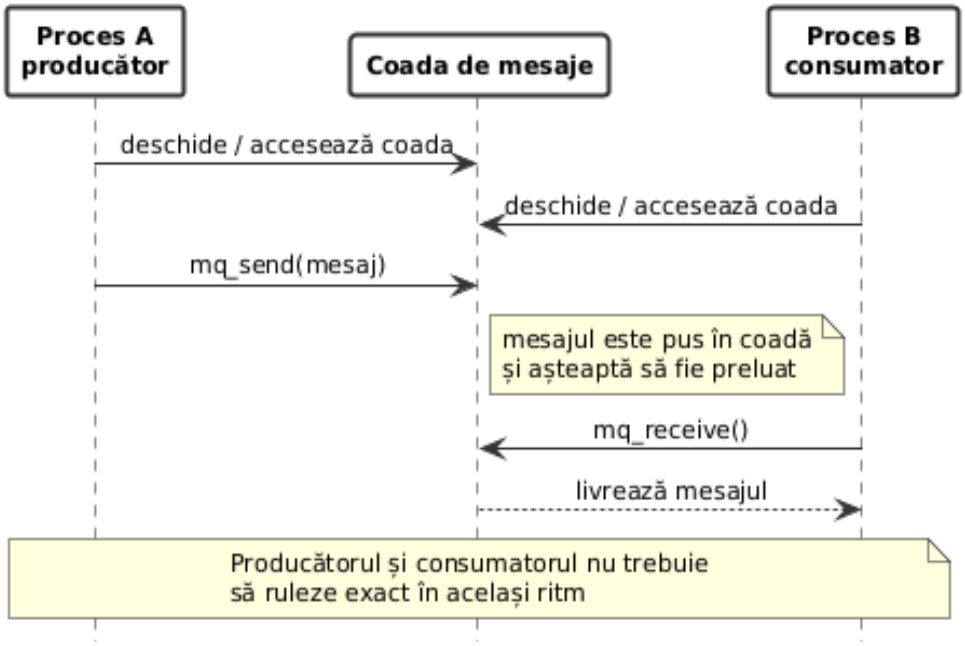


Figura 2.4. Interacțiunea într-o coadă de mesaje locală (*send*, stocare temporară și *receive*)

Cele două figuri evidențiază **diferența față de memoria partajată**. Procesele nu mai modifică direct aceeași zonă de memorie, ci comunică prin mesaje păstrate temporar în coadă. Coada devine un punct intermediar care absoarbe diferențele de ritm dintre producător și consumator.

În practică, mesajul are de obicei un antet minim, de exemplu tip, lungime și identificator de corelație pentru diagnostic, urmat de încărcătura utilă. Avantajul este claritatea contractului: se lucrează cu mesaje, nu cu fluxuri brute de octeți. Acest lucru simplifică testarea și depanarea, deoarece fiecare mesaj poate fi observat, validat și salvat în jurnal ca unitate separată.

Coada poate fi gestionată de *kernel*, de exemplu **POSIX** sau **System V**, sau poate fi construită în spațiul utilizator, de exemplu în memoria partajată. Cozile din *kernel* oferă API simplu și blocare automată la capete: producătorul poate fi pus în așteptare când coada este plină, consumatorul când coada este goală. Costul este trecerea prin apeluri de sistem pentru fiecare operație. Cozile în spațiul utilizator pot reduce acest cost, dar cer reguli stricte despre cine scrie, cine citește, cum se marchează pozițiile valide și ce bariere de memorie se folosesc.

Politica de control al presiunii din amonte, *backpressure*, este esențială. Când consumatorul rămâne în urmă, coada trebuie să limiteze producția, altfel memoria se umple. Variantele obișnuite sunt blocarea producătorului, respingerea mesajelor noi cu un cod explicit sau degradarea controlată, de exemplu aruncarea celor mai vechi mesaje necritice. Alegerea trebuie să fie vizibilă în contract, iar în jurnal și prin metrici se urmăresc dimensiunea cozii, numărul de mesaje respinse, latența și debitul.

Prioritizarea și **ordonarea** se stabilesc la nivel de coadă. O coadă simplă păstrează ordinea de sosire, FIFO, o coadă cu priorități alege următorul mesaj după o cheie numerică. Dacă aplicația are atât mesaje critice cât și mesaje voluminoase, merită separarea în cozi diferite. Astfel se evită blocajul în care un mesaj mare întârzie multe mesaje mici și urgente. Pentru operații sensibile, consumatorul trebuie proiectat idempotent, adică același mesaj procesat de două ori să producă același efect, ceea ce permite retransmisii mai sigure.

Transferul *zero-copy* este tehnica prin care datele trec între fișiere, socketuri sau procese fără a fi copiate din *kernel* în spațiul utilizator și înapoi. Câștigul este reducerea numărului de copii de memorie și a comutărilor între moduri, scade utilizarea CPU și crește debitul. În Linux se folosesc apeluri precum sendfile sau splice [62], [63]. Regula practică este simplă, dacă aplicația doar „cară" octeți între două capete, folosește *zero-copy*, dacă trebuie să inspecteze sau să modifice încărcătura, acceptă copiile necesare și minimizează-le, de exemplu cu `writev` într-o singură scriere. Câștigul este relevant mai ales pentru blocuri mari de date, pentru mesaje foarte mici avantajul poate fi neglijabil.

Totuși, *zero-copy* nu trebuie privit ca un mecanism general care simplifică programarea. De multe ori, el complică durata de viață a *bufferelor*, sincronizarea și portabilitatea. Este util mai ales în sisteme cu trafic intens, servere web, streaming, baze de date, brokeri de mesaje sau procesare de fișiere mari. Pentru aplicații obișnuite, claritatea protocolului și corectitudinea sunt mai importante decât eliminarea unei copii de memorie.

În ansamblu, **coada** definește „ce" și „când" se livrează, iar ***zero-copy*** optimizează „cum" se transportă. Mecanismele IPC sunt potrivite atunci când procesele colaborează pe aceeași mașină sau în același sistem de operare. Pe măsură ce aplicațiile cresc, apare însă o altă formă de integrare: mai multe aplicații folosesc aceeași bază de date sau schimbă informații prin tabele comune. Această variantă pare simplă la început, dar ridică probleme de cuplare, tranzacții, evoluția schemei și responsabilitate asupra datelor.

2.3. Integrare prin baze de date

Integrarea prin baze de date apare natural **atunci când mai multe componente au nevoie de aceleași informații**. Este o cale tentantă pentru că datele par „centrul stabil" al aplicației. În practică trebuie clarificat ce se obține când diferite părți colaborează prin tabele comune și ce riscuri apar când baza de date devine locul de schimb de mesaje, versiuni și convenții [32].

2.3.1. Baza de date ca punct de integrare, avantaje și limite

Baza de date ca punct de integrare înseamnă că mai **multe aplicații sau module se sincronizează prin aceeași schemă, scriu și citesc aceleași tabele** și **interpretează aceleași coloane**. Avantajul este simplitatea la început, nu se proiectează contracte suplimentare, nu se ridică infrastructură de mesaje, iar rapoartele se pot construi direct peste modelul comun. Pentru date relativ statice și consum în mod preponderent de citire, această abordare poate fi suficientă. Figura 2.5 compară această integrare prin schemă comună cu varianta în care datele sunt deținute de fiecare componentă și accesate prin contracte explicite.

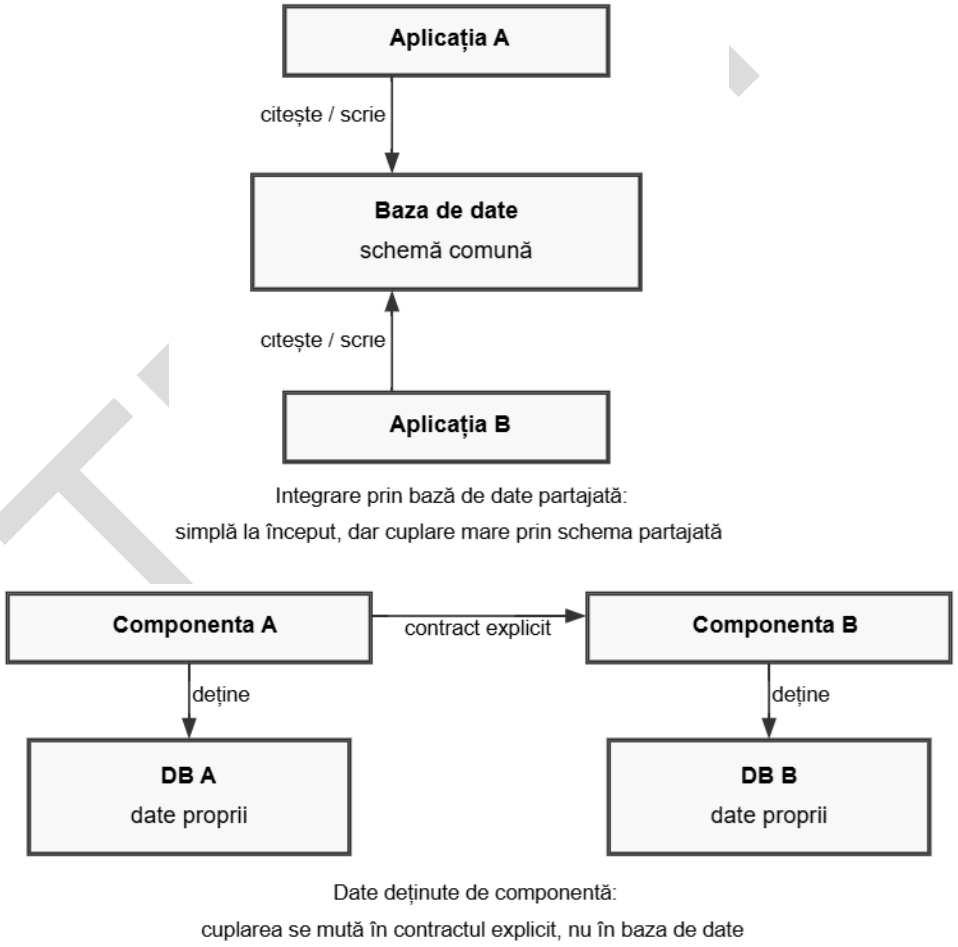


Figura 2.5. Integrare prin bază de date partajată față de date deținute de componentă

Se observă **diferența** dintre două abordări. În prima, **baza de date devine interfața implicită** dintre aplicații, iar **schema comună ajunge să fie contractul real**. În a doua, **fiecare**

componentă își deține propriile date și **colaborează cu celelalte prin contracte explicite**, API-uri sau mesaje. Prima variantă este simplă la început, dar poate produce cuplare strânsă. A doua cere mai multă proiectare, dar permite o evoluție mai controlată.

Limitele apar când echipele și ritmul de schimbare cresc. Evoluția unei coloane sau a unei reguli afectează toate componentele care o folosesc, apar dependențe invizibile între tabele și cod, iar o refactorizare minoră devine migrare dificilă. Cazul tipic problematic este *anti-pattern*-ul „bază de date partajată" (*shared DB*, mai multe aplicații scriu în aceeași schemă), în care aplicații separate își sincronizează pașii prin aceleași tabele, rezultând cuplare strânsă și schimbări de schemă care cer migrare coordonată [32], [22]. Efectul este cuplare strânsă, versiuni care se blochează reciproc și imposibilitatea de a aplica politici diferite de securitate sau performanță pe subsisteme diferite.

O alternativă sănătoasă este ca fiecare componentă să dețină modelul său și să expună contracte explicite pentru colaborare, iar baza de date să fie locul unde persistă starea acelei componente, nu și interfața lui către ceilalți. Această separare mută însă problema de la acces comun la date spre coordonarea schimbărilor. De aceea, trebuie înțeles ce poate garanta o tranzacție locală și ce devine dificil atunci când o acțiune traversează mai multe componente.

2.3.2. Tranzacții și izolare, ce înseamnă ACID în integrare

O **tranzacție** este un **grup de operații care se tratează ca o singură unitate de lucru**. Imaginea clasică este transferul între două conturi, se scade dintr-unul și se adaugă în celălalt, iar rezultatul este corect doar dacă ambii pași reușesc împreună. Proprietățile **A.C.I.D.** explică ce promite o bază de date când confirmă o tranzacție. **Atomicitate** înseamnă că fie se aplică toate modificările, fie se anulează toate, astfel nu rămâne contul sursă scăzut fără ca destinația să fie crescută. **Consistență** înseamnă că regulile declarate rămân adevărate după confirmare, de exemplu soldul nu devine negativ dacă regula interzice acest lucru. **Izolare** înseamnă că tranzacțiile par să ruleze pe rând, chiar dacă în realitate se suprapun, astfel fiecare vede o stare coerentă. **Durabilitate** înseamnă că, după confirmare, efectele nu se pierd la o pană de curent sau la repornirea serverului.

Izolarea merită o explicație separată, pentru că aici apar compromisurile între corectitudine și performanță. Nivelurile de izolare controlează cât de mult se „văd" tranzacțiile între ele. La izolare scăzută pot apărea citiri alterate, o tranzacție citește modificări neconfirmate ale alteia. La izolare intermediară dispar citirile alterate, dar pot apărea citiri ne repetabile, aceeași interogare dă rezultate diferite în cadrul aceleiași tranzacții, deoarece altcineva a modificat rândul între timp. La izolare înaltă se evită și fenomenul numit *phantom*, apar sau dispar rânduri care corespund aceleiași condiții. În practică se alege un nivel care protejează invariantele reale ale aplicației, apoi se măsoară impactul asupra concurenței și se ajustează interogările și indecșii pentru a reduce blocajele.

Cât timp pașii unei acțiuni ating entități din aceeași bază de date, o tranzacție locală ACID este soluția naturală. Problemele apar când pașii traversează două componente sau două baze de date. Coordonarea strictă prin **confirmare în două faze** asigură atomicitate peste granițe, dar adaugă latență, blocări și fragilitate operațională. De aceea, în arhitecturi moderne se preferă altă abordare, fiecare componentă își protejează invarianta locală cu tranzacții obișnuite, iar colaborarea se face prin contracte explicite și mecanisme de refacere, comenzi idempotente care pot fi reluate fără efecte secundare, evenimente care propagă schimbările, operații compensatorii care anulează un pas anterior dacă un pas ulterior eșuează, plus verificări ulterioare ale integrității [22].

2.3.3. Schimb de date fiabil și modele moderne

Vom trece în revistă **patru modele care asigură schimb de date fiabil între componente**, când se folosesc și cum se combină, cu accent pe evitarea tranzacțiilor distribuite și pe trasabilitate. Figura 2.6 și Figura 2.7 sintetizează ideea comună a acestor modele: componenta își actualizează mai întâi propria stare printr-o tranzacție locală, iar schimbarea este propagată ulterior prin evenimente sau mesaje.

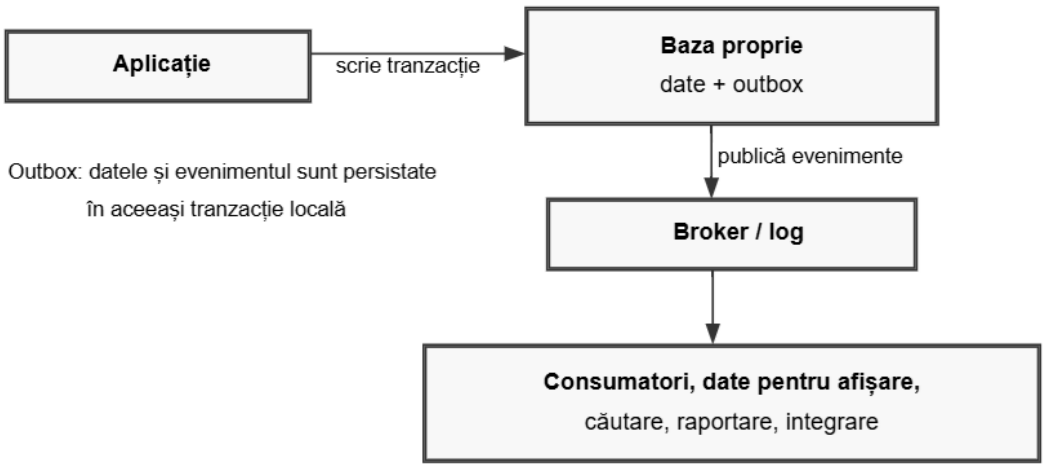


Figura 2.6. Schimb de date fiabil (tranzacție locală, *outbox* și publicare de evenimente)

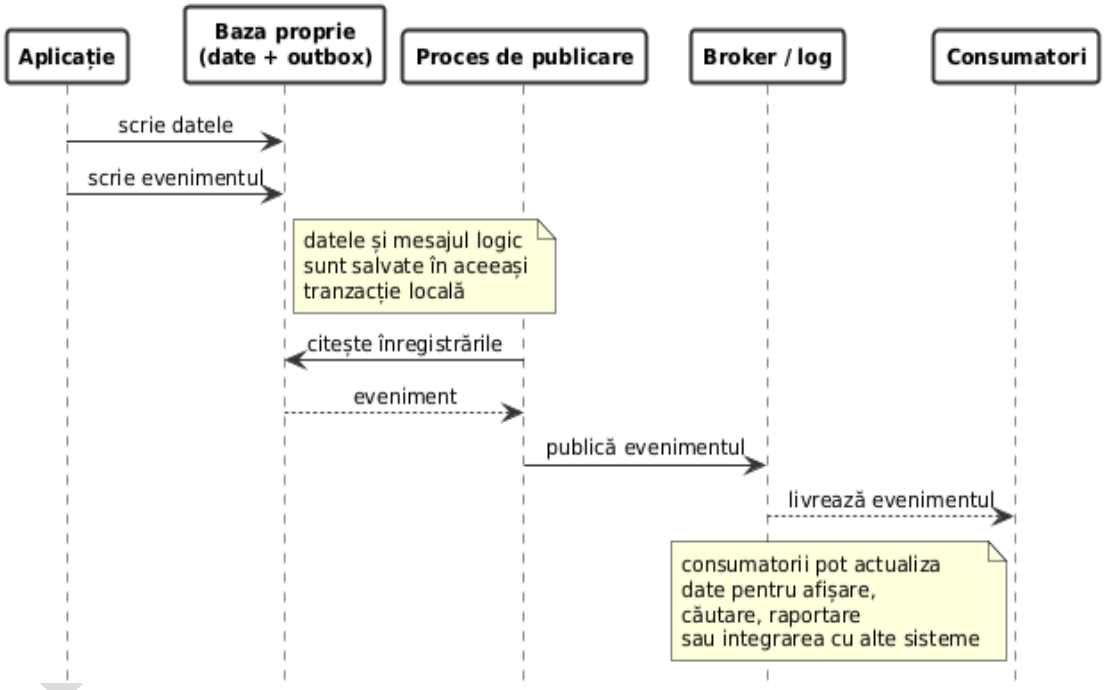


Figura 2.7. Interacțiunea într-un schimb de date fiabil (scriere locală, publicare și consum al evenimentului)

Cele două figuri evidențiază aceeași idee din două perspective. Mai întâi, componenta își protejează starea în propria bază de date, printr-o tranzacție locală. Apoi, schimbarea este propagată către alte componente prin evenimente sau mesaje. Astfel se evită tranzacțiile distribuite grele, dar se păstrează trasabilitatea și posibilitatea de refacere a fluxului.

***Change Data Capture* (CDC)** este tehnica prin care schimbările dintr-o bază de date sunt observate și trimise mai departe ca evenimente. În loc ca aplicațiile să facă interogări periodice, CDC citește jurnalul de tranzacții al bazei, detectează inserări, actualizări și ștergeri, apoi publică evenimente corespunzătoare. Avantajul este latența mică și consumul redus de resurse, deoarece nu se mai scanează tabele la intervale fixe. CDC este util când datele trebuie replicate către un motor de căutare sau un alt serviciu care își menține date pentru afișare/interogare rapidă. Limita importantă este că CDC vede ce s-a schimbat, nu de ce s-a schimbat, și nu impune singur un contract semantic. De aceea, evenimentele rezultate trebuie documentate și versionate.

Outbox/Inbox este un *pattern* operațional care asigură că situația „am scris în baza mea și am trimis un eveniment" nu se desincronizează. La scriere, aplicația pune modificarea în propriile tabele și salvează un mesaj în tabelul *outbox*, în aceeași tranzacție locală. Un proces dedicat publică acele mesaje către broker, iar consumatorii le preiau și le marchează în tabele *inbox*, pentru a evita procesarea dublă. Beneficiul este atomicitatea locală fără tranzacții distribuite: dacă tranzacția reușește, mesajul există sigur în *outbox* și va fi publicat; dacă eșuează, nu se trimite nimic. În practică, acest model ajută și la idempotentă, deoarece fiecare mesaj are un identificator, iar consumatorul poate verifica dacă l-a procesat deja [22].

CQRS (*Command--Query Responsibility Segregation*) **s**epară scrierile de citiri. Comenzile schimbă starea printr-un model optimizat pentru consistență și reguli, iar interogările citesc din structuri optimizate pentru performanță, afișare, cache sau denormalizări. Avantajul este că fiecare parte poate evolua independent: schema pentru scrieri rămâne curată și strictă, iar structura pentru citire poate fi adaptată pentru interfețe grafice sau rapoarte fără a afecta logica tranzacțională. Dezavantajul este apariția consistenței eventuale: după o scriere, datele pentru citire pot avea o întârziere scurtă până când primesc evenimentul și se actualizează. Pentru începători, regula simplă este să folosească CQRS când citirile sunt mult mai numeroase decât scrierile sau au nevoi foarte diferite de formatare [32], [42].

Event Sourcing păstrează istoricul ca o secvență de evenimente imutabile, iar starea curentă se calculează prin aplicarea lor în ordine. În loc să se salveze doar „starea finală a comenzii", se salvează evenimente precum „Comandă creată", „Articol adăugat", „Plată confirmată" etc. Beneficiile sunt auditul complet, posibilitatea de a reconstrui starea la orice moment din trecut și refacerea după erori prin re-redarea jurnalului. Costurile sunt complexitatea mai mare de modelare, necesitatea migrațiilor controlate la schimbări de schemă de eveniment și nevoia frecventă de snapshots pentru a accelera reconstruirea stării. Pentru multe sisteme, o combinație pragmatică funcționează mai bine: scrieri clasice și evenimente derivate pentru integrare, fără a merge până la *Event Sourcing* complet [42].

Outbox oferă mecanismul sigur pentru publicarea evenimentelor după o tranzacție locală. CDC poate observa aceleași schimbări direct din bază când nu se dorește modificarea aplicației, dar necesită grijă semantică pentru ca evenimentele să fie ușor de înțeles și de versionat. Evenimentele publicate pot alimenta structuri de citire în CQRS, permițând interogări rapide și independente. Dacă se adoptă *Event Sourcing*, aceste structuri de citire se construiesc nativ din jurnalul de evenimente. Indiferent de combinație, cheile rămân contractele clare ale evenimentelor, identificatorii de deduplicare și observabilitatea, prin înregistrări structurate de jurnal și corelarea mesajelor pe întregul flux, pentru a demonstra că „am scris, am publicat, s-a consumat, s-a aplicat”.

Pentru problema „am nevoie să trimit în mod fiabil un eveniment după ce am salvat în baza mea”, *outbox* este prima alegere. Când aplicația existentă nu poate fi modificată, CDC devine o cale de integrare, cu atenție la semantica evenimentelor. Dacă citirile au nevoi foarte diferite față de scrieri sau trebuie scalate separat, CQRS poate aduce claritate și flexibilitate pe termen lung.

2.3.4. *SQLite* ca bază de date locală integrată în aplicație

SQLite merită menționat separat deoarece ocupă o poziție diferită față de serverele clasice de baze de date. **Nu rulează ca proces server separat**, ci **este o bibliotecă inclusă în aplicație**. Baza de date este, de regulă, un fișier local, iar aplicația folosește SQL, tabele, indecși și tranzacții fără să instaleze un server de baze de date. De aceea, *SQLite* este întâlnit în **aplicații mobile**, aplicații desktop, **dispozitive *embedded***, instrumente locale, prototipuri și teste [64].

Această diferență îl face util ca **exemplu de integrare locală**. Aplicația nu comunică prin rețea cu un server, ci apelează o bibliotecă. Totuși, din punct de vedere conceptual, rămân valabile ideile discutate anterior: schema este un contract, tranzacțiile protejează invariantele, iar schimbările de structură trebuie gestionate prin migrații. Figura 2.8 arată această diferență de execuție: *SQLite* este apelat ca motor integrat în aplicație, iar datele sunt citite și scrise într-un fișier local.

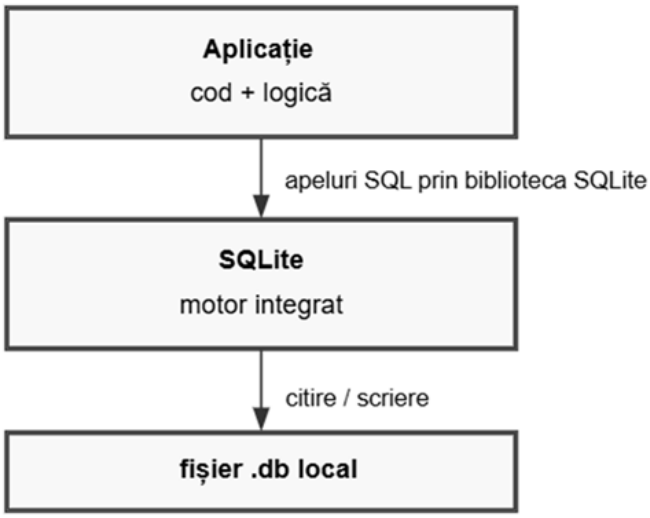


Figura 2.8. *SQLite* (motor de baze de date integrat în aplicație, fără server separat)

Un **exemplu minimal în Java**, prin JDBC, poate crea o bază locală, introduce un rând și îl citește. Codul presupune că driverul *SQLite* JDBC este disponibil în proiect.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class SQLiteDemo {

    public static void main(String[] args) throws Exception {

        // definesc URL-ul JDBC catre fisierul local SQLite
        String url = "jdbc:sqlite:demo.db";

        // deschid conexiune si creez un obiect pentru executare instructiuni SQL
        try (Connection conn = DriverManager.getConnection(url);
             Statement st = conn.createStatement()) {

            // creez tabela accounts daca nu exista deja
            st.executeUpdate("CREATE TABLE IF NOT EXISTS accounts (" +
```

```
                    "id INTEGER PRIMARY KEY, " +
                    "balance REAL NOT NULL)");

                // inserez sau inlocuiesc un cont de test
                st.executeUpdate("INSERT OR REPLACE INTO accounts(id, balance) " +
                    "VALUES (1, 100.0)");

                // execut interogarea pentru citirea conturilor
                try (ResultSet rs = st.executeQuery(
                        "SELECT id, balance FROM accounts")) {

                    // parcurg rezultatele si afisez fiecare rand
                    while (rs.next()) {
                        System.out.println(
                            rs.getInt("id") + ": " + rs.getDouble("balance")
                        );
                    }
                }
            }
        }
    }
```

Exemplul arată că **aplicația folosește SQL obișnuit, dar baza de date este locală**. Nu există un server separat, un port de rețea sau un proces de administrat. Fișierul `demo.db` conține tabelele și datele. Blocul `try-with-resources` închide automat conexiunea, instrucțiunea SQL și rezultatul interogării, ceea ce reduce riscul de resurse rămase deschise. Pentru aplicații mici sau locale, acest model este foarte practic.

Totuși, *SQLite* nu trebuie confundat cu o soluție de integrare între mai multe servicii independente. Dacă mai multe aplicații încep să scrie direct în același fișier de bază de date, apar probleme asemănătoare cu baza de date partajată: cuplare prin schemă, blocaje și migrații coordonate. Folosit sănătos, *SQLite* este baza locală a unei aplicații sau componente, nu interfața ascunsă dintre sisteme diferite.

Tranziția spre secțiunile următoare este importantă. Integrarea prin baze de date este potrivită când datele persistente sunt centrul problemei. Când însă interacțiunea principală este apelul unei operații la distanță, intrăm în zona RPC și RMI. Acolo contractul nu mai este tabela, ci interfața apelată la distanță.

2.4. RPC și RMI (integrare sincronă)

După execuția locală în același proces, după comunicarea între procese pe aceeași mașină și după integrarea prin baze de date, următorul pas este apel**ul direct la capacități expuse de alt proces, ca și cum ar fi funcții locale**. Asta înseamnă ***Remote Procedure Call*** (**RPC**), respectiv ***Remote Method Invocation*** (**RMI**) în Java, integrare sincronă în care apelantul așteaptă răspunsul [65]. Avantajul este simplitatea mentală, codul pare local. Costul este cuplarea la contractul sincron și la caracteristicile rețelei, latență, erori de transport, depășirea timpului de așteptare.

În continuare sunt clarificate conceptele, apoi ilustrate cu un exemplu minim în Java RMI. Figurile 2.9 și 2.10 surprind aceeași idee din două perspective complementare: **structura apelului** și **succesiunea interacțiunilor**. Figura 2.9 arată **iluzia principală a RPC/RMI**. Clientul apelează o metodă ca și cum ar fi locală, dar apelul trece printr-un *stub*, este serializat, transmis prin rețea, reconstruit pe server și executat de serviciul real. Figura 2.10 detaliază ordinea acestor pași: apelul clientului, serializarea (*marshalling*) parametrilor, transportul mesajului, deserializarea (*unmarshalling*) pe server, execuția metodei și întoarcerea rezultatului.

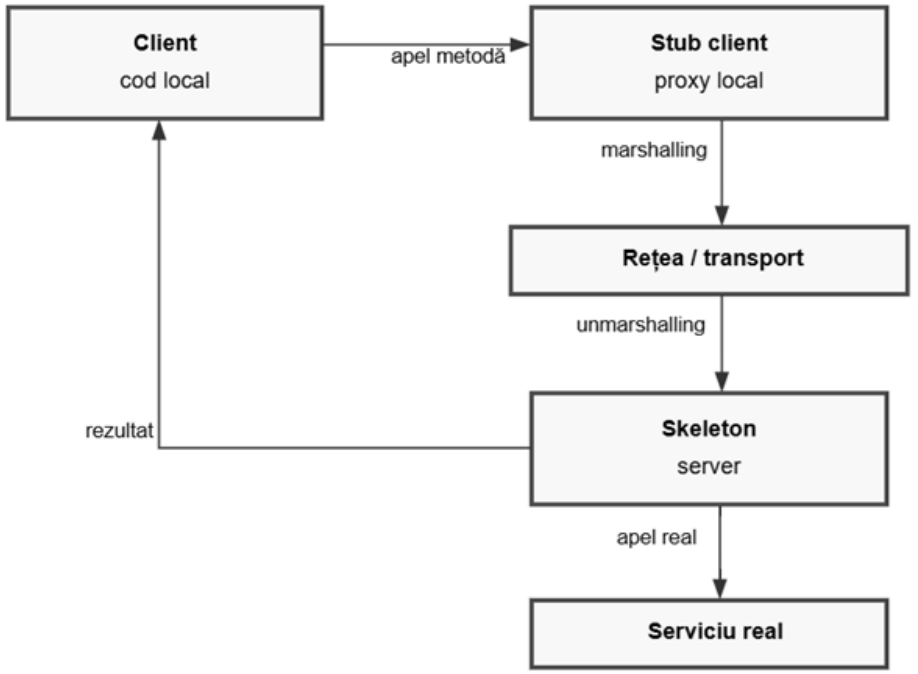


Figura 2.9. Structura unui apel RPC/RMI (apel aparent local, execuție într-un proces la distanță)

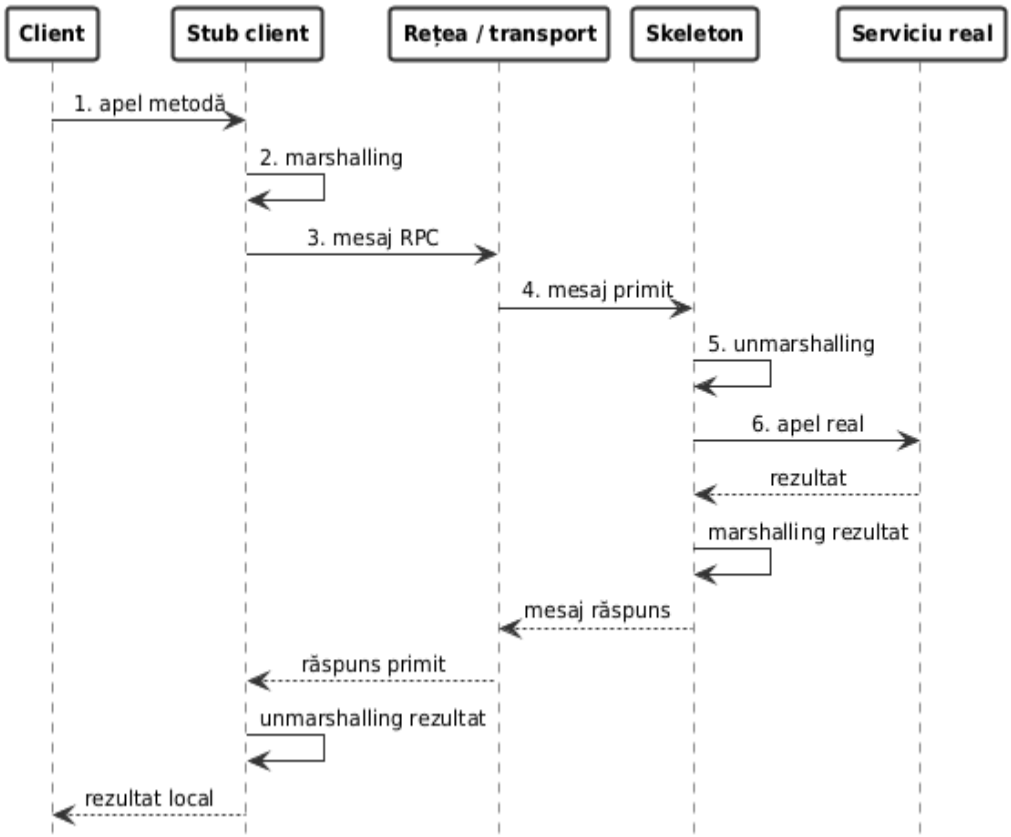


Figura 2.10. Interacțiunea într-un apel RPC/RMI (de la apelul clientului la răspuns)

Această iluzie este utilă, dar poate deveni periculoasă dacă este luată prea literal. Un apel la distanță nu are aceleași proprietăți ca un apel local. Poate dura mai mult, poate eșua din cauza rețelei, poate fi executat de două ori în cazul unei retransmisii sau poate să fi fost executat pe server chiar dacă răspunsul nu mai ajunge la client. De aceea, interfețele la distanță trebuie proiectate cu atenție: operații clare, erori declarate, depășirea timpului de așteptare, reguli de retransmisie și idempotentă.

2.4.1. Principalele concepte ilustrate în Java

Un **apel la distanță,** ***Remote Procedure Call*** (RPC), face ca **invocarea unei metode de pe o altă mașină** să **arate ca un apel local**. Pentru ca această iluzie să funcționeze, sistemul se sprijină pe trei piese, un contract clar, transformarea datelor pentru transport cu serializare și deserializare, respectiv obiecte de legătură pe fiecare capăt, *stub* pe client și *skeleton* pe server.

Contractul. În Java, contractul este de regulă o interfață care descrie operațiile, parametrii, tipul rezultatului și erorile ce pot apărea. Contractul este ceea ce generează codul de legătură și stabilește compatibilitatea între părți.

De **exemplu**, pentru un **calculator simplu**:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

// contractul RPC: nume metoda, tipuri, erori declarate
public interface Calculator extends Remote {
    int add(int a, int b) throws RemoteException;
}
```

Interfața extinde `Remote`, fiecare metodă declară `throws RemoteException`, semnalând că erorile de rețea fac parte din contract. Aceasta este „forma” pe care o cunosc și clientul, și serverul.

Transformarea datelor pentru transport. Parametrii și rezultatul nu pot circula ca „variabile Java”, ei sunt transformați într-o reprezentare de transmis, serializare, apoi reconstruiți la destinație, deserializare. Reprezentarea poate fi textuală, de exemplu JSON, sau binară. Intuitiv, un apel la `add(7, 5)` ar putea fi echivalat cu un mesaj textual de forma:

```
{ "method": "add", "params": { "a": 7, "b": 5 } }
```

iar răspunsul cu:

```
{ "result": 12 }
```

În Java RMI, această codificare este binară și transparentă pentru programator, ideea rămâne aceeași, se împachetează valorile la plecare și se despachetează la sosire.

Stub **pe client.** Pe partea de client, un obiect *stub* se prezintă ca implementare a interfeței `Calculator`, însă în loc să execute local metoda, el serializează parametrii, trimite mesajul prin rețea, așteaptă răspunsul și îl deserializează. Pentru codul client, linia `calc.add(7, 5)` pare un apel obișnuit, doar că trece prin *stub* care ascunde detaliile de transport.

Skeleton **pe server.** Pe partea de server, un *skeleton* primește mesajele, deserializează parametrii în valori Java, apelează implementarea reală a contractului, de exemplu `CalculatorImpl`, apoi serializează rezultatul și îl trimite înapoi. Programatorul scrie logica de domeniu, *skeleton*-ul se ocupă de conectarea la rețea.

De remarcat faptul că ***Stub*-ul este** un ***proxy* pe partea de client** pentru serviciul la distanță, adică o „față locală” a serviciului de pe server. Codul clientului apelează *stub*-ul ca pe o implementare obișnuită, iar acesta serializează parametrii, trimite mesajul și deserializează răspunsul. ***Skeleton*-ul este** componenta de **pe partea de serverului** care primește invocarea venită din rețea, deserializează parametrii în valori Java, invocă implementarea reală și serializează rezultatul înapoi către client. El este un ***proxy*** pentru client, adică reprezentantul cererilor venite din rețea. Figura 2.11 arată că, indiferent de tehnologia folosită, RMI, gRPC, Thrift sau alt RPC, pașii conceptuali rămân asemănători [65], [66].

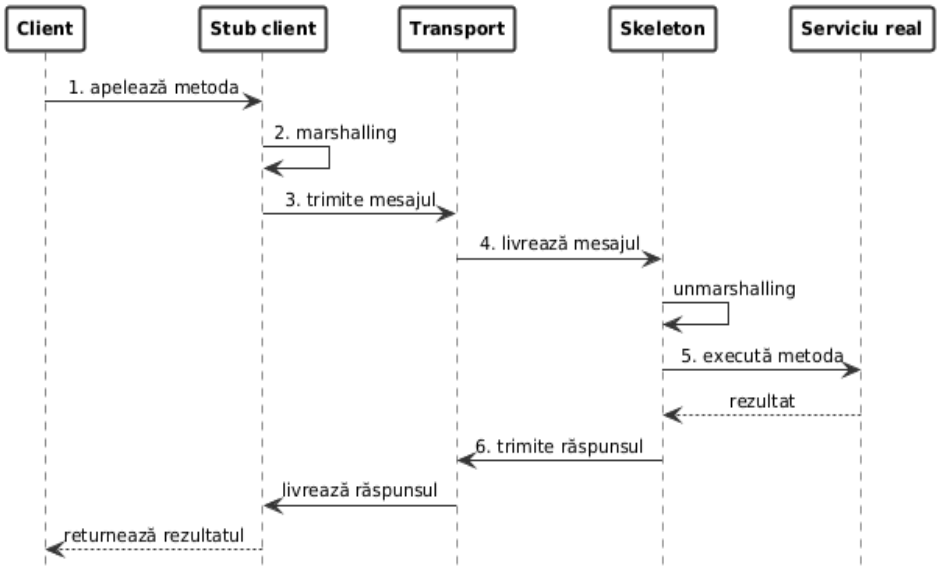


Figura 2.11. Pașii unui apel RPC/RMI (de la apelul clientului la răspuns)

Implementarea pe server pentru calculatorul simplu:

```
// CalculatorImpl.java
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class CalculatorImpl extends UnicastRemoteObject implements Calculator {
    // constructorul expune obiectul ca remote
    protected CalculatorImpl() throws RemoteException { super(); }
    @Override
    public int add(int a, int b) throws RemoteException {
        // logica reala a serverului, aici este triviala
        return a + b;
    }
}
```

Aici `UnicastRemoteObject` asigură *skeleton*-ul, adică partea care primește apelurile la distanță și le direcționează către metoda reală. Implementarea conține doar logica, fără cod de rețea.

Pornirea serverului și publicarea în registru pentru calculatorul simplu:

```
// Server.java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
```

```
public class Server {
    public static void main(String[] args) throws Exception {
        // porneste un registry local pe portul 1099 (daca nu exista deja)
        LocateRegistry.createRegistry(1099);

        // creeaza implementarea si o publica in registry cu un nume cunoscut
        Calculator calc = new CalculatorImpl();
        Registry reg = LocateRegistry.getRegistry();
        reg.rebind("calc", calc);
        System.out.println("Server RMI pornit, serviciul 'calc' e disponibil");
    }
}
```

Registry este ca o agendă telefonică simplă pentru programe, ține o listă cu nume de servicii și adresele lor de rețea. Serverul notează numele în această agendă, iar clientul caută după nume și primește un *stub* (adică un obiect reprezentant care știe cum să apeleze serviciul la adresa corectă).

Fluxul cap la cap: clientul obține un *stub* pentru Calculator dintr-un registru, apelează `add(7, 5)`, *stub*-ul face serializare și expediază mesajul, *skeleton*-ul de pe server face deserializare și invocă implementarea, rezultatul `12` este serializat și returnat, *stub*-ul îl deserializează și îl livrează ca un `int` local. Toate acestea se bazează pe contractul comun, care garantează că semnăturile, tipurile și regulile de eroare sunt cunoscute de ambele părți.

La nivel de infrastructură Java, **două utilitare clasice pun „în mișcare" exemplul** de mai sus: ***rmiregistry*** și ***rmic***. Registrul poate fi pornit ca proces separat cu *rmiregistry* sau încorporat în server prin `LocateRegistry.createRegistry(1099)`, efectul este același, publicarea numelui serviciului și expunerea adresei la care clienții găsesc *stub*-ul. Generarea *stub*-urilor a evoluat în timp, în versiunile vechi se rula ***rmic*** peste clasele la distanță pentru a produce fișiere *stub* și, uneori, *skeleton*, acestea erau livrate clientului și încărcate la rulare, începând cu Java 5, pentru protocolul JRMP, stub-urile se generează **dinamic** la *runtime*, astfel în multe proiecte *rmic* nu mai este necesar, dar rămâne relevant pentru scenario de compatibilitate.

În practică, pașii sunt simpli, se pornește registrul, serverul exportă obiectul la distanță și îl înregistrează sub un nume, clientul face `lookup("calc")` și primește un *stub* care vorbește protocolul potrivit. Dacă se folosesc clase transmise prin rețea, este important ca ***classpath*-ul** să conțină aceleași definiții de tip pe ambele părți, altfel apare `ClassNotFoundException`.

Din punct de vedere operațional, portul 1099 al registrului și porturile pe care `UnicastRemoteObject` publică obiectele trebuie deschise în *firewall*, iar erorile de transport se tratează explicit în contract, `RemoteException` în RMI sau coduri standardizate în alte RPC-uri, pentru a face depanarea și testarea reproductibile.

Un apel la distanță poate eșua din două motive, logică de aplicație sau transport. De aceea interfețele la distanță declară erori specifice, în Java RMI `throws RemoteException`, iar în alte RPC-uri se trimit coduri de eroare standardizate. Contractul trebuie să documenteze aceste situații, împreună cu comportamentele așteptate la depășirea timpului de așteptare și la reîncercare, altfel testarea și depanarea devin dificile. În practică, un apel la distanță are latență și incertitudine, spre deosebire de un apel local, iar acest lucru trebuie reflectat în designul interfeței, de exemplu parametri idempotenți, rezultate paginate și erori consistente.

Pentru un exemplu didactic, RMI este util deoarece face vizibilă ideea de obiect la distanță. Totuși, în aplicațiile moderne, același model conceptual apare și sub alte forme: HTTP cu JSON, gRPC cu *Protobuf*, Thrift sau servicii interne cu contracte generate. Diferențele țin de formatul datelor, transport, generarea codului și regulile de compatibilitate. Ideea comună rămâne aceeași: clientul lucrează cu un contract, iar infrastructura transformă apelul local aparent într-un schimb de mesaje prin rețea.

2.4.2. Familii RPC și limbaje de descriere a interfețelor (IDL)

În practică există mai multe **„familii" de RPC** care diferă prin felul în care descriu contractele, codifică datele și transportă mesajele. Un ***Interface Description Language*** (**IDL**) este un limbaj neutru față de limbaje de programare, folosit pentru a descrie public, clar și verificabil interfețele unui serviciu, adică numele operațiilor, parametrii și tipurile lor, rezultatele, erorile, regulile de compatibilitate. Un IDL bun permite generarea automată de cod e

legătură, *stubs* pe client și *skeletons* pe server, în mai multe limbaje. Astfel, clientul și serverul vorbesc aceeași „gramatică" fără a scrie manual partea de transport.

Exemple uzuale sunt: WSDL pentru SOAP descrie operații și mesaje pe bază de XML, *Protocol Buffers* pentru gRPC definesc servicii și tipuri în fișiere `.proto` din care se generează cod, *Thrift* are propriul IDL din care se generează clienți și servere în multe limbaje, CORBA IDL și DCE IDL au servit același scop în sisteme istorice [39], [66], [67]. Fără un IDL, contractele se documentează doar în text și riscul de nepotrivire între părți crește, cu un IDL, contractul devine o sursă unică de adevăr din care derivă cod, teste și documentație. Figura 2.12 sintetizează acest rol al IDL-ului în RPC.

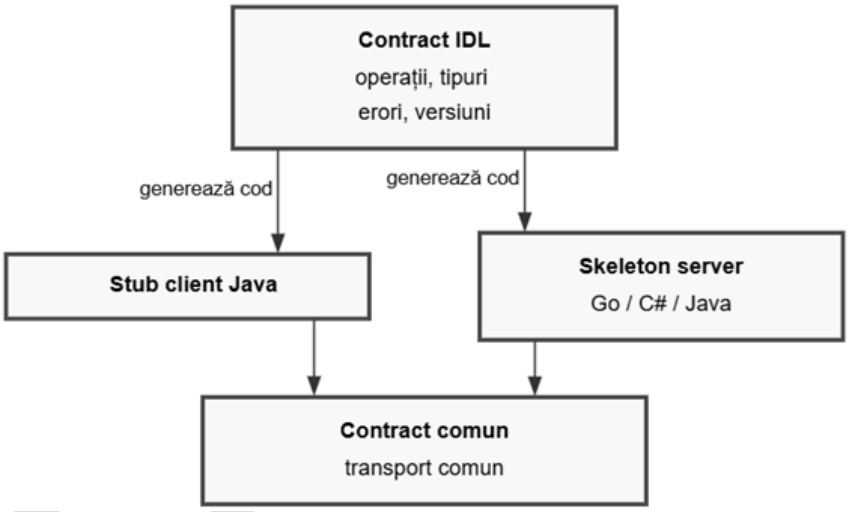


Figura 2.12. IDL în RPC (contract comun și cod generat pentru mai multe limbaje)

Contractul este scris o dată, apoi se generează cod de legătură pentru client și server, uneori în limbaje diferite. Astfel, compatibilitatea nu depinde de interpretări informale, ci de o descriere verificabilă a operațiilor și tipurilor.

Istoric, ONC RPC și DCE RPC au definit IDL-uri și *runtime*-uri binare folosite mult în sisteme Unix și Windows, CORBA a dus ideea mai departe cu un IDL neutru de limbaj și IIOP, iar în zona web au apărut SOAP și XML-RPC, apoi variante mai ușoare precum JSON-RPC. În prezent, pentru performanță și contracte stricte se folosesc des *Thrift* și *gRPC*, care combină IDL-uri concise cu transporturi eficiente. Mai jos detaliem pe scurt SOAP și XML-RPC, respectiv JSON-RPC, deoarece sunt ușor de înțeles pentru un început, iar la final facem un rezumat al alegerii.

SOAP este un protocol bazat pe XML care definește clar structura mesajului (*Envelope*, *Header*, *Body*) și, în mod uzual, descrie contractele cu **WSDL**, un document care stabilește operațiile, tipurile, mesajele și *endpoint*-urile. **XML-RPC** este o variantă mai simplă, tot pe XML, fără întreaga suită **WS-***, mesajele sunt `methodCall()` și `methodResponse()` cu tipuri de bază.

Un apel SOAP minimal pentru o sumă (trunchiat pentru claritate) și **răspunsul SOAP** asociat:

```
<!-- Request SOAP -->
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns:add xmlns:ns="http://exemplu/Calculator">
      <a>7</a>
      <b>5</b>
    </ns:add>
  </soap:Body>
</soap:Envelope>

<!-- Response SOAP -->
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns:addResponse xmlns:ns="http://exemplu/Calculator">
      <result>12</result>
    </ns:addResponse>
  </soap:Body>
</soap:Envelope>
```

Un apel XML-RPC echivalent:

```
<!-- Request XML-RPC -->
<methodCall>
  <methodName>add</methodName>
  <params>
    <param><value><int>7</int></value></param>
    <param><value><int>5</int></value></param>
  </params>
</methodCall>
```

SOAP aduce contracte stricte și extensii standardizate pentru securitate, tranzacții și fiabilitate, utile în medii *enterprise* care cer politici declarative și interoperabilitate controlată [39]. XML-RPC este ușor de implementat și citit, potrivit pentru prototipuri simple sau integrarea cu sisteme vechi. XML este verbos, ceea ce înseamnă încărcătură utilă (*payload*) mai mare și *parsing* mai lent. SOAP poate deveni complex din cauza specificațiilor WS-*, iar versionarea contractelor se face atent pentru a păstra compatibilitatea. În multe proiecte moderne, dacă nu ai nevoie de WS-Security sau de tranzacții la nivel de mesaj, soluțiile bazate pe JSON sunt mai ușoare de operat.

JSON-RPC este un protocol simplu care descrie apeluri prin obiecte JSON cu câmpuri standard, `jsonrpc` (versiunea protocolului), `method`, `params`, `id` pentru corelare. Transportul este de obicei HTTP sau *WebSocket*, iar răspunsurile au fie rezultat, fie eroare [47]. Este foarte ușor de implementat în orice limbaj, JSON este compact și rapid de procesat, iar integrarea cu *frontend*-uri este naturală. Se pot trimite și notificări fără *id* (nu se așteaptă răspuns) sau loturi de apeluri, ceea ce reduce latența pe conexiuni cu *overhead* fix.

Apelul `add` cu JSON-RPC, răspunsul asociat și un exemplu de eroare:

```
// Request JSON-RPC
{ "jsonrpc": "2.0", "method": "add", "params": { "a": 7, "b": 5 }, "id": 1 }

// Response JSON-RPC
{ "jsonrpc": "2.0", "result": 12, "id": 1 }

// Error JSON-RPC
{ "jsonrpc": "2.0", "error": { "code": -32602, "message": "Invalid params" },
"id": 1 }
```

Protocolul nu impune o schemă strictă pentru tipuri, deci disciplina contractelor revine echipei. În proiecte mai mari se recomandă documentarea exactă a metodelor și a formelor de date, plus reguli clare pentru erori și idempotentă. Dacă este nevoie de tipare binare compacte și de generarea de *stub*-uri dintr-un IDL, JSON-RPC nu oferă nativ această parte, în schimb *gRPC* sau *Thrift* o fac.

RPC și RMI explică mecanismul de bază al apelului sincron la distanță. Următorul pas istoric a fost *middleware*-ul clasic, care a încercat să generalizeze această idee pentru sisteme *enterprise* eterogene: obiecte distribuite, limbaje diferite, servicii de nume, tranzacții, securitate și interoperabilitate. CORBA este exemplul reprezentativ pentru această etapă.

2.5. Middleware clasic

Middleware-ul este **stratul de software care se așază între aplicații și infrastructură** (rețea, OS, drivere) și **oferă servicii comune**: comunicație, serializare, tranzacții, securitate, descoperire de servicii, astfel încât aplicațiile să nu mai implementeze de la zero aceste capabilități. Dacă **IPC local oferă doar „țevi”** pentru octeți, iar **RPC definește un contract de apel**, ***middleware*-ul adaugă reguli, protocoale și instrumente pentru interoperabilitate** între **procese**, **mașini** și chiar **limbaje** diferite. Beneficiul este uniformizarea modului de comunicare și observabilitate pe proiecte mari. Riscurile apar când stratul devine prea „deștept” și ascunde costuri sau când se centralizează prea multă logică într-un singur loc.

Middleware-ul clasic a apărut din nevoia de a lega aplicații scrise în limbaje diferite, rulate pe platforme diferite și distribuite pe mai multe mașini. Scopul nu era doar transmiterea unor octeți prin rețea, ci oferirea unui strat comun pentru contracte, localizare, serializare, apeluri la distanță, securitate și uneori tranzacții. În această zonă, CORBA este exemplul istoric cel mai reprezentativ.

2.5.1. CORBA (*Common Object Request Broker Architecture*)

Common Object Request Broker Architecture (**CORBA**) a fost un **standard OMG** (*Object Management Group*) care **a promis obiecte distribuite între limbaje și platforme diferite** [67]. Ideea de bază a fost să descrii contractul o singură dată și să lași unelte să genereze stubs pe client și skeletons pe server pentru C, C++, Java, Python și altele, astfel încât apelurile să arate „ca locale", deși traversează rețeaua.

Figurile 2.13 și 2.14 prezintă aceeași idee din două perspective complementare. Figura 2.13 arată **structura conceptuală a unui apel CORBA**: clientul rezolvă mai întâi numele obiectului în *Naming Service*, obține o referință la distanță, apoi apelează prin *stub*. ORB-ul clientului și ORB-ul serverului comunică prin IIOP, iar pe server POA (*Portable Object Adapter*) face legătura dintre cererea la distanță și obiectul concret, numit servant. Figura 2.14 detaliază **succesiunea interacțiunilor** dintre participanți, de la rezolvarea referinței până la întoarcerea rezultatului.

Contractul CORBA se scrie în *Interface Definition Language* (*IDL*), limbajul de definire a interfețelor într-o formă neutră față de limbajele de programare. Din IDL se generează automat cod de conectare pentru fiecare limbaj, clienții și serverele vorbesc aceeași semnătură, aceleași tipuri și aceleași erori, ceea ce face posibilă interoperabilitatea multi-limbaj.

Brokerul de obiecte, ORB (*Object Request Broker*) este componenta care primește cererea, face serializare și deserializare, găsește obiectul țintă și îi transmite apelul, apoi returnează rezultatul. Pentru programator, ORB este „motorul” invizibil care leagă *stub*-ul de pe client de *skeleton*-ul de pe server.

IIOP (*Internet Inter-ORB Protocol*) este **formatul standardizat al mesajelor CORBA** pe rețea. El garantează că un ORB de la un producător poate vorbi cu ORB-ul altuia, atât timp cât ambele respectă IIOP și același IDL.

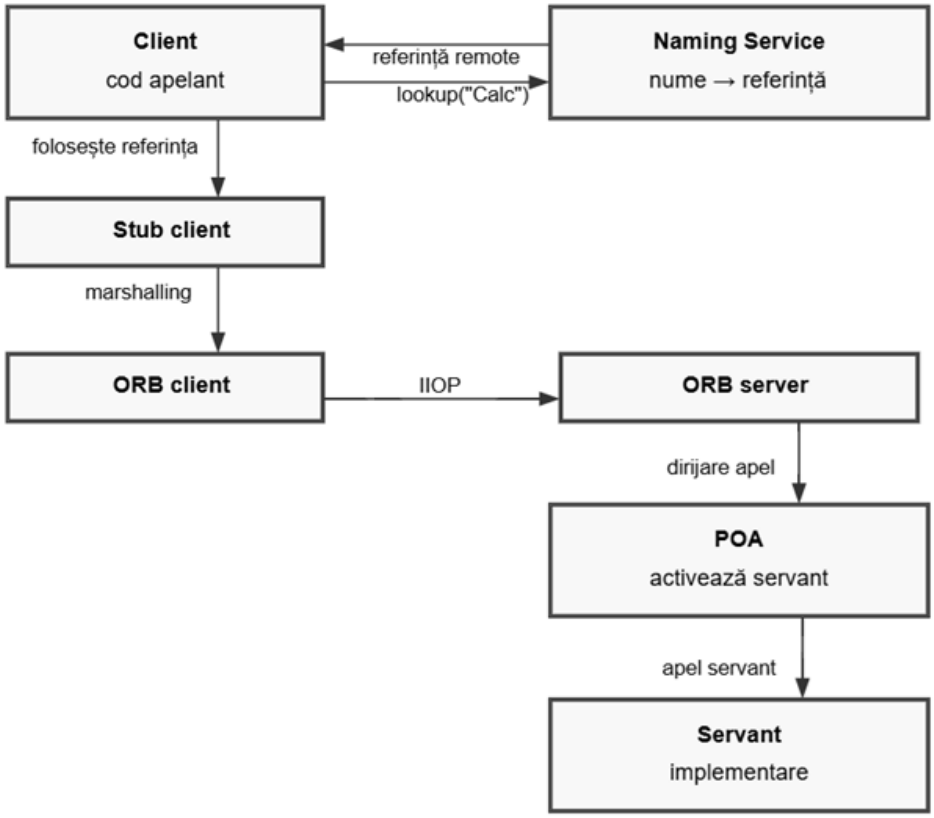


Figura 2.13. Structura unui apel CORBA (Naming Service, ORB, POA și servant)

Cele două figuri arată că, în CORBA, apelul la distanță este ascuns în spatele unui contract comun și al unei infrastructuri standardizate. ***Naming Service* ajută la găsirea obiectului distribuit**, ORB-ul intermediază comunicarea, iar POA leagă cererea de implementarea concretă. Astfel, clientul poate invoca un obiect la distanță aproape ca pe unul local, chiar dacă în spate există pași suplimentari de localizare, serializare și transport.

În CORBA ***Naming Service*** este un **registru unde un server publică un obiect sub un nume convenit**. Clientul face un ***lookup* (căutare) după acest nume**, primește înapoi o referință la distanță, adică o descriere completă a țintei care include ce trebuie pentru apeluri ulterioare, de exemplu adresa gazdei, portul, identificatorul obiectului, tipul interfeței. Pasul în care numele este rezolvat în această referință se numește **legare (*binding*)**, din acel moment clientul are un *stub* prin care poate invoca metode ca și cum ar fi locale. Dacă serverul se mută sau se schimbă versiunea obiectului, serverul repornește și face *rebind* în registru pe același nume, iar clienții pot relua căutarea pentru a obține noua referință.

POA (*Portable Object Adapter*) este componenta CORBA care face legătura între obiectele din aplicație, numite *servants*, și ORB, *runtime*-ul care primește și trimite apelurile la distanță. POA are câteva roluri cheie: creează referințe la distanță pentru *servants*, mapează cererile primite pe metoda corectă, activează sau dezactivează *servants*.

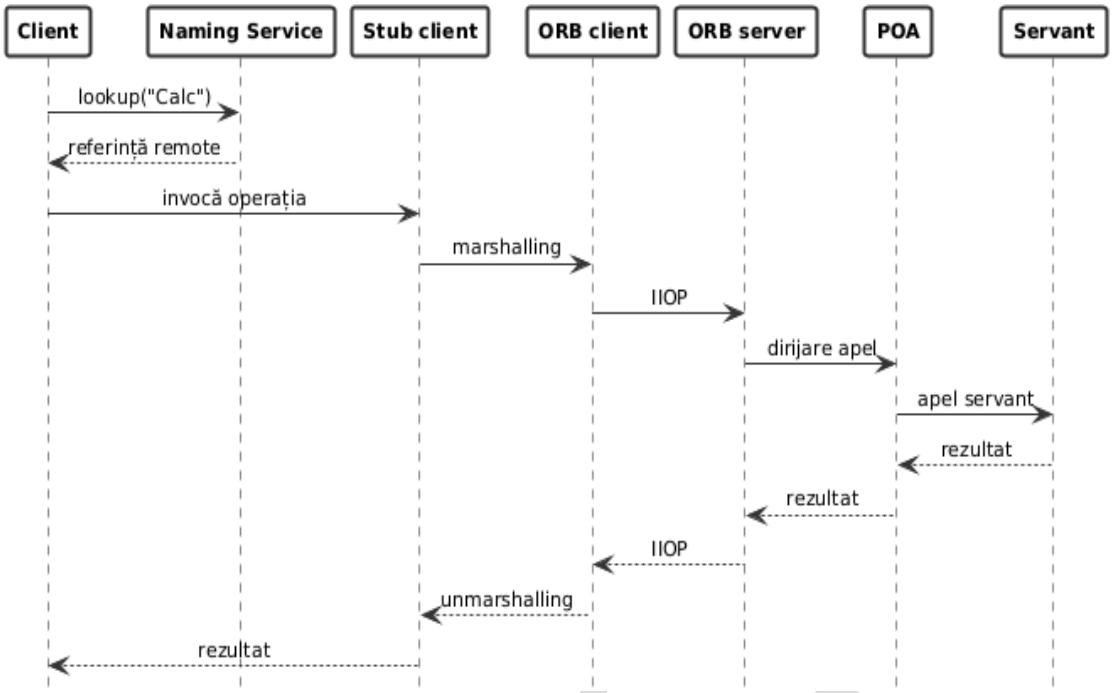


Figura 2.14. Interacțiunea într-un apel CORBA (rezolvare în *Naming Service*, apel la distanță și răspuns)

Important este că **programatorul lucrează cu un contract și** cu **o referință la distanță**, nu cu adrese și mesaje brute. Aceasta a fost promisiunea CORBA: interoperabilitate între limbaje și platforme prin IDL, generare de cod și un broker de obiecte comun. În același timp, tocmai această infrastructură bogată a adus și complexitate: configurare, generare de cod, compatibilitate între ORB-uri, depanare distribuită și costuri operaționale.

2.5.2. Ilustrare cu Java și C#

Interoperabilitatea poate fi ilustrată printr-un **exemplu CORBA minimal**: un server scris în Java expune un serviciu de calcul, iar un client scris în C# îl apelează prin aceeași interfață. Scopul este să se vadă cum arată în practică piesele discutate la nivel de concepte: contractul comun (IDL), generarea de tipuri și cod de conectare pentru fiecare limbaj, publicarea în serviciul de nume (*naming*) și legarea (*binding*) în timpul execuției (*runtime*), respectiv rolul *stub*-ului pe client și al *skeleton*-ului pe server. Exemplele rămân mici și cu rol demonstrativ, accentul este pe fluxul cap-la-cap, nu pe detalii de configurare.

Pentru a urmări ușor exemplul, trebuie privite trei piese: contractul comun, serverul care implementează contractul și clientul care obține referința la distanță și o invocă.

Un contract IDL (comun Java și C#) minim pentru un calculator cu o singură operație de adunare arată astfel:

```
// calc.idl
module demo {
  interface Calc {
    long add(in long a, in long b);
  };
};
```

Acest IDL definește un modul (`namespace`) `demo` **și o interfață** `Calc` cu metoda `add()`. Tipurile sunt cele standard CORBA (`long` aici este întreg pe 32 de biți). Din acest fișier se generează cod specific fiecărui limbaj:

- în Java, cu `idlj -fall calc.idl` rezultă interfețe, clase utilitare (*helper*) și scheletul (*skeleton*) necesar;
- în C#, un compilator/*binding* CORBA pentru .NET (de exemplu IIOP.NET) generează tipurile echivalente și *stub*-urile client.

Avantajul este dublu: contractul rămâne stabil și verificabil independent de implementări, iar compatibilitatea de tipuri este asigurată de generator. Dacă interfața evoluează, schimbarea se face întâi în IDL, apoi se regenerează codul în ambele lumi, menținând sincronizarea între client și server.

Mai jos este **varianta minimală de server CORBA în Java**. Se pornește un ORB, se activează POA, se creează „servantul" care implementează interfața generată din IDL, se obține o referință către obiectul la distanță și se publică în *Naming Service* sub un nume cunoscut, de exemplu `Calc`. Presupunem că a fost rulat generatorul pentru IDL și există clasele `demo.CalcPOA` și tipurile auxiliare în *classpath*.

```
// CalcImpl.java
import demo.*;                  // generat din IDL
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.CosNaming.*;

// clasa de baza este generata de idlj (CalcPOA), noi implementam logica
public class CalcImpl extends demo.CalcPOA {
    @Override
    public int add(int a, int b) {
        return a + b;      // logica simpla pe server
    }

    public static void main(String[] args) throws Exception {
        // initializare ORB si POA
        ORB orb = ORB.init(args, null);
        POA rootPoa =
            POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
        rootPoa.the_POAManager().activate();

        // creeaza servant si il transforma in obiect CORBA
        CalcImpl servant = new CalcImpl();
        org.omg.CORBA.Object ref = rootPoa.servant_to_reference(servant);
        demo.Calc href = demo.CalcHelper.narrow(ref);

        // leaga in Naming Service cu numele "Calc"
        org.omg.CORBA.Object ns = orb.resolve_initial_references("NameService");
        NamingContextExt nc = NamingContextExtHelper.narrow(ns);
        NameComponent[] name = nc.to_name("Calc");
        nc.rebind(name, href);

        System.out.println("Server CORBA pornit, serviciul 'Calc' inregistrat");
        orb.run(); // bucla principala, asteapta apeluri
    }
}
```

Serverul implementează interfața generată din IDL, pornește ORB și POA, publică obiectul sub numele `Calc` în *Naming Service*. Din acest moment, orice client care știe numele `Calc` îl poate rezolva și apela.

Codul unui client CORBA în C# (exemplu cu **IIOP.NET**):

```
// Client.cs
using System;
// urmatoarele namespaces sunt tipice pentru un binding CORBA .NET
using omg.org.CosNaming;              // Naming service
using Ch.Elca.Iiop;                   // ORB IIOP.NET
using Ch.Elca.Iiop.Idl;               // Utilitare IDL
using demo;                           // tipurile generate din calc.idl

class Client {
    static void Main(string[] args) {
        // initializare ORB
        IiopClient orb =
             (IiopClient)OrbServices.GetSingleton().CreateClientOrb();
        // obtine referinta la NameService
        NamingContext nameSvc =
             (NamingContext)orb.ResolveInitialReferences("NameService");
        // rezolva numele "Calc" si il converteste (narrow) la demo.Calc
        NameComponent[] path =
              new NameComponent[] { new NameComponent("Calc", "") };
        var obj = nameSvc.resolve(path);
        Calc calc = (Calc)orb.Narrow(obj, typeof(Calc));
        // apel remote, pare local in codul C#
        int r = calc.add(7, 5);
        Console.WriteLine("7 + 5 = {0}", r);
    }
}
```

Clientul pornește ORB-ul, întreabă *Naming Service* după numele `Calc`, primește o referință la distanță și o convertește la tipul `Calc` generat din IDL, apoi apelează `add` exact ca pe o metodă locală. *Stub*-ul serializează parametrii, trimite mesajul IIOP, așteaptă răspunsul și îl deserializează.

Exemplul arată, pe scurt: **același contract IDL este implementat pe Java, iar clientul scris în C# îl invocă prin IIOP**. Conceptual, *stub*-ul de pe client și *skeleton*-ul de pe server fac conectarea la rețea, astfel încât logica rămâne simplă, ca la un apel local, dar cu contracte și erori potrivite pentru rețea. Contractul comun este IDL, ambele lumi, Java și C#, generează cod din aceeași specificație (aceasta este cheia interoperabilității).

CORBA ilustrează foarte bine etapa *middleware*-ului de obiecte distribuite: apeluri sincrone, contracte formale, cod generat și infrastructură de localizare. Modelul este puternic, dar presupune ca apelantul și furnizorul să fie disponibili în același timp, iar apelul să aștepte răspunsul. În multe sisteme moderne, această sincronizare strictă devine o limită. De aceea, următoarea etapă importantă este *middleware*-ul orientat pe mesaje, unde aplicațiile comunică asincron, prin cozi, topicuri și brokeri.

2.6. Mesagerie și *middleware* orientat pe mesaje (asincron)

După mecanismele sincrone de tip RPC/RMI și *middleware*-urile de obiecte (CORBA) în care apelantul și furnizorul „stau față în față” și se așteaptă unul pe altul, urmează **abordarea asincronă**, în care **părțile cooperează fără sincronizare strictă în timp**. Când ritmul sau disponibilitatea diferă (trafic în vârf, operații lente, intermitență), este mai sănătos să nu blochezi apelantul, ci să trimiți mesaje care vor fi procesate când există resurse.

Mesajele permit **colaborarea fără a bloca părțile între ele**: un producător trimite un pachet de date cu antet și corp, consumatorul îl preia **când poate**, nu neapărat în același moment. Acesta este rolul *Message-Oriented Middleware* (**MOM**), un strat care primește, stochează, rutează și livrează mesaje după reguli clare [21]. Câștigul este decuplarea în timp, locație și ritm, serviciile nu mai stau față în față ca la RPC, ci comunică prin mesaje, ceea ce simplifică scalarea și limitează efectul erorilor locale. Contractul rămâne esențial (schema mesajului, tip, versiune, coduri de eroare), astfel încât testarea și depanarea să se facă la interfață, nu în detaliile interne.

O punte istorică între lumea componentelor *enterprise* și *middleware*-ul orientat pe mesaje este reprezentată de ***Message-Driven Beans*** **din EJB**. Un *Message-Driven Bean* este o componentă care nu este apelată direct printr-o metodă de către client, ci reacționează la mesaje livrate de container [28]. În loc ca un client să invoce sincron o operație, mesajul este pus într-o coadă sau publicat pe un topic, iar componenta îl procesează atunci când mesajul este livrat.

Această idee face vizibilă diferența dintre apelul sincron și colaborarea asincronă. În apelul sincron, apelantul așteaptă răspunsul. În mesagerie, producătorul trimite mesajul și își poate continua execuția, iar consumatorul procesează mesajul mai târziu. De aici apar avantajele principale ale mesageriei: decuplare în timp, reziliență mai bună la indisponibilitatea temporară a consumatorului și posibilitatea de a regla ritmul prin cozi.

2.6.1. Cozi față de topicuri, broker față de *brokerless*

În comunicarea prin mesaje există pe de o parte diferite **modele de distribuție**, adică felul în care un mesaj ajunge la consumatori, fie punct la punct prin cozi, fie către mai mulți abonați prin topicuri, și pe de altă parte diferite **moduri de transport**, adică felul în care mesajele circulă, fie printr-un broker care le primește, le păstrează și le livrează, fie direct între procese fără broker, caz în care aplicația controlează livrarea și relivrările. Figura 2.15 compară cele două modele de distribuție, coadă și topic.

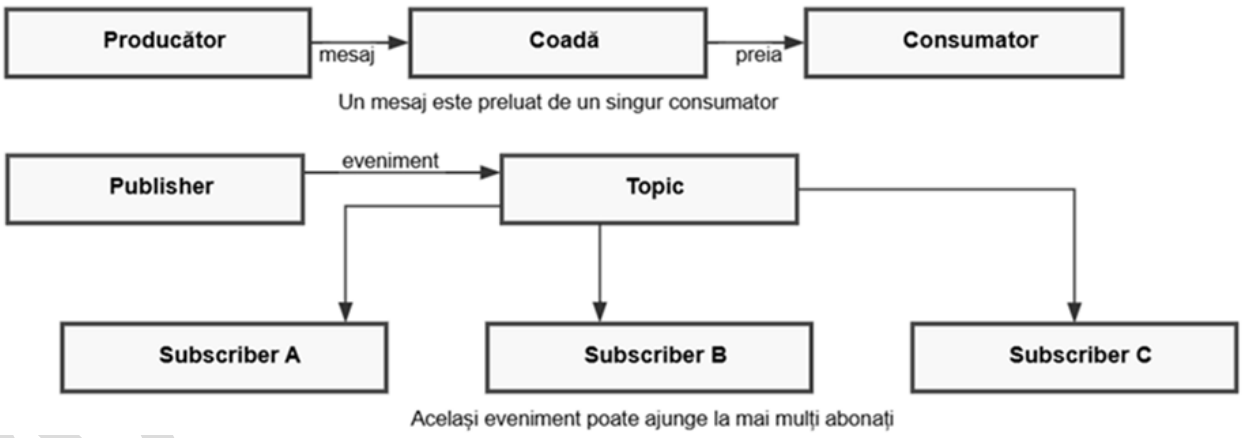


Figura 2.15. Coadă față de topic (*point-to-point* față de *publish-subscribe*)

Diferența principală este relația dintre mesaj și consumatori. Într-o coadă, mesajul este preluat de un singur consumator, chiar dacă există mai mulți consumatori disponibili. Într-un topic, același eveniment poate fi livrat către mai mulți abonați. Coada este potrivită pentru împărțirea muncii, de exemplu procesarea unor comenzi. Topicul este potrivit pentru notificarea mai multor părți interesate, de exemplu actualizarea stocului, trimiterea unei notificări și actualizarea unui raport.

Coada (*queue*) este un **canal punct la punct**, fiecare mesaj este consumat o singură dată de unul dintre consumatori. Modelul tipic este distribuirea de muncă, mai multe instanțe egale stau pe aceeași coadă și „își împart" mesajele, de exemplu procesarea de facturi sau generarea de rapoarte, când una cade, celelalte preiau sarcina. Confirmarea la citire (*ack*) și retransmiterea în caz de eșec sunt mecanismele care fac fluxul robust. **Topicul** (*topic*) este un canal unul la mulți, emitentul publică, fiecare abonat primește propria copie. Este potrivit pentru difuzare de evenimente, de exemplu „comanda a fost confirmată", oriunde există mulți ascultători independenți, actualizări UI, audit, notificări. Persistența și fereastra de reținere pot fi configurate, de la livrare imediată până la păstrare pe o perioadă scurtă pentru abonați care se reconectează.

Brokerul este un **server intermediar** care gestionează cozi și topicuri, păstrează mesaje pe disc, aplică politici de securitate și de rutare și expune metrici. Exemple uzuale sunt *RabbitMQ* pentru AMQP și brokerii MQTT. Avantajul este operarea simplificată, un loc clar unde se observă și se administrează fluxurile. Prețul este un punct suplimentar care trebuie configurat și monitorizat corect.

Soluțiile *brokerless* (fără broker) conectează direct procesele între ele, topologia și politicile sunt în mare parte în aplicație. *ZeroMQ*, de exemplu, oferă tipare de socketuri (*request-reply, publish-subscribe, push-pull*) care pot rula în același proces, între procese sau peste rețea, fără un server central. Câștigul este latență mică și simplitate operațională, însă persistența, retransmiterea și securitatea devin responsabilitatea aplicației. Există și soluții „ușoare cu server", precum NATS, care păstrează simplitatea interfeței, dar folosesc un server fie și minimal pentru rutare și *clustering*. Figura 2.16 compară aceste două variante, cu broker și fără broker.

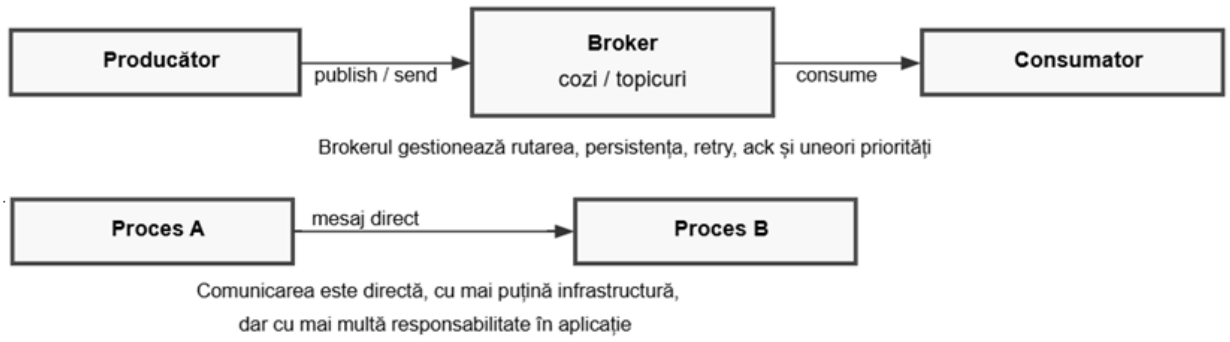


Figura 2.16. Broker față de *brokerless* (intermediar de mesaje față de comunicare directă)

Un broker simplifică aplicațiile deoarece preia responsabilități precum rutarea mesajelor, persistența, confirmările, retransmisiile și uneori prioritizarea. Costul este apariția unei componente centrale care trebuie operată, monitorizată și scalată. În modelele *brokerless*, procesele comunică direct, ceea ce poate reduce latența și infrastructura, dar mută mai multă responsabilitate în aplicație: descoperirea serviciilor, retransmisia, *buffering* și tratarea erorilor.

2.6.2. JMS, AMQP cu RabbitMQ, MQTT, NATS, ZeroMQ

JMS (*Java Message Service*) este o **interfață standard în Java pentru mesagerie**, adică un contract de **API care abstractizează cozi și topicuri**. Codul folosește JMS, iar furnizorul concret poate fi schimbat la configurare, util când se dorește portabilitate între produse. JMS definește noțiuni precum destinații, mesaje text sau binare, sesiuni și *acknowledge*, astfel încât logica aplicației să rămână decuplată de brokerul ales [68].

AMQP (*Advanced Message Queuing Protocol*) este un **protocol deschis pentru mesagerie**, *RabbitMQ* este cel mai întâlnit broker care îl implementează. Modelul este „*exchange, binding, queue*". Emitentul publică într-un rutator de mesaje (*exchange*), legăturile de rutare (*bindings*) trimit mesajele către una sau mai multe cozi (*queues*), consumatorii citesc din cozi. Se pot configura confirmări, rute alternative, politici de retransmitere. Este potrivit pentru fluxuri de lucru clasice, distribuire de sarcini, integrare între servicii interne [68].

MQTT este un **protocol pub sub foarte ușor pentru dispozitive și rețele instabile**, clienții se abonează la „topicuri" și primesc mesaje scurte. Oferă niveluri de calitate a serviciului, QoS 0, livrare „pe cât posibil", QoS 1, cel puțin o dată, QoS 2, exact o dată, astfel se poate alege între cost și siguranța livrării. Este popular în IoT, senzori, telemetrie, unde conexiunile sunt intermitente și consumul de resurse trebuie minimizat [68].

NATS este un sistem de mesagerie ușor, orientat spre publicare-abonare, cerere-răspuns și cozi de lucru simple. Pune accent pe latență redusă, simplitate operațională și rutare prin servere NATS, fără complexitatea unui broker AMQP complet.

ZeroMQ este o **bibliotecă de socluri cu tipare și fără broker**, oferă primitive pentru legături între fire, procese sau mașini cu *overhead* foarte mic. Tiparele predefinite, dialogul cerere-răspuns*, publish-subscribe, push-pull*, ajută la compunerea de conducte de procesare (*pipelines*). *ZeroMQ* nu păstrează mesaje pe disc și nu „ține minte" clienți, aplicația decide cum face persistență, retransmitere și securitate, în schimb câștigă simplitate și performanță [68].

În practică, **alegerea se face după context**, dacă ai nevoie de durabilitate, rute declarative și administrare centralizată, un broker AMQP sau MQTT este un început solid, dacă ai nevoie de latență foarte mică și topologii simple, *ZeroMQ* sau NATS pot reduce semnificativ complexitatea. Indiferent de instrument, disciplina contractelor de mesaj și a măsurătorilor (latență, debit, dimensiuni cozi, rate de eroare) rămâne cheia pentru testare și operare previzibile.

Aceste tehnologii nu sunt interschimbabile doar prin schimbarea unei biblioteci. Ele exprimă **compromisuri** diferite: JMS standardizează o interfață în ecosistemul Java, AMQP/*RabbitMQ* pune accent pe broker și rutare, MQTT este foarte potrivit pentru IoT și comunicații ușoare, NATS privilegiază simplitatea și latența redusă, iar *ZeroMQ* oferă primitive de mesagerie *brokerless* la nivel de bibliotecă.

2.6.3. Ilustrări: coadă cu JMS și topic cu MQTT

Urmează o **ilustrare minimală cu JMS 2.0** pe scenariul „sumă" folosind un schimb cerere-răspuns simplu. Un **client** trimite pe o **coadă** un mesaj cu doi întregi, un **worker** consumă mesajul, calculează suma și răspunde pe coada de răspuns indicată în mesaj. Folosim `MapMessage` pentru claritate și `JMSReplyTo` plus `JMSCorrelationID` pentru a lega cererea de răspuns. Codul rămâne neutru față de furnizorul JMS (ActiveMQ, Artemis etc.), doar presupune că există un `ConnectionFactory` și o `Queue`.

Înainte de cod, Figura 2.17 amintește elementele generale care trebuie urmărite în orice flux de mesagerie: producătorul, brokerul, consumatorul, confirmarea procesării și tratarea erorilor. Figura arată **elementele esențiale ale unui schimb de mesaje**. Producătorul trimite mesajul, brokerul îl păstrează sau îl rutează, consumatorul îl procesează, iar confirmarea (*ack*) arată dacă mesajul poate fi considerat livrat și tratat corect. Dacă procesarea eșuează, brokerul sau aplicația trebuie să decidă dacă mesajul se retransmite, se mută într-o coadă de erori sau se abandonează. În exemplul JMS care urmează, aceste idei apar într-o formă simplificată, de tip *request-reply*.

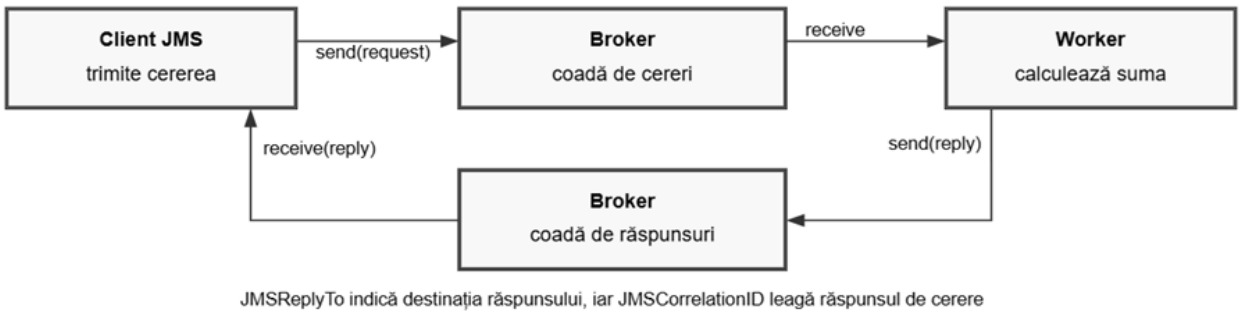


Figura 2.17. Flux de mesagerie (producător, broker, consumatori, confirmare și reîncercare)

Codul pentru **Worker (consumator)** calculează și răspunde:

```
// CalcWorker.java
import jakarta.jms.*;

// presupunem ca exista un ConnectionFactory cf si o coada Queue reqQueue
// (in practica, se obtin din JNDI sau din codul de bootstrap al brokerului)
public class CalcWorker {
    public static void main(String[] args) throws Exception {
        ConnectionFactory cf = /* obtain CF */;
        Queue reqQueue = /* obtain queue "calc.req" */;

        try (JMSContext ctx = cf.createContext(JMSContext.AUTO_ACKNOWLEDGE)) {
            JMSConsumer consumer = ctx.createConsumer(reqQueue);
            System.out.println("Worker ready...");

            while (true) {
                Message msg = consumer.receive();          // asteapta cererea
                if (!(msg instanceof MapMessage)) continue; // filtrare simpla

                MapMessage m = (MapMessage) msg;
                int a = m.getInt("a");
                int b = m.getInt("b");
                int sum = a + b;

                Destination replyTo = msg.getJMSReplyTo();  // unde raspund
                if (replyTo != null) {
                    MapMessage reply = ctx.createMapMessage();
                    reply.setInt("result", sum);
                    // leaga raspunsul de cerere
                    reply.setJMSCorrelationID(msg.getJMSCorrelationID());
                    ctx.createProducer().send(replyTo, reply);
                }
                System.out.println("Processed " + a + " + " + b + " = " + sum);
            }
        }
    }
}
```

***Worker*-ul ascultă pe coada de cereri, citește** cei doi operanzi din `MapMessage`, **calculează** suma și **trimite** un `MapMessage` de **răspuns către destinația** din `JMSReplyTo`, copiind `JMSCorrelationID` ca să poată fi corelat de client cu cererea inițială. În proiectele reale, `ConnectionFactory` și `Queue` se obțin din JNDI sau se configurează prin codul specific brokerului, iar securitatea, depășirea timpului de așteptare și politicile de retransmitere se setează explicit.

Codul pentru **Client (producător)** trimite cererea și așteaptă răspunsul:

```
// CalcClient.java
import jakarta.jms.*;
import java.util.UUID;

public class CalcClient {
    public static void main(String[] args) throws Exception {
        ConnectionFactory cf = /* obtain CF */;
        Queue reqQueue = /* obtain queue "calc.req" */;

        try (JMSContext ctx = cf.createContext(JMSContext.AUTO_ACKNOWLEDGE)) {
            // coada temporara pentru raspuns
            TemporaryQueue replyQueue = ctx.createTemporaryQueue();

            // mesaj de cerere
            MapMessage req = ctx.createMapMessage();
            req.setInt("a", 7);
            req.setInt("b", 5);

            // setari request-reply
            String corrId = UUID.randomUUID().toString();
            req.setJMSCorrelationID(corrId);
            req.setJMSReplyTo(replyQueue);

            // trimitere cerere
            ctx.createProducer().send(reqQueue, req);

            // asteptare raspuns cu acelasi correlation id
            String selector = "JMSCorrelationID = '" + corrId + "'";
            JMSConsumer replyConsumer =
                 ctx.createConsumer(replyQueue, selector);
            MapMessage reply = (MapMessage) replyConsumer.receive(5000);

            if (reply != null) {
                int result = reply.getInt("result");
                System.out.println("7 + 5 = " + result);
            } else {
                System.out.println("Nu a sosit raspuns in interval asteptare.");
            }
        }
    }
}
```

Clientul creează o coadă temporară pentru răspuns, **pune în cerere** `JMSReplyTo` și **un** `JMSCorrelationID` **unic**, apoi **trimite mesajul pe coada de cereri**. **Așteaptă pe coada temporară** un mesaj care are același `JMSCorrelationID`, **citește câmpul** `result` și **afișează suma.**

Exemplul JMS:

- arată diferența față de RPC: părțile sunt **decuplate în timp**, clientul nu blochează serverul, iar *worker*-ul poate procesa în ritmul lui.
- demonstrează **contractul de mesaj** minimal: chei `a`, `b` în cerere, `result` în răspuns, plus metadate standard JMS (`JMSReplyTo`, `JMSCorrelationID`) pentru corelare.
- e **ușor de scalat**: pui mai mulți *workers* pe aceeași coadă pentru distribuirea uniformă a cererilor.
- e **ușor de testat**: poți scrie teste care trimit mesaje pe coadă și verifică răspunsul pe coada temporară fără să depinzi de implementarea internă a *worker*-ului.

Refacem acum același „sumă” pe **MQTT**, păstrând ideea de *request–reply*, dar folosind topicuri. MQTT nu are cerere-răspuns nativ, însă îl obținem ușor: clientul publică pe `calc/req` un mesaj cu `a`, `b`, un `replyTo` (topicul pe care vrea răspunsul) și un `corrId` (pentru corelare). *Worker*-ul este abonat la `calc/req`, calculează și publică răspunsul pe `replyTo`, copiind `corrId`. Mai jos se folosesc **Eclipse Paho MQTT (Java, v3)**, broker la `tcp://localhost:1883`, încărcătură utilă (*payload*) JSON.

Codul pentru **Worker (abonat la cereri, publică răspunsuri):**

```
// CalcWorkerMqtt.java
import org.eclipse.paho.client.mqttv3.*;
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;

public class CalcWorkerMqtt {

    public static void main(String[] args) throws Exception {
        String broker = "tcp://localhost:1883";
        String clientId = "worker-1";
        String reqTopic = "calc/req";

        MqttClient client =
            new MqttClient(broker, clientId, new MemoryPersistence());
        MqttConnectOptions opts = new MqttConnectOptions();
        opts.setAutomaticReconnect(true);
        opts.setCleanSession(true);
        client.connect(opts);

        ObjectMapper json = new ObjectMapper();

        client.subscribe(reqTopic, (topic, msg) -> {
            try {

                // parsare mesaj JSON: a, b, replyTo, corrId
                ObjectNode m = (ObjectNode) json.readTree(msg.getPayload());
                int a = m.get("a").asInt();
                int b = m.get("b").asInt();
                String replyTo = m.get("replyTo").asText();
                String corrId = m.get("corrId").asText();

                int sum = a + b;

                // construire raspuns
                ObjectNode reply = json.createObjectNode();
                reply.put("result", sum);
                reply.put("corrId", corrId);

                byte[] payload = json.writeValueAsBytes(reply);
                // QoS 1: livrare cel putin o data
                client.publish(replyTo, new MqttMessage(payload));
                System.out.println("Processed " + a + " + " + b + " = " + sum);
            } catch (Exception e) {
                System.out.println("Worker error: " + e.getMessage());
            }
        });

        System.out.println("Worker MQTT ready, listening on topic " + reqTopic);
    }
}
```

Acest program pornește un „*worker*” MQTT care **se conectează la broker**, **se abonează la `calc/req`** și **așteaptă cereri în format JSON**. La fiecare mesaj, extrage `a`, `b`, `replyTo` și `corrId`, calculează suma, apoi publică răspunsul pe topicul indicat în `replyTo`. În răspuns, include result și același `corrId`, astfel clientul poate corela ușor cererea cu răspunsul. Practic, *worker*-ul joacă rolul „serviciului” care procesează cereri, iar `replyTo` + `corrId` implementează în mod explicit modelul cerere-răspuns peste topicuri MQTT.

Codul pentru **Client (publică cererea, ascultă pe un topic de răspuns temporar):**

```
// CalcClientMqtt.java
import org.eclipse.paho.client.mqttv3.*;
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;
import java.util.UUID;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.TimeUnit;

public class CalcClientMqtt {
    public static void main(String[] args) throws Exception {
        String broker = "tcp://localhost:1883";
        String clientId = "client-" + UUID.randomUUID();
        String reqTopic = "calc/req";

        // topic dedicat de raspuns pentru acest client (temporar/efemer)
        String replyTopic = "calc/reply/" + clientId;

        MqttClient client =
             new MqttClient(broker, clientId, new MemoryPersistence());
        MqttConnectOptions opts = new MqttConnectOptions();
        opts.setAutomaticReconnect(true);
        opts.setCleanSession(true);
        client.connect(opts);

        ObjectMapper json = new ObjectMapper();
        ArrayBlockingQueue<String> inbox = new ArrayBlockingQueue<>(1);

        // abonare la topicul de raspuns
        client.subscribe(replyTopic, (topic, msg) -> {
            try {
                String s = new String(msg.getPayload());
                inbox.offer(s);
            } catch (Exception e) {
                System.out.println("Client rx error: " + e.getMessage());
            }
        });

        // compunere cerere
        String corrId = UUID.randomUUID().toString();
        ObjectNode req = json.createObjectNode();
        req.put("a", 7);
        req.put("b", 5);
        req.put("replyTo", replyTopic);
        req.put("corrId", corrId);

        byte[] payload = json.writeValueAsBytes(req);
        client.publish(reqTopic, new MqttMessage(payload));

        // asteptare raspuns un timp scurt
        String replyStr = inbox.poll(5, TimeUnit.SECONDS);
        if (replyStr == null) {
```

```
                System.out.println("Nu a sosit raspunsul in interval asteptare.");
                return;
            }
            // verificare corelarea si afisare rezultat
            ObjectNode r = (ObjectNode) json.readTree(replyStr);
            if (!corrId.equals(r.get("corrId").asText())) {
                System.out.println("Raspuns necunoscut (corrId diferit).");
                return;
            }
            int result = r.get("result").asInt();
            System.out.println("7 + 5 = " + result);
        }
    }
```

Acest program pornește „clientul" MQTT, **generează** un `replyTopic` **dedicat** (în funcție de `clientId`), **se abonează la el** și **publică pe `calc/req` o cerere JSON** cu a=7, b=5, `replyTo=<topicul dedicat>` și un `corrId` unic. Apoi așteaptă scurt timp un mesaj pe `replyTopic`, verifică dacă `corrId` din răspuns corespunde și afișează `result`. Astfel, din perspectiva clientului, cererea este trimisă asincron, răspunsul vine pe canalul indicat, iar corelarea se face simplu prin `corrId`, similar exemplului JMS (dar fără broker de tip JMS și fără API-uri de cerere-răspuns dedicate).

În **exemplul MQTT:**

- dialogul cerere-răspuns se construiește din **topicuri**: cereri pe `calc/req`, răspunsuri pe un topic specific indicat în `replyTo`.
- **corelarea** se face prin `corrId` copiat în răspuns, analog cu `JMSCorrelationID`.
- **decuplarea în timp** rămâne: worker-ul procesează în ritmul lui; clientul poate retransmite ușor publicând din nou.
- **scalarea** e simplă: mai mulți workeri abonați la `calc/req` împart mesajele (QoS și retenția se aleg după caz).

Exemplele cu JMS și MQTT arată două forme diferite ale aceleiași idei. În cazul cozii, mesajul este de obicei o sarcină care trebuie procesată de un singur consumator. În cazul topicului, mesajul este mai apropiat de un eveniment care poate interesa mai mulți abonați. Această distincție pregătește secțiunea următoare, unde integrarea web folosește HTTP, SOAP sau REST, și secțiunea despre arhitecturi orientate spre evenimente, unde evenimentele devin mecanismul principal de colaborare.

2.7. Integrarea web

După procese locale, IPC pe aceeași mașină, *middleware*, RPC și mesagerie, nivelul următor este **integrarea prin web, adică servicii accesibile prin HTTP, cu mesaje text sau binare ce traversează rețele eterogene** [46]. Avantajul practic este interoperabilitatea: orice platformă care poate folosi HTTP poate consuma aceste servicii, iar instrumentele de rețea, mecanismele de jurnalizare și instrumentele de monitorizare sunt deja mature. La acest nivel, contractul devine centrul preocupării: ce puncte de acces există, ce metode acceptă, ce formate de date folosesc, ce coduri de eroare întorc, cum se autentifică apelurile și ce politici de control al ratei se aplică.

Există **două mari familii istorice** de integrare web. Prima, **serviciile web** sau **WS-*** (SOAP, WSDL, *WS-Security* și extensiile înrudite), oferă un model puternic orientat pe contract formal, cu descrieri detaliate, politici și standarde pentru securitate și fiabilitate, util acolo unde

sunt necesare reguli stricte și guvernanță de tip *enterprise* [39]. A doua, **REST** peste HTTP cu JSON (sau alte limbaje similare), tratează protocolul HTTP ca pe un set de reguli deja utile, resurse, metode standard, coduri de stare, etichete de cache, fiind preferată pentru simplitate, vizibilitate și evoluție rapidă [45], [46], [47]. În ambele cazuri se urmărește aceeași țintă, contracte clare, compatibilitate în timp și o operare observabilă, diferența este în cât de multă formalizare se alege la nivelul contractului și al instrumentelor. Secțiunile următoare ilustrează concis ambele abordări, un exemplu Java pentru SOAP și un exemplu simplu pentru REST, plus bune practici de versionare și compatibilitate.

2.7.1. Servicii web WS-* (SOAP, WSDL, WS-Security): avantaje și limite

Modelul WS-* tratează integrarea ca pe un contract formal: mesajele sunt SOAP (documente XML într-un plic standard), descrierea interfeței este WSDL (o „schemă" a operațiilor, tipurilor și adreselor), iar politicile de securitate se exprimă prin *WS-Security* (semnare, criptare, tokenuri). Avantajul este predictibilitatea: din WSDL se pot genera automat clase client/server, erorile și tipurile sunt strict definite, iar politicile pot fi impuse uniform [39]. Limitele apar la caracterul verbos al XML, costul de analiză lexicală (*parsing*) și la curba de învățare, de aceea WS-* rămâne preferat în medii *enterprise* unde guvernanța și standardizarea sunt prioritare, în timp ce REST/JSON este mai comod în sisteme orientate pe viteză de livrare.

Serviciile web din familia WS-* au fost construite în jurul ideii de **contract formal și mesaje standardizate**. **WSDL descrie operațiile, tipurile și *endpoint*-ul**, **SOAP definește formatul mesajului**, iar **extensii precum *WS-Security* adaugă reguli pentru semnare, criptare și autentificare la nivel de mesaj**. Figura 2.18 arată relația dintre contractul WSDL comun și mesajele SOAP de cerere și răspuns.

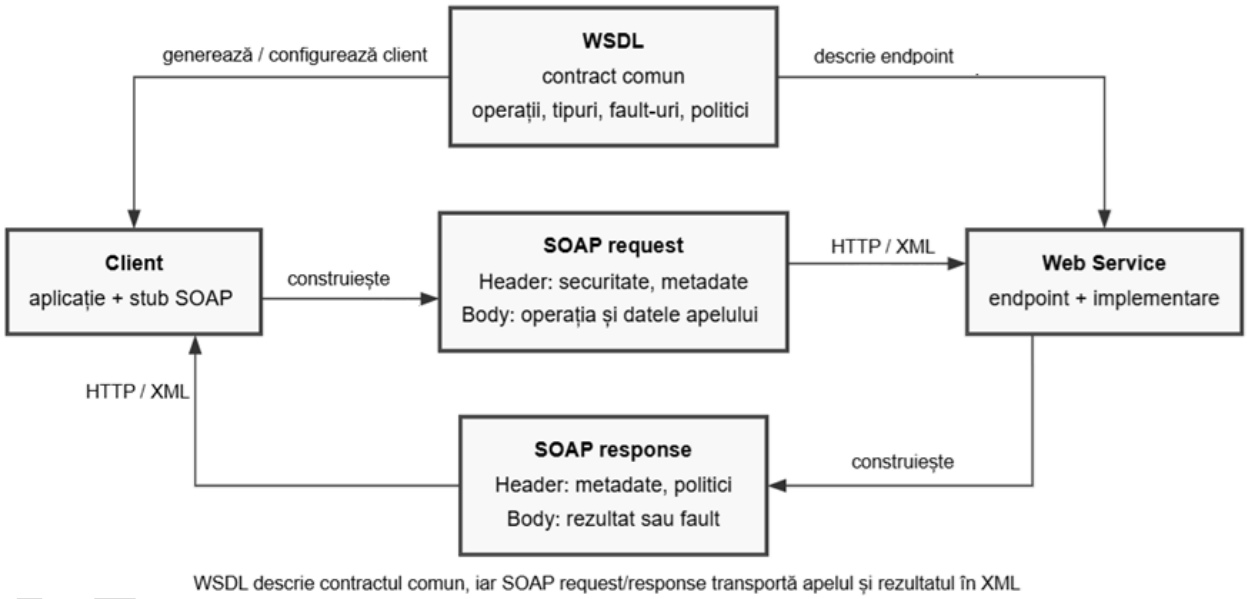


Figura 2.18. Servicii web WS-* (contract WSDL comun și mesaje SOAP *request/response*)

Figura arată **rolul contractului formal**. Clientul poate genera cod pornind de la WSDL, apoi transmite mesaje SOAP către un punct final (*endpoint*). Avantajul este precizia contractului și suportul pentru cerințe *enterprise*, de exemplu securitate la nivel de mesaj, tranzacții sau politici. Costul este complexitatea suplimentară a formatului XML, a standardelor și a configurației.

WS-* este potrivit mai ales acolo unde contractul formal, interoperabilitatea strictă și politicile *enterprise* sunt mai importante decât simplitatea apelului. În schimb, pentru multe aplicații web moderne, în special API-uri publice sau servicii interne mai simple, această familie poate părea greoaie. De aici a venit popularitatea REST, care folosește direct mecanismele HTTP: metode, URI-uri, coduri de stare, antete și reprezentări precum JSON.

Un exemplu minimal în Java (JAX-WS) arată cum arată contractul și cum se publică un serviciu simplu „`add`”:

```
// CalculatorWS.java
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
public interface CalculatorWS {
    @WebMethod
    int add(int a, int b);
}

// CalculatorWSImpl.java
import javax.jws.WebService;

@WebService(endpointInterface = "CalculatorWS")
public class CalculatorWSImpl implements CalculatorWS {
    @Override
    public int add(int a, int b) {
        // logica simpla a serviciului
        return a + b;
    }
}

// Publish.java
import javax.xml.ws.Endpoint;

public class Publish {
    public static void main(String[] args) {
        // publica serviciul la o adresa HTTP
        Endpoint.publish("http://localhost:8080/calc", new CalculatorWSImpl());
        System.out.println("SOAP service up at /calc?wsdl");
    }
}
```

Interfața anotată `@WebService` devine **contractul** din care *runtime*-ul generează WSDL (accesibil la `/calc?wsdl`). **Implementarea** separă logica de detaliile de transport, iar `Endpoint.publish` expune serviciul la HTTP fără server suplimentar pentru un demo. Clientul nu a fost inclus aici ca să rămână compact, dar fluxul tipic este: se ia WSDL-ul, se generează stubs cu unelte precum `wsimport`, apoi se invocă `stub.add(7,5)` exact ca un apel local; apoi are loc serializarea în SOAP, trimiterea și deserializarea.

WSDL descrie operațiile, tipurile de date și adresele; dacă se declară *fault*-uri, acestea devin erori verificate în stubs. Această formalizare permite verificări automate de compatibilitate și îi ajută pe clienți să se protejeze la schimbări.

WS-Security adaugă în antetul SOAP semnături digitale, criptare și *token*-uri (de exemplu SAML) astfel încât integritatea, confidențialitatea și autentificarea să fie parte a contractului. În practică se configurează la nivel de politică (*policy*) și se aplică uniform tuturor operațiilor, avantajos în organizații cu cerințe stricte.

2.7.2. Servicii web REST (HTTP, versionare și compatibilitate)

După exemplul minimal din 1.7.3, putem formula acum regulile generale ale stilului REST. **REST folosește direct protocoalele web**, fiecare **resursă** are un **URL stabil**, operațiile folosesc **verbele HTTP** (GET, POST, PUT, PATCH, DELETE), iar **datele circulă de obicei ca JSON** [45], [46], [47]. Ideea practică este să fie tratat serverul ca un set de resurse, nu ca o colecție de proceduri. Astfel, interfața devine predictibilă, ușor de testat și bine sprijinită de instrumentele web. Un răspuns REST bun include **coduri HTTP** corecte (200, 201, 400, 404, 409, 500), antete utile (*ETag, Cache-Control*), formate clare și reguli de compatibilitate explicite. HATEOAS (*Hypermedia as the Engine of Application State*) înseamnă că răspunsurile pot include linkuri către acțiunile următoare posibile. Astfel, clientul nu trebuie să cunoască dinainte toate rutele, ci poate descoperi pașii următori din răspunsurile primite.

În REST, **contractul nu este doar corpul mesajului**. El include combinația dintre adresa URI a resursei, metoda HTTP, antete, coduri de stare și formatul reprezentării trimise sau primite. Figura 2.19 ilustrează un apel simplu: clientul cere o resursă printr-un URI, folosește o metodă HTTP, iar serverul răspunde cu un cod de stare și o reprezentare, de exemplu JSON.

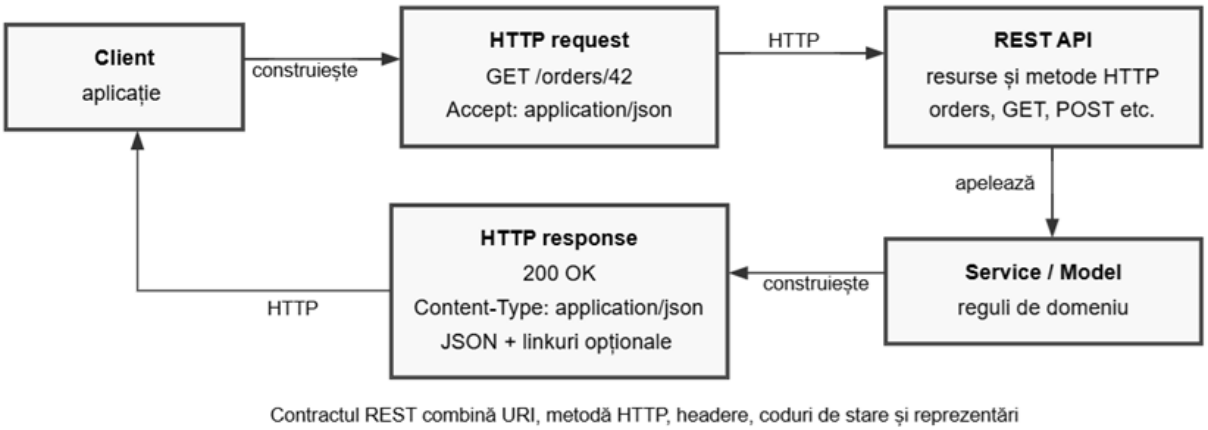


Figura 2.19. Serviciu REST (resurse HTTP, coduri de stare și reprezentări JSON)

Într-un API REST, metoda HTTP are semnificație. **GET** citește o resursă și ar trebui să fie fără efecte laterale asupra stării aplicației. **POST** creează sau declanșează o operație care nu se potrivește direct cu actualizarea unei resurse existente. **PUT** înlocuiește o resursă sau o creează la o adresă cunoscută. **PATCH** modifică parțial o resursă. **DELETE** elimină resursa. Folosirea consecventă a acestor metode face API-ul mai previzibil și mai ușor de testat.

Exemplu minimal Java, un punct final (*endpoint*) REST JSON pentru o sumă.

```
// build.gradle or pom.xml ar include Spring Web, aici aratam doar clasa
// SumController.java
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import java.net.URI;
import java.util.Map;

@RestController
@RequestMapping("/api")
public class SumController {
```

```
    @PostMapping("/sum")
    public ResponseEntity<Map<String, Object>>
                sum(@RequestBody Map<String, Integer> in) {
        // validari simple
        if (in == null || !in.containsKey("a") || !in.containsKey("b")) {
            return ResponseEntity.badRequest().body(Map.of(
                "error", "missing fields a or b"
            ));
        }
        int a = in.get("a");
        int b = in.get("b");
        int result = a + b;
        // raspuns REST cu incarcatura JSON si linkuri catre pasi urmatori
        Map<String, Object> body = Map.of(
            "result", result,
            "_links", Map.of(
                "self", "/api/sum",
                "example", "/api/examples/sum?a=7&b=5"
            )
        );
        // 200 OK pentru calcul, 201 Created daca am crea o resursa noua
        return ResponseEntity.ok(body);
    }
    @GetMapping("/examples/sum")
    public ResponseEntity<Map<String, Object>> example(@RequestParam int a,
@RequestParam int b) {
        // exemplu GET idempotent
        return ResponseEntity.ok(Map.of("result", a + b));
    }
}
```

Un controler expune două rute, un `POST /api/sum` care primește JSON cu câmpurile `a` și `b` și întoarce `result`, plus linkuri `_links` care arată următorii pași posibili (auto descriere prin HATEOAS, în formă simplă), și un GET pentru probă. Codurile HTTP reflectă starea, `400` pentru intrări greșite, `200` pentru succes. Într-o aplicație reală s-ar adăuga validări declarative, mesaje de eroare consistente și antete de cache acolo unde are sens.

Exemplu minimal React, apel către *endpoint* și randare.

```
// SumWidget.jsx
import { useState } from "react";

export default function SumWidget() {
  const [a, setA] = useState(0);
  const [b, setB] = useState(0);
  const [result, setResult] = useState(null);
  const [error, setError] = useState(null);

  async function handleCompute() {
    setError(null);
    setResult(null);
    try {
      const resp = await fetch("/api/sum", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ a: Number(a), b: Number(b) })
      });
      if (!resp.ok) {
        // tratez eroarea REST pe cod HTTP
        const e = await resp.json().catch(() => ({}));
```

```
        throw new Error(e.error || `HTTP ${resp.status}`);
      }
      const data = await resp.json();
      setResult(data.result);
    } catch (e) {
      setError(e.message);
    }
  }
  return (
    <div>
      <input value={a} onChange={e => setA(e.target.value)} />
      <input value={b} onChange={e => setB(e.target.value)} />
      <button onClick={handleCompute}>Compute</button>
      {result !== null && <div>Result: {result}</div>}
      {error && <div style={{ color: "red" }}>Error: {error}</div>}
    </div>
  );
}
```

O componentă trimite un POST cu JSON, verifică `resp.ok`, parsează răspunsul și afișează rezultatul sau eroarea. Interfața nu cunoaște detalii de server, vede doar rute, metode și formate. Această decuplare permite testarea UI fără server real, cu dubluri care răspund pe aceleași rute.

Pe scurt, **HATEOAS** înseamnă că în răspunsuri se pot include linkuri către acțiuni valide în starea curentă, de exemplu `_links.next`, `_links.cancel`. Clientul nu mai codifică rute, urmează linkurile din răspuns. În practică, multe API-uri folosesc o formă minimală, linkuri către resurse înrudite, ceea ce deja ajută la descoperire și la compatibilitate.

Versionare și compatibilitate. API-ul se menține compatibil în timp adăugând câmpuri noi ca opționale, fără a schimba semnificațiile celor existente. Când chiar este nevoie de o schimbare ruptoare, se introduce o nouă versiune (de exemplu prefix de rută, v2), apoi ambele versiuni conviețuiesc o perioadă. O alternativă este negocierea prin antete (de exemplu `Accept: application/vnd.example.sum+json;v=2`), utilă când nu se dorește multiplicarea rutelor. Indiferent de stil, regulile trebuie documentate, incluzând coduri de eroare stabile, mesaje clare și exemple reale. Testele se scriu la nivel de contract, nu împotriva implementării interne, astfel clienții rămân protejați la refactorizări.

Bune practici esențiale. Folosește verbele HTTP potrivite, tratează idempotent acolo unde se poate, păstrează mesaje de eroare consecvente, adaugă `ETag` și `If-Match` pentru actualizări sigure când există concurență, oferă paginație pentru colecții mari și folosește identificatori de corelație în antete pentru depanare. Astfel API-ul rămâne previzibil, extensibil și ușor de operat în timp.

WS-* și REST nu sunt doar două formate diferite, ci **două stiluri diferite de integrare web**. WS-* pune accent pe contract formal, extensii *enterprise* și mesaje standardizate independent de detaliile HTTP. REST pune accent pe resurse, simplitatea protocolului HTTP, coduri de stare și reprezentări ușor de consumat. În practică, alegerea depinde de context. Sisteme *enterprise* cu cerințe formale de securitate și interoperabilitate pot justifica WS-*. API-urile web moderne, microserviciile și aplicațiile mobile folosesc frecvent REST, mai ales cu JSON, pentru simplitate și compatibilitate largă.

Integrarea web este, în mod obișnuit, **centrată pe cerere și răspuns**. Clientul cere ceva, serverul răspunde. În multe sisteme moderne, însă, colaborarea nu pornește întotdeauna de la o cerere directă, ci de la apariția unui eveniment: o comandă plasată, o plată confirmată, un senzor actualizat sau un fișier încărcat. Această schimbare de perspectivă duce spre arhitecturile orientate spre evenimente.

2.8. Arhitecturi orientate spre evenimente (*event-driven*)

După ce am văzut integrarea sincronă (RPC, REST), următorul pas firesc este **colaborarea prin evenimente**. Un ***eveniment*** **descrie ceva ce s-a întâmplat deja în sistem** (de exemplu „Comandă plasată"), **are un nume stabil, o schemă pentru date** și **un identificator de corelație**. Un **producător** publică evenimentul într-un canal (topic), unul sau mai mulți **consumatori** îl procesează independent, la ritmul lor.

Câștigul este decuplarea în timp, în locație și în ritm, serviciile nu mai stau față în față, ci se coordonează prin mesaje. Modelul favorizează coregrafia (fiecare participant reacționează la evenimente), extensibilitatea (se pot adăuga consumatori noi fără a atinge sursa) și reziliența (căderile locale nu opresc întregul flux).

Cheia calității rămâne **contractul**: numele evenimentului, schema, versiunea și regulile de livrare (ordonare, retenție) [21], [22]. Figura 2.20 sintetizează această relație dintre producător, broker și consumatori.

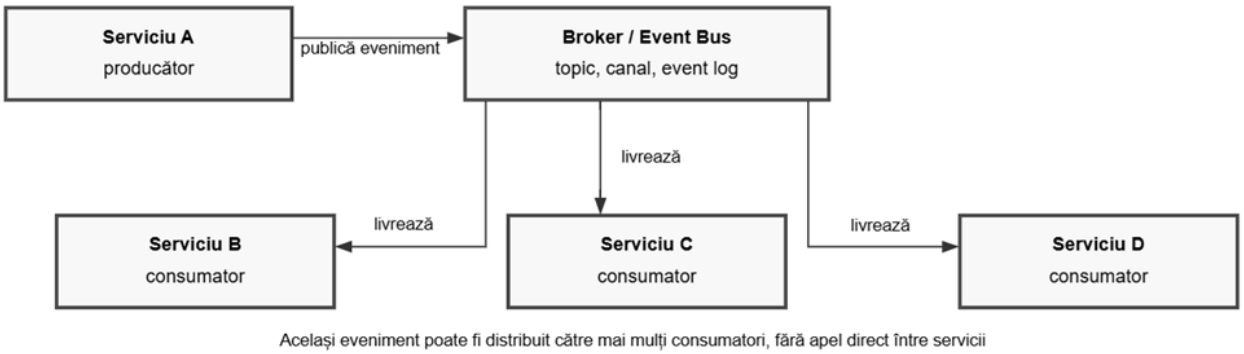


Figura 2.20. Arhitectură orientată spre evenimente (producător, broker și consumatori)

Figura arată **ideea de bază**: serviciul care produce evenimentul nu apelează direct toți consumatorii. El publică un mesaj care descrie ceva ce s-a întâmplat, iar consumatorii interesați reacționează independent. Astfel, producătorul nu trebuie să cunoască toate efectele ulterioare ale schimbării.

Într-o arhitectură orientată spre evenimente, **accentul nu cade pe comanda directă** „execută această operație", **ci pe anunțul „acest lucru s-a întâmplat"**. De exemplu, un serviciu de comenzi poate publica evenimentul `OrderPlaced`, iar alte servicii pot reacționa: unul rezervă stocul, altul inițiază plata, altul trimite notificarea, altul actualizează un raport. Producătorul evenimentului nu trebuie să știe câți consumatori există și ce fac aceștia.

Este utilă diferența dintre **comandă** și **eveniment**. O comandă exprimă o intenție: „rezervă stocul", „trimite factura", „creează contul". Ea este, de obicei, adresată unui destinatar care trebuie să decidă dacă o poate executa. Un eveniment exprimă un fapt deja produs: „comanda a fost plasată", „plata a fost confirmată", „contul a fost creat". Evenimentul nu cere direct cuiva să facă ceva, ci anunță o schimbare la care alte părți pot reacționa. În practică, sistemele folosesc adesea ambele forme: comenzi pentru solicitări explicite și evenimente pentru propagarea schimbărilor. Figura 2.21 clarifică această diferență de intenție.

Comanda are un **destinatar mai clar** și cere executarea unei acțiuni. **Evenimentul anunță o schimbare** și permite mai multor consumatori să reacționeze fără ca producătorul să îi cunoască direct.

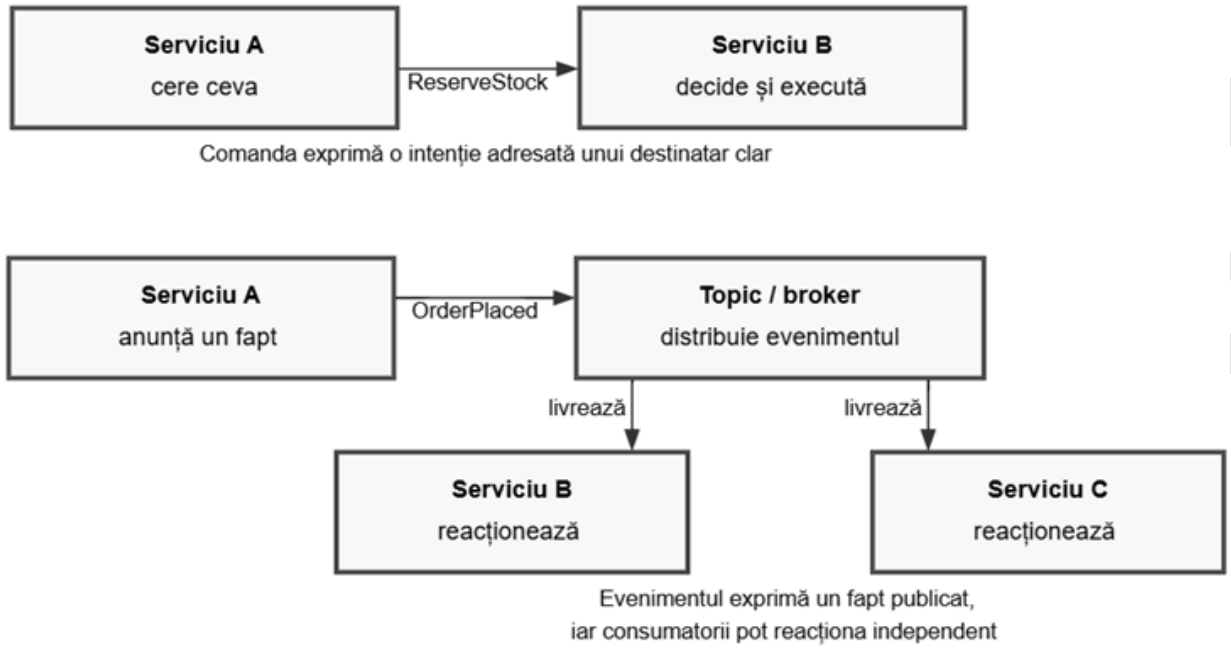


Figura 2.21. Comandă față de eveniment (intenție adresată față de fapt publicat)

Un efect important al arhitecturilor orientate spre evenimente este **consistența eventuală**. Dacă serviciul de comenzi publică `OrderPlaced`, serviciul de stoc, serviciul de plăți și serviciul de notificări nu se actualizează neapărat în aceeași milisecundă. Pentru o perioadă scurtă, sistemul poate fi într-o stare intermediară: comanda există, dar plata nu este încă confirmată sau notificarea nu a fost trimisă. În locul unei tranzacții globale, sistemul folosește evenimente, stări intermediare și, uneori, acțiuni compensatorii. Această abordare este mai scalabilă, dar cere proiectare atentă a stărilor și a erorilor.

Consumatorii de evenimente trebuie proiectați cu grijă, deoarece același eveniment poate fi livrat mai mult de o dată, mai ales când există retransmisie, confirmări sau recuperare după erori. De aceea, **procesarea ar trebui să fie idempotentă**: același eveniment procesat de două ori nu trebuie să producă efecte greșite. De exemplu, dacă evenimentul `PaymentConfirmed` ajunge de două ori, sistemul nu trebuie să emită două facturi sau să trimită de două ori aceeași plată mai departe. O soluție obișnuită este păstrarea unui identificator unic al evenimentului și verificarea faptului că acesta nu a mai fost procesat.

Un avantaj al evenimentelor este că pot forma un **jurnal al schimbărilor importante** din sistem. Evenimentele păstrate într-un jurnal de evenimente pot fi folosite pentru audit, reconstrucția unor proiecții, refacerea unor consumatori sau diagnosticarea unor incidente. Totuși, un jurnal de evenimente nu este automat o soluție completă de *event sourcing*. *Event sourcing* presupune ca starea curentă să fie derivată din succesiunea evenimentelor, ceea ce cere reguli stricte de versionare, compatibilitate și reconstrucție. În multe aplicații, evenimentele sunt folosite doar pentru integrare, nu ca sursă unică a adevărului [42].

Pentru ilustrare practică folosim un **flux minim cap-la-cap**: un serviciu publică evenimentul „`ComandaPlasata`” într-un topic JMS, folosind un contract simplu cu câmpuri clare, inclusiv un identificator de corelație, iar un alt serviciu se abonează la același topic și reacționează independent. Scopul este să se vadă decuplarea în timp și în ritm, faptul că publicatorul nu așteaptă consumatorul, precum și disciplina de contract, schemă, versiune și idempotentă care face posibilă testarea și depanarea fără a depinde de implementările interne.

Codul serviciului care publică evenimentul „ComandaPlasata” într-un topic JMS:

```
// OrderController.java (serviciul publicator)
import org.springframework.jms.core.JmsTemplate;
import org.springframework.web.bind.annotation.*;
import java.util.Map;
import java.util.UUID;

@RestController
@RequestMapping("/api/orders")
public class OrderController {
  private final JmsTemplate jms;
  public OrderController(JmsTemplate jms) { this.jms = jms; }
  @PostMapping
  public Map<String, Object> place(@RequestBody Map<String, Object> in) {
    // validari minimale
    String orderId = UUID.randomUUID().toString();
    String correlationId = UUID.randomUUID().toString();
    Map<String, Object> event = Map.of(
      "type", "ComandaPlasata",
      "orderId", orderId,
      "items", in.get("items"),
      "total", in.get("total"),
      "ts", System.currentTimeMillis(),
      "correlationId", correlationId,
      "schemaVersion", 1
    );
    // publicare pe un topic JMS
    jms.convertAndSend("topic.orders", event);
    return Map.of("status", "OK", "orderId", orderId, "correlationId",
correlationId);
  }
}
```

Un punct final (*endpoint*) REST primește o comandă, generează un `orderId` și un `correlationId`, compune evenimentul „ComandaPlasata” și îl publică pe un topic. API-ul HTTP rămâne simplu, iar integrarea cu restul ecosistemului se face prin evenimente, fără a aștepta sincron pașii următori (plată, stoc, notificare).

Codul serviciului care se abonează la topicul JMS și consumă evenimentul „ComandaPlasata”:

```
// InventoryListener.java (serviciul consumator)
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;
import java.util.Map;
@Component
public class InventoryListener {
  @JmsListener(destination = "topic.orders", containerFactory = "jmsFactory",
subscription = "inventory")
  public void onOrderPlaced(Map<String, Object> event) {
    // validare eveniment
    if (!"ComandaPlasata".equals(event.get("type"))) return;
    String orderId = (String) event.get("orderId");
    // aici s-ar rezerva stocul pentru items (exemplu simplificat)
    System.out.println("Rezerv stoc pentru comanda " + orderId);

    // in caz real: mesaj de diagnostic cu correlationId
  }
}
```

Un consumator se abonează la `topic.orders` și reacționează doar la evenimentele de tip „`ComandaPlasata`”. Logica de stoc poate rula în ritmul ei, poate face retransmitere, poate emite evenimentul „`StocRezervat`” fără a modifica serviciul care a publicat comanda. Astfel se obține coregrafie între servicii, fiecare participant implementează regulile sale locale.

Această abordare merită folosită atunci când pașii următori nu trebuie să blocheze fluxul inițial (confirmarea comenzii poate merge imediat, iar plata, stocul, notificarea pot continua în fundal), când există mulți ascultători independenți (audit, rapoarte, UI reactiv), când reziliența și scalarea separată sunt priorități.

Arhitectura orientată spre evenimente este potrivită atunci când mai multe părți trebuie să reacționeze la schimbări fără cuplare directă, când sistemul trebuie să absoarbă vârfuri de trafic sau când procesele pot continua asincron. Nu este însă o soluție universală. Pentru interogări simple și răspunsuri imediate, un API sincron poate fi mai clar. Pentru fluxuri critice, în care utilizatorul trebuie să primească imediat o decizie, evenimentele trebuie combinate cu validări sincrone sau cu stări explicite de tip „în procesare”. Alegerea depinde de natura interacțiunii, de toleranța la întârziere și de costul complexității operaționale.

Evenimentele reduc cuplarea directă dintre servicii, dar **nu elimină nevoia de guvernanță**. Contractele evenimentelor trebuie documentate, versiunile trebuie gestionate, iar politicile de securitate, audit și compatibilitate trebuie stabilite explicit. Această nevoie de coordonare la nivel de sistem duce spre discuția despre SOA, integrare și guvernanță.

2.9. SOA, integrare și guvernanță

După ce au fost prezentate RPC, mesageria și integrarea web, pasul firesc este **un nivel de integrare care le organizează pe toate sub un set comun de reguli**. Aici se poziționează **SOA**, ca stil arhitectural care **dă coerență la scară, contracte clare, politici comune, mecanisme de compunere a pașilor și instrumente de operare** [38]. Figura 2.22 sintetizează această relație dintre servicii, contracte, guvernanță și mecanisme de integrare.

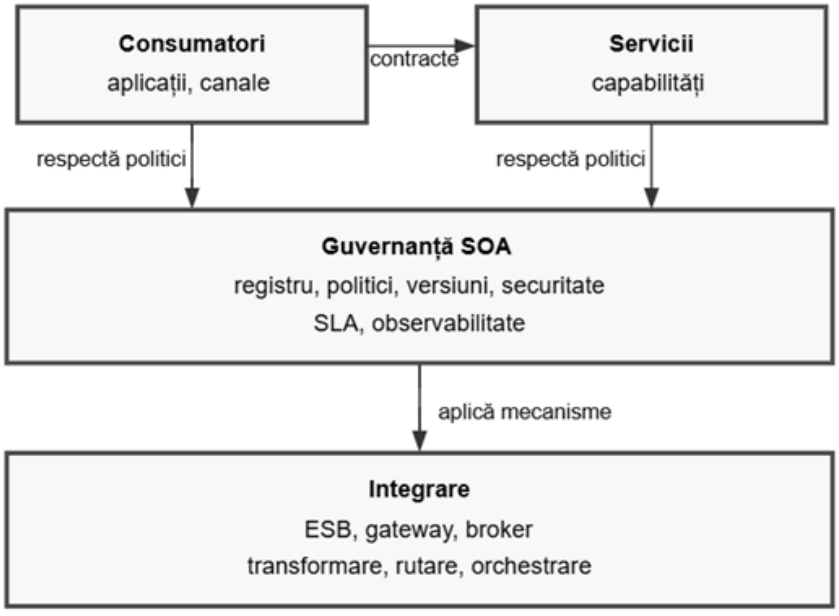


Figura 2.22. SOA, integrare și guvernanță (contracte, politici și mecanisme de integrare)

Integrarea spune **cum colaborează serviciile** în procese de *business*, **guvernanța** spune **cum se definesc, se publică, se securizează și evoluează contractele**, astfel echipele pot lucra independent, iar sistemul rămâne predictibil și ușor de operat. Scopul secțiunii nu este un manual complet de SOA, ci încadrarea ei ca nivelul la care practicile de integrare văzute până acum devin un cadru coerent, măsurabil și evolutiv.

Figura arată că SOA nu înseamnă doar existența unor servicii. Este nevoie și de reguli comune: unde sunt descrise serviciile, cine are voie să le folosească, cum se versionază contractele, ce politici de securitate se aplică, ce niveluri de disponibilitate se promit și cum se observă comportamentul sistemului. Integrarea oferă mecanismele tehnice, iar guvernanța stabilește regulile de folosire și evoluție.

2.9.1. ESB, orchestrare (BPEL) și coregrafie (reguli locale)

Enterprise Service Bus, abreviat **ESB**, este un **mijlocitor care primește mesaje, aplică transformări, validează, rutează după reguli și aplică politici de securitate și observabilitate**. Un ESB oferă puncte unice pentru jurnalizare, colectare de metrici, autentificare, autorizare și limitare de rată, în plus expune conectori către protocoale și sisteme eterogene [41]. Utilitatea principală este standardizarea traversării interfețelor, însă ESB nu trebuie să devină locul unde se scrie logica de *business*, altfel apare un monolit distribuit dificil de schimbat.

Orchestrarea și coregrafia sunt **două moduri de coordonare a unui flux distribuit**. Figura 2.23 compară orchestrarea cu alternativa bazată pe coregrafie și reguli locale.

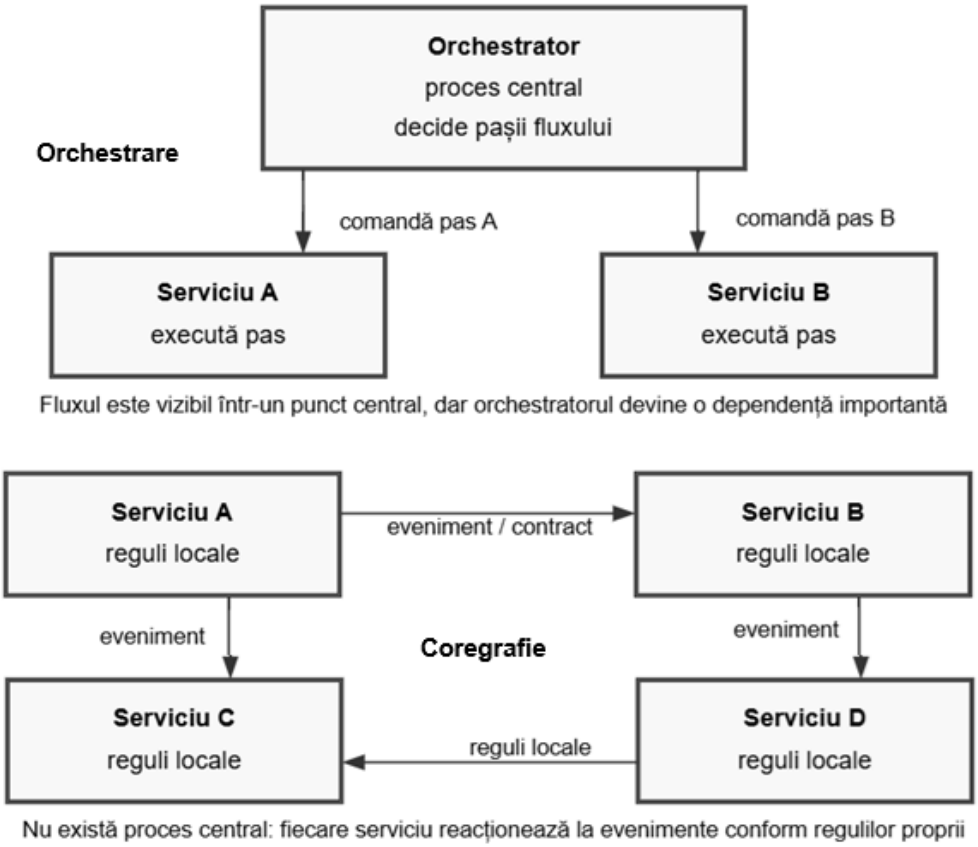


Figura 2.23. Orchestrare față de coregrafie (proces central față de reguli locale)

În **orchestrare**, un proces central decide ordinea pașilor, invocă serviciile participante și poate trata erorile, retransmisiile și compensațiile. Fluxul este mai ușor de urmărit și de controlat, deoarece există un punct central în care se vede starea procesului. Costul este dependența de orchestrator sau de motorul de proces, care trebuie operat atent din punctul de vedere al scalării, rezilienței și migrației definițiilor de proces.

Un exemplu clasic de **orchestrare** este **BPEL**, *Business Process Execution Language*, care permite definirea declarativă a pașilor, ramurilor, corelațiilor de mesaje și tranzacțiilor logice. Acest tip de abordare este potrivit când procesul de *business* are o ordine clară, compensații importante și nevoie de vizibilitate centrală.

În **coregrafie** nu există un coordonator central. Fiecare serviciu emite sau consumă mesaje pe baza unor contracte publice și aplică reguli locale. Fluxul global rezultă din colaborarea serviciilor, nu dintr-un motor unic care comandă toți pașii. Avantajul este autonomia mai mare a echipelor, scalarea mai naturală și posibilitatea de a adăuga consumatori noi fără modificarea serviciilor existente. Costul este că procesul global devine mai greu de observat și de diagnosticat, deci sunt necesare contracte de evenimente clare, înregistrări de jurnal corelate și urmărirea parcursului fluxului între servicii.

Practic, procesele bine structurate, cu pași ordonați și reguli de compensare, se potrivesc mai bine cu **orchestrarea**. Domeniile cu schimbări frecvente, reacții independente și variabilitate mare se potrivesc mai bine cu **coregrafia**. În sisteme reale, cele două modele se combină adesea: procesele critice sunt orchestrate, iar evenimentele sunt folosite pentru extensii, integrare, raportare și actualizarea unor date derivate.

Recomandările rămân aceleași indiferent de model: transformările de date se păstrează cât mai aproape de granițe, regulile de rutare se versionează ca și contractele, iar politicile de securitate și de limitare de rată (*rate limiting*) se aplică uniform, fie în ESB, fie în *gateway*. Pentru operare sunt necesare măsurători clare pentru debit, latență și rate de eroare [43].

2.9.2. Registru, politici și versionarea contractelor

Registrul de servicii SOA este catalogul unde se publică metadate despre servicii, nume, descrieri, *endpoint*-uri, versiuni, scheme (OpenAPI, XSD), politici de acces, contacte operaționale. Consumatorii caută serviciile în registru și verifică detaliile înainte de integrare. Un registru bun păstrează istoricul versiunilor, legături către documentație și starea de sănătate raportată de observabilitate [38], [40].

Registrul **nu este doar o listă de *endpoint*-uri**. El poate conține descrierea serviciului, contractul, versiunea, proprietarul, starea curentă, politica de securitate și informații de compatibilitate. Pentru consumatori, registrul clarifică ce serviciu există și cum se folosește. Pentru furnizori, oferă un loc în care schimbările de contract pot fi anunțate și urmărite.

Politicile stabilesc reguli non-funcționale aplicate consecvent, autentificare și autorizare, criptare în tranzit, limite de rată, reguli de *caching*, cerințe de jurnalizare și corelare a cererilor, precum și obiective de serviciu, adică ținte măsurabile pentru disponibilitate, latență și rate de eroare. Politicile trebuie să fie declarative și atașate contractelor, astfel testele și monitorizarea pot valida automat respectarea lor.

Versionarea contractelor susține evoluția fără întreruperi. Variantele compatibile se livrează prin adăugare de câmpuri opționale, fără a schimba semnificații existente, variantele incompatibile se publică sub o versiune nouă, în antet sau în cale, de exemplu v1, v2. Pentru fiecare schimbare se menține o perioadă de tranziție cu ambele versiuni active, se marchează *deprecated* ce va fi retras, și se oferă ghid de migrare. Testele pot verifica la livrare că furnizorul nu afectează

consumatorii existenți, iar verificările de contract în conducte de procesare blochează schimbările necompatibile. În registru se actualizează întotdeauna versiuni, *endpoint*-urile, scheme și politici asociate. Versionarea contractelor este importantă deoarece consumatorii nu migrează toți în același timp. O versiune nouă ar trebui să fie, pe cât posibil, compatibilă înapoi: câmpuri noi opționale, coduri de eroare documentate, semnificații păstrate. Când o schimbare rupe compatibilitatea, versiunea veche se marchează ca depreciată și se păstrează o perioadă. Astfel, integrarea devine controlată, nu o succesiune de surprize pentru echipele care depind de serviciu.

În ansamblu, **integrarea** definește cum circulă mesajele și cum se compun pașii, **guvernanța** definește cum se scriu și se schimbă contractele. Când ambele sunt aplicate disciplinat, serviciile rămân autonome, schimbările locale nu propagă efecte nedorite, iar operarea se sprijină pe măsurători și reguli verificabile.

Figura 2.24 prezintă **structura mecanismului SOA**: registrul de servicii, consumatorul, furnizorul și politicile comune. Figura 2.25 completează perspectiva structurală cu **secvența tipică de utilizare**: furnizorul publică serviciul, consumatorul consultă registrul, primește contractul sau referința, verifică politicile aplicabile și apoi invocă serviciul.

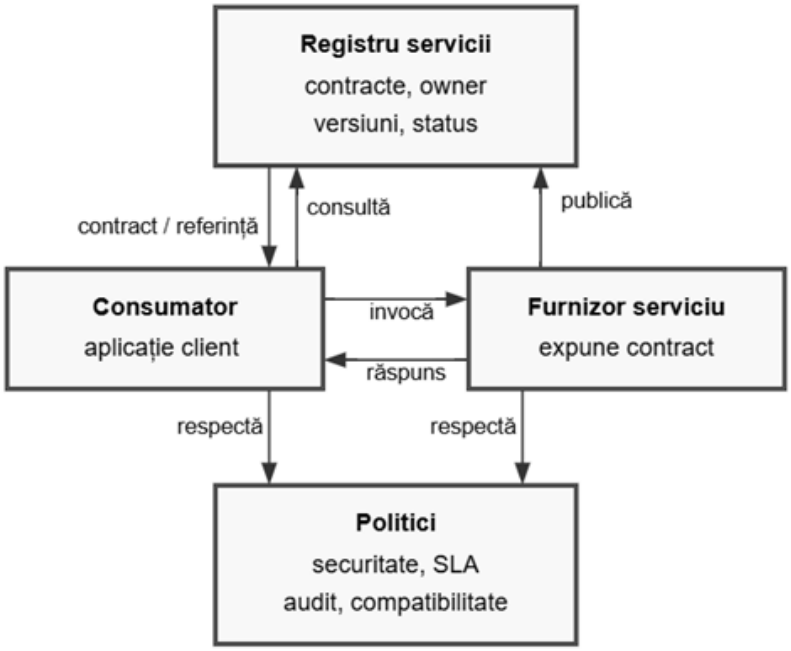


Figura 2.24. Structura mecanismului SOA (registru, consumator, furnizor și politici)

Împreună, cele două figuri separă clar descoperirea serviciului de invocarea efectivă. Registrul nu execută serviciul, ci face contractele vizibile, versionate și controlabile. Politicile nu sunt doar documentație, ci reguli care pot ghida securitatea, auditul, monitorizarea, compatibilitatea și respectarea acordurilor privind nivelul serviciului (*Service Level Agreements*, SLA). Astfel, SOA combină reutilizarea serviciilor cu un mecanism de guvernanță a modului în care serviciile sunt publicate, descoperite, apelate și operate.

SOA a arătat că **serviciile au nevoie de contracte, politici și guvernanță**. Următoarea întrebare este unde și cum rulează aceste servicii. Pe măsură ce aplicațiile au crescut și livrarea a devenit mai rapidă, infrastructura a evoluat de la servere fizice la mașini virtuale, containere și platforme cloud. Această evoluție schimbă modul în care serviciile sunt ambalate, pornite, scalate și operate.

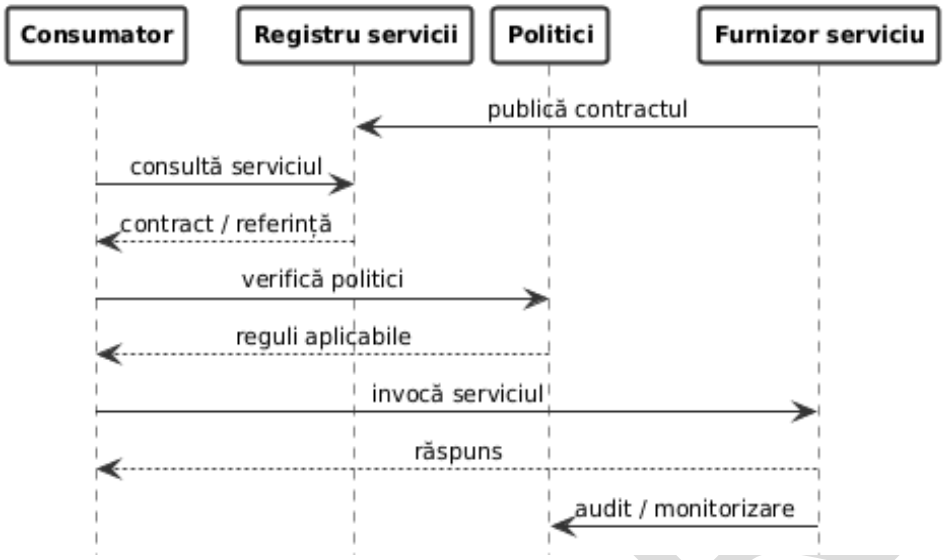


Figura 2.25. Secvența de utilizare SOA prin registru și politici

2.10. Virtualizare, containerizare și cloud

După ce au fost stabilite modurile de integrare la nivel de procese, mesaje și servicii, apare întrebarea practică: **unde rulează componentele**, **cum sunt izolate**, cum sunt **scalate** și cum sunt **operate**. Răspunsul modern combină trei idei compatibile cu SOA și microserviciile: **virtualizarea**, pentru împărțirea sigură a resurselor fizice; **containerizarea**, pentru împachetarea aplicațiilor împreună cu dependențele lor; și **cloudul**, pentru consumul de infrastructură, platforme și servicii gestionate, cu automatizare și facturare în funcție de utilizare [69], [70]. Figura 2.26 sintetizează rolul acestor trei concepte în operarea serviciilor moderne.

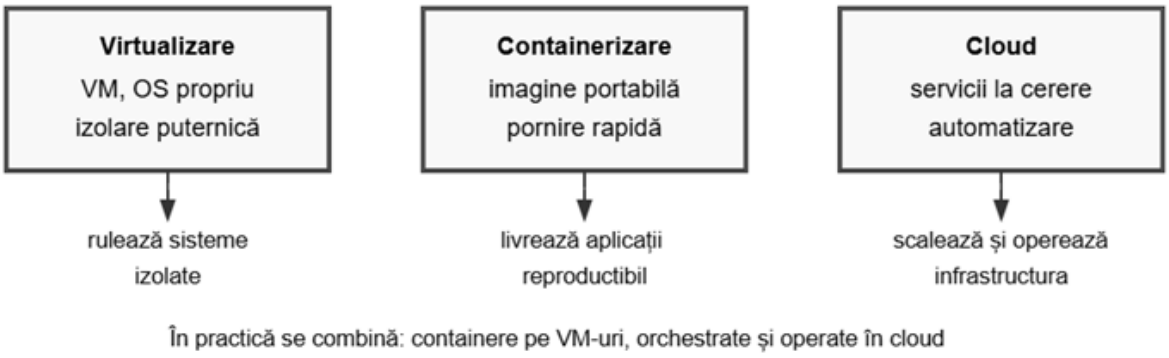


Figura 2.26. Virtualizare, containerizare și cloud (izolare, portabilitate și operare)

Virtualizarea creează mașini virtuale izolate, fiecare cu propriul sistem de operare. **Containerizarea** oferă o unitate portabilă și reproductibilă de livrare a aplicației, pornind mai rapid și folosind mai eficient resursele decât o mașină virtuală completă. **Cloudul** adaugă servicii consumabile la cerere, automatizări de operare, scalare și capabilități gestionate. În practică, cele trei concepte nu se exclud: containerele pot rula pe mașini virtuale, iar ambele sunt frecvent operate într-un mediu cloud.

2.10.1. Virtualizare și containerizare

Virtualizarea împarte în siguranță resursele unui server fizic între mai multe mașini virtuale (VM). Un *hypervisor* pornește VM-uri care au propriul sistem de operare, drivere și resurse izolate. Beneficiul este izolarea puternică și compatibilitatea cu aplicații tradiționale, costul este amprenta mai mare și timpii de pornire mai lungi. Virtualizarea rămâne potrivită pentru aplicații care cer un OS dedicat, pentru separare strictă între clienți și pentru stive software heterogene ce trebuie ținute la distanță.

Containerizarea împachetează aplicația cu dependențele ei și pornește pe același nucleu al gazdei. Izolarea se face prin *namespaces* și *cgroups*, rezultatul este o unitate ușoară care pornește în câteva secunde și se distribuie ca o imagine reproductibilă. *Docker* este ecosistemul cel mai folosit pentru construire și rulare de imagini, alternative notabile sunt *Podman* și *containerd*. Imaginile se publică în registre și se rulează identic pe laptop, în centru de date sau în cloud, ceea ce reduce diferențele de mediu și simplifică testarea [69].

Figura 2.27 compară **cele două moduri de împachetare și izolare**. În cazul mașinii virtuale, fiecare instanță include propriul sistem de operare guest, rulat peste un *hypervisor*. Aceasta oferă o izolare puternică, dar cu un consum mai mare de resurse și timp de pornire mai mare. În cazul containerelor, aplicația și dependențele ei rulează peste un *runtime* de containere și partajează sistemul de operare gazdă. Rezultatul este o pornire mai rapidă și o densitate mai bună de rulare, cu prețul unei izolări mai puțin complete decât în cazul mașinilor virtuale.

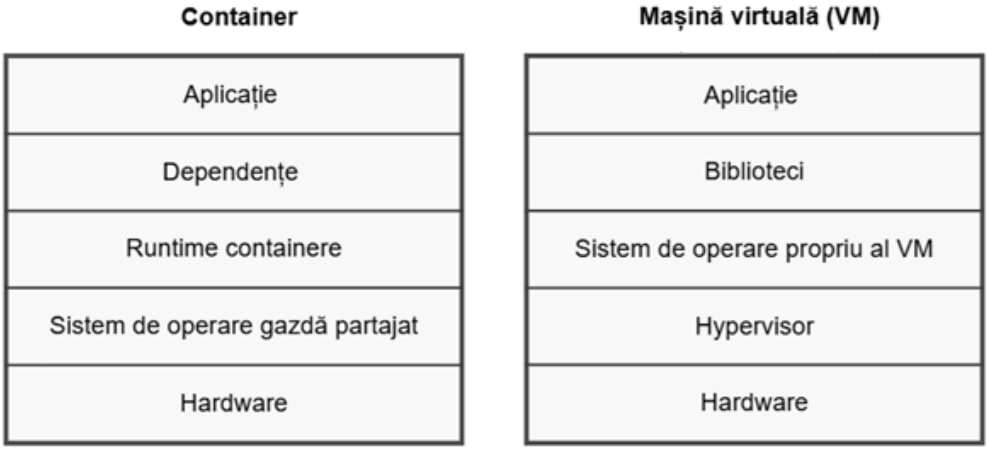


Figura 2.27. Mașină virtuală față de container (OS propriu al mașinii virtuale față de OS gazdă partajat)

Containerele se potrivesc natural cu serviciile. Fiecare microserviciu are *Dockerfile*, *build* automat, imagine versionată, verificări de sănătate și configurare prin variabile de mediu. În producție se scalează orizontal prin multiplicarea instanțelor, iar actualizările se fac gradual, versiunile noi rulează alături de cele vechi până când traficul este comutat. Pentru lucru local, ***Docker Compose*** pornește mai multe servicii cu rețea comună, util pentru scenariile de integrare.

Alegerea între VM și containere este o decizie de izolare, portabilitate și ciclu de viață. VM-urile sunt potrivite pentru monolite, pentru aplicații cu cerințe stricte de OS sau pentru clienți care cer separare puternică. Containerele sunt potrivite pentru servicii web, lucrători pe cozi și procesări scalabile, adică pentru componente care pornesc rapid, expun o verificare de sănătate (*health check*), se configurează din exterior și pot fi multiplicate. În practică se combină, orchestratorul rulează pe VM-uri, iar aplicațiile rulează în containere.

Bune practici esențiale țin de imagini și operare. Imaginile prea mari încetinesc livrarea, se folosesc imagini de bază minimaliste și build-uri în mai multe etape. Configurația în imagine duce la duplicări, se trece pe variabile de mediu și secrete externe. Rețelele implicite pot ascunde dependențe fragile, expunerea se descrie explicit, iar readiness și liveness probes verifică starea reală. Observabilitatea nu vine automat, se introduc înregistrări structurate de jurnal, metrici standard și identificatori de corelație ai cererilor între servicii. Securitatea imaginilor se verifică prin scanare periodică și prin reguli simple (principiul minimului privilegiu, actualizări regulate).

2.10.2. Cloud, orchestrare și operarea serviciilor

Cloud înseamnă că **infrastructura, platformele și unele capabilități de integrare se obțin ca servicii**, nu ca servere administrate manual. La nivel **IaaS** (*Infrastructure as a Service*, infrastructură ca serviciu), se lansează mașini virtuale, volume și rețele. La nivel **PaaS** (*Platform as a Service*, platformă ca serviciu), se folosesc medii de rulare, containere, baze de date gestionate, funcții cloud sau servicii de mesagerie. La nivel **SaaS** (*Software as a Service*, software ca serviciu), se consumă aplicații complete, precum email, CRM sau colaborare [70]. Din perspectiva SOA, avantajul este că rețeaua, stocarea, echilibrarea sarcinilor și securitatea devin elemente modulare, iar echipele se pot concentra mai mult pe contracte, logică de domeniu și operare controlată.

Figura 2.28 arată **diferența de responsabilitate**. La nivel IaaS, echipa primește resurse de infrastructură și administrează mai mult din stiva software. La nivel PaaS, platforma preia o parte din administrare, de exemplu mediu de execuție, baze de date gestionate, containere, funcții cloud sau mesagerie. La nivel SaaS, consumatorul folosește direct aplicația completă, iar furnizorul gestionează aproape întreaga infrastructură. Pe măsură ce urcăm de la IaaS la SaaS, crește gradul de servicii gestionate și scade partea de administrare directă pentru echipa care consumă serviciul.

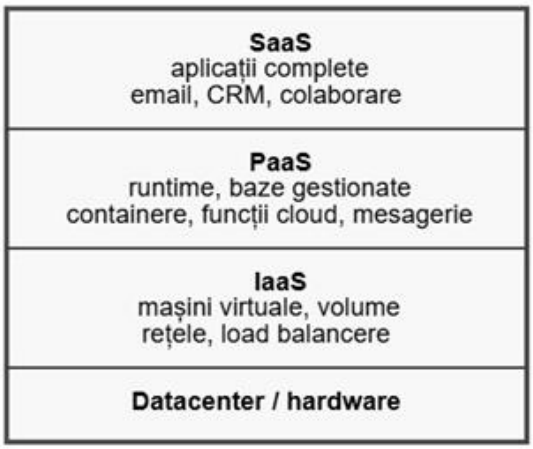


Figura 2.28. Modele cloud (IaaS, PaaS și SaaS)

Pe măsură ce numărul de containere crește, este nevoie de orchestrare. ***Kubernetes*** este standardul de facto pentru programarea, repornirea și scalarea automată a containerelor. El oferă concepte clare: unitatea de rulare (***Pod***), mecanismul care menține numărul dorit de replici și gestionează actualizările graduale (***Deployment***), adresa stabilă în cluster (**Service**), intrarea HTTP din exterior (***Ingress***), structuri pentru configurare și secrete (***ConfigMap*** și ***Secret***) [69]. Platforme precum *OpenShift* sau serviciile cloud gestionate simplifică operarea prin integrare cu registre de imagini, politici, securitate și monitorizare.

Figura 2.29 nu acoperă toate conceptele *Kubernetes*, ci separarea traseului traficului de mecanismul de control și rolurile principale. ***Ingress*** expune intrarea HTTP din exterior, ***Service*** oferă o adresă stabilă în *cluster*, iar traficul ajunge la Pod-urile selectate. ***Deployment*** nu este propriu-zis pe traseul cererii, ci controlează ciclul de viață al ***Pod***-urilor: creează replici, le repornește la nevoie și permite actualizări graduale.

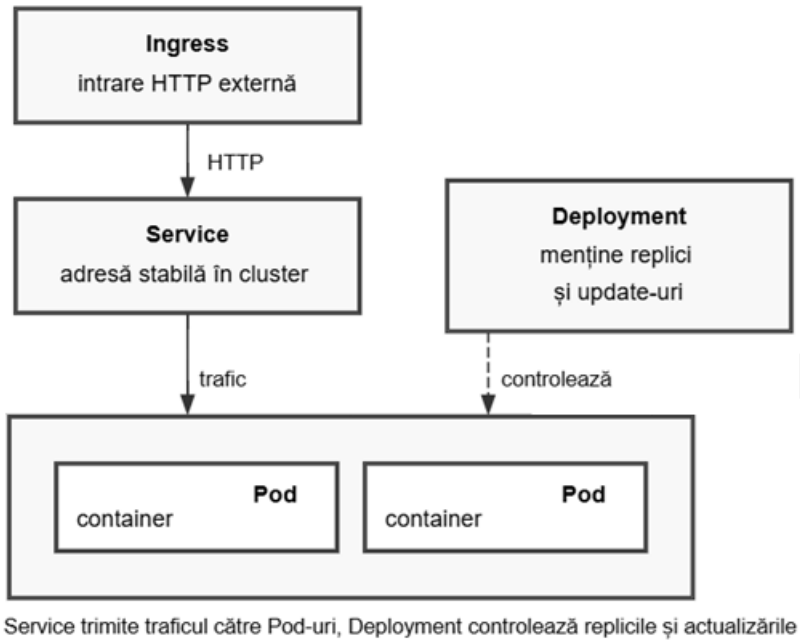


Figura 2.29. Kubernetes simplificat (Ingress, Service, Deployment și Pod-uri)

Cloudul aduce și capabilități gestionate care se potrivesc direct cu modelele din capitol: cozi și topicuri gestionate pentru mesagerie, baze de date gestionate cu replicare și backup, servicii de monitorizare și diagnostic pentru aplicații distribuite. Aceste piese reduc efortul operațional și pot crește fiabilitatea, dar cer aceeași disciplină discutată anterior: contracte clare, politici explicite, versiuni controlate și observabilitate coerentă.

Împreună, virtualizarea oferă izolarea resurselor, containerizarea oferă unitatea portabilă de livrare, iar orchestrarea și cloudul oferă mecanismele prin care serviciile se scalează și se operează în siguranță. Astfel, deciziile de integrare discutate în capitol devin implementabile la scară, cu livrări reproductibile, interfețe clare, izolare, observabilitate și evoluție controlată.

2.11. Concluzii și perspective

Capitolul a urmărit principalele arhitecturi și mecanisme prin care părțile unui sistem software pot fi integrate. **Primul capitol** a discutat **organizarea codului** în funcții, obiecte, componente, *framework*-uri și servicii, pe când **acest capitol** a mutat atenția spre **legăturile dintre acestea**. O arhitectură nu este definită doar de modulele ei, ci și de felul în care acestea schimbă date, invocă operații, publică evenimente, partajează resurse și sunt operate în infrastructură.

La nivel local, **procesele și firele de execuție** arată prima diferență importantă: izolarea față de partajare. Procesele oferă spații de adrese separate și protecție mai bună împotriva erorilor, dar au nevoie de mecanisme explicite de comunicare. Firele comunică ușor prin memoria aceluiași

proces, dar cer sincronizare pentru a evita condițiile de cursă. De aici apar **mecanismele IPC**: memorie partajată, conducte, socketuri locale, cozi de mesaje și tehnici de transfer optimizat precum *zero-copy*. Fiecare alegere înseamnă un compromis între viteză, simplitate, siguranță și claritatea protocolului.

Integrarea prin **baze de date** a arătat o altă formă de colaborare, aparent simplă, dar adesea periculoasă dacă devine interfață ascunsă între aplicații. O bază de date partajată poate accelera dezvoltarea la început, însă schema comună ajunge rapid un contract implicit greu de schimbat. Modelele mai sănătoase separă proprietatea asupra datelor și folosesc mecanisme precum tranzacții locale, *outbox*, *inbox*, CDC, CQRS sau evenimente pentru schimburi fiabile. *SQLite* a completat discuția ca exemplu de bază de date locală integrată în aplicație, utilă fără server separat.

RPC și RMI au introdus apelul sincron la distanță. Ele fac ca o operație la distanță să semene cu un apel local, prin *stub*-uri, *skeleton*-uri, serializare și contracte. Această asemănare este utilă, dar poate induce în eroare: un apel la distanță poate avea latență, erori de transport, retransmisii și probleme de compatibilitate. *Middleware*-ul clasic, reprezentat prin CORBA, a dus această idee mai departe, încercând să ofere interoperabilitate între limbaje, platforme și sisteme diferite. Lecția istorică este importantă: contractele formale și infrastructura comună sunt utile, dar complexitatea operațională trebuie ținută sub control.

Mesageria a schimbat perspectiva de la apel sincron la colaborare asincronă. Cozile, topicurile, brokerii și modelele *publish-subscribe* permit decuplare în timp, absorbția vârfurilor de trafic și reacții independente ale mai multor consumatori. Totuși, mesageria aduce propriile reguli: confirmări (*ack*), retransmisii, cozi de mesaje eșuate (*dead-letter queue*), idempotență, controlul presiunii din amonte (*backpressure*) și observabilitate. Arhitecturile orientate spre evenimente continuă această direcție: sistemele nu comunică doar prin comenzi directe, ci și prin fapte publicate, precum `OrderPlaced` sau `PaymentConfirmed`. Câștigul este flexibilitatea, costul este consistența eventuală și diagnosticarea mai dificilă a fluxurilor distribuite.

Integrarea web a arătat două stiluri importante. WS-* pune accent pe contract formal, SOAP, WSDL și politici *enterprise*, fiind potrivit în contexte unde interoperabilitatea strictă și securitatea la nivel de mesaj sunt esențiale. REST folosește mai direct mecanismele HTTP, resurse, metode, coduri de stare și reprezentări, de obicei JSON, devenind natural pentru API-uri web, aplicații mobile și microservicii. În ambele cazuri, problema centrală rămâne aceeași: contractul trebuie să fie clar, versionabil și compatibil în timp.

SOA, guvernanța și cloudul au ridicat discuția la nivel de sistem. Serviciile nu sunt doar *endpoint*-uri, ci capabilități care trebuie documentate, securizate, monitorizate și versionate. ESB-ul, orchestrarea, coregrafia, registrele și politicile au încercat să controleze complexitatea integrării la scară mare. Virtualizarea, containerizarea și cloudul au schimbat apoi modul în care aceste servicii sunt ambalate, pornite, scalate și operate. *Kubernetes*, serviciile gestionate, IaaS, PaaS și SaaS arată că arhitectura modernă include nu doar codul, ci și infrastructura ca parte a soluției.

Privind în ansamblu, **nu există un mecanism universal de integrare**. Memoria partajată este rapidă, dar cere disciplină. Baza de date comună este comodă, dar cuplează puternic. RPC este clar pentru operații sincrone, dar ascunde costuri de rețea. Mesageria decuplează, dar complică urmărirea fluxurilor. REST este simplu și larg adoptat, dar trebuie proiectat atent pentru versionare și erori. Evenimentele oferă flexibilitate, dar cer idempotență și observabilitate. Cloudul simplifică operarea unor părți, dar introduce dependențe de platformă și noi forme de complexitate.

Capitolul următor folosește aceste idei în studii de caz, pentru a arăta concret cum se schimbă granițele și comunicarea atunci când aceeași funcționalitate evoluează de la monolit la servicii REST și evenimente.

Capitolul 3. Studii de caz

După ce Capitolul 1 a pus **bazele gândirii software**, de la programarea structurată la servicii, iar Capitolul 2 a prezentat **principalele moduri de integrare**, de la procese locale la integrare web și SOA, acest capitol trece la practică printr-o serie de **evoluții controlate ale aceleiași funcționalități** [32], [37], [22].

Vom folosi un exemplu unic și ușor de recunoscut, gestionarea unei comenzi, plasare comandă, rezervare stoc, inițiere plată, confirmare, și îl vom implementa pe rând ca monolit stratificat, monolit modular, microservicii aliniate pe limite de context, apoi integrare prin evenimente și modernizarea interfețelor publice [22], [42].

La fiecare pas vom urmări ce se câștigă și ce costuri apar, granițele și contractele, testarea și observabilitatea, compatibilitatea în timp și implicațiile operaționale. Scopul nu este să demonstrăm că o variantă este întotdeauna superioară, ci să înțelegem criteriile de alegere și compromisurile, astfel încât aceeași problemă să poată fi construită responsabil în stiluri arhitecturale diferite, păstrând claritatea, testabilitatea și evoluția sigură.

Figura 3.1 sintetizează **traseul studiului de caz**. Nu schimbăm problema de *business*, ci schimbăm granițele arhitecturale. În **prima** variantă, toate piesele sunt în același proces. Pornim de la un monolit stratificat, îl disciplinăm prin module și porturi interne, apoi extragem servicii REST acolo unde autonomia devine utilă. În **a doua** variantă, granițele interne devin mai clare. În **a treia**, unele granițe devin granițe de rețea. În etapa **finală**, unele interacțiuni sunt transformate în evenimente, pentru a reduce cuplarea directă și pentru a permite reacții independente ale altor componente. Colaborarea nu mai este controlată doar prin apeluri directe, ci și prin evenimente.

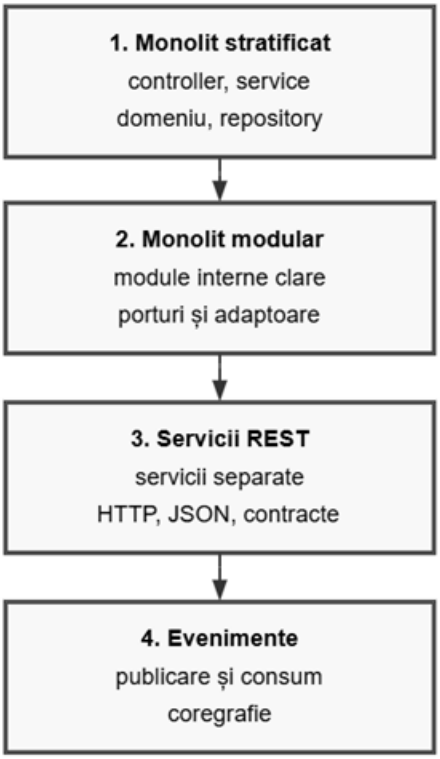


Figura 3.1. Evoluția studiului de caz (de la monolit la servicii și evenimente)

3.1. Codul de start comun (tipuri și porturi reutilizabile)

Înainte de a porni transformările, fixăm un **nucleu stabil** pe care îl vom păstra neschimbat în toate etapele. Acest nucleu conține tipurile de bază ale domeniului, de exemplu comanda și rezultatul operației, precum și porturile, adică interfețele prin care logica aplicației comunică cu exteriorul, stoc, plăți și, mai târziu, un *bus* de evenimente. Ideea este să separăm clar ceea ce vrem să obținem, contractele, de modul concret în care aceste contracte sunt implementate [37], [42].

Această separare ne permite să începem cu un monolit simplu, apoi să înlocuim treptat implementările locale cu unele bazate pe HTTP sau pe mesagerie, fără să modificăm regulile de *business*. Cu alte cuvinte, păstrăm contractele și schimbăm doar adaptoarele. În monolit, porturile sunt implementate prin obiecte locale. În varianta cu servicii, aceleași porturi devin clienți REST. În varianta orientată pe evenimente, apare și un `EventBus` compatibil cu aceeași logică de aplicație. Avantajul este dublu, testarea rămâne simplă, prin dubluri la interfețe, iar fiecare pas de evoluție devine o substituție controlată, nu o rescriere [5], [37].

Figura 3.2 arată **nucleul care rămâne stabil pe parcursul întregului capitol**. Tipurile de domeniu descriu problema de *business*, iar logica aplicației exprimă operațiile principale, independent de infrastructura concretă. Porturile definesc dependențele către exterior, fără a fixa de la început dacă acestea vor fi implementate local, prin HTTP sau prin mesagerie.

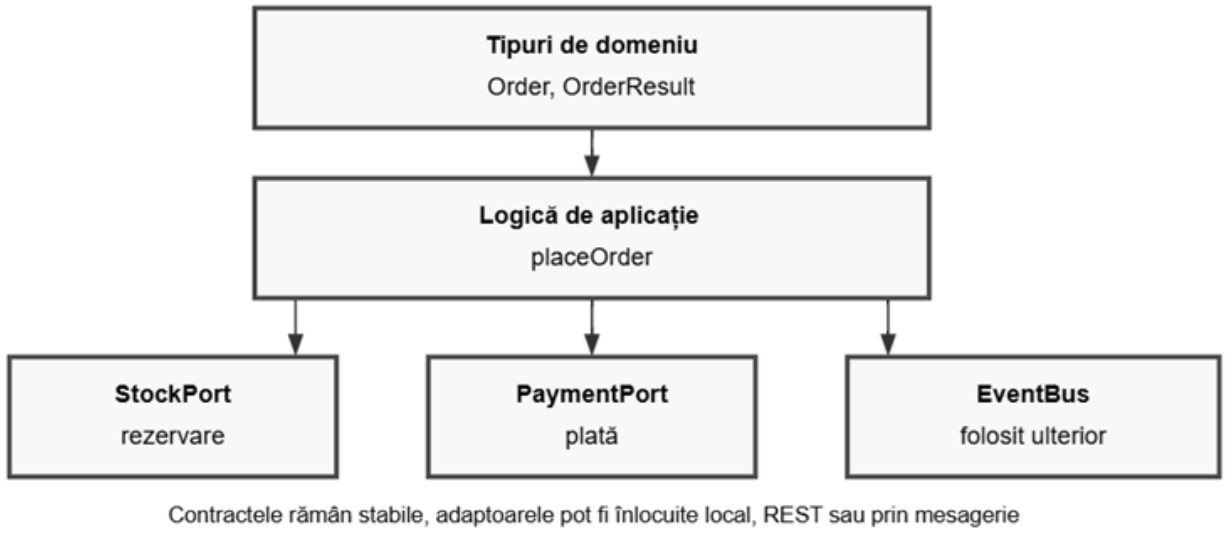


Figura 3.2. Codul de start comun (tipuri de domeniu și porturi reutilizabile)

La început, aceste porturi pot fi legate de implementări locale, potrivite unui monolit. Mai târziu, aceleași contracte pot fi conectate la servicii REST, la baze de date separate sau la un mecanism de publicare a evenimentelor. Acesta este avantajul central al abordării: codul de *business* nu depinde direct de infrastructură, iar schimbările arhitecturale pot fi introduse treptat, prin înlocuirea adaptoarelor, nu prin rescrierea logicii de bază.

```
// codul de start
import java.util.UUID;

// rezultatul contractului placeOrder
public final class Result {
    public final boolean ok;
```

```
    public final String orderId;
    public final String status;
    public final String message;
    public Result(boolean ok, String orderId, String status, String message) {
this.ok = ok; this.orderId = orderId; this.status = status; this.message = message;
    }
    @Override public String toString() {
    return "ok=" + ok + " id=" + orderId + " status=" + status + " msg=" + message;
    }
}

// entitatea minimalista de domeniu
public final class Order {
    public final String id = UUID.randomUUID().toString();
    public final int quantity;
    public final int unitPrice;
    public String status = "NEW";
    public Order(int q, int p) {
        if (q <= 0 || p <= 0) throw new IllegalArgumentException("bad input");
        quantity = q; unitPrice = p;
    }
    public int total() { return quantity * unitPrice; }
}

// porturi (interfete) pentru dependente externe
public interface StockPort {
    boolean reserve(int qty);
    void release(int qty);
}
public interface PaymentPort {
    boolean charge(String orderId, int amount);
}

// extensie pentru etapa event-driven
public interface EventBus {
    void publish(String topic, String payload);
    void subscribe(String topic, java.util.function.Consumer<String> handler);
}
```

Acest cod de start nu rezolvă încă problema completă, ci fixează vocabularul comun al studiului de caz. `Order`, `Result`, `StockPort`, `PaymentPort` și `EventBus` definesc piesele care vor fi refolosite în variantele următoare. Avantajul este că schimbăm arhitectura fără să schimbăm sensul problemei: aceeași comandă, aceleași rezultate, aceleași dependențe exprimate prin contracte. Astfel, diferențele dintre monolit, servicii REST și evenimente devin ușor de urmărit.

3.2. Monolit stratificat: punctul de pornire

Pornim cu o aplicație într-un singur proces, organizată pe straturi clare: **intrare**, **aplicație**, **domeniu** și **infrastructură**. Domeniul conține entitățile și regulile de *business*, stratul de aplicație coordonează cazul de utilizare, infrastructura oferă implementări tehnice pentru persistență și colaboratori, iar metoda `main()` joacă rolul unui punct minim de intrare, un *controller* cu rol didactic. Fluxul „plasează comanda, verifică stoc, inițiază plata, confirmă" rulează local, fără dependențe de rețea și cu latență minimă. Scopul acestei etape este să fixăm **contractul** de intrare/ieșire al operației `placeOrder()` și să fie făcute vizibile granițele interne pe care le vom rearanja, pas cu pas, în variantele modulare, pe servicii și apoi pe evenimente [32]. Figura 3.3 sintetizează această organizare stratificată.

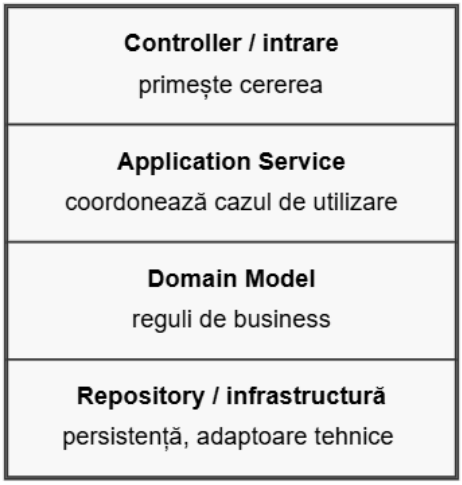


Figura 3.3. Monolit stratificat (separarea intrării, aplicației, domeniului și infrastructurii)

În monolitul stratificat, **toate piesele rulează în același proces, dar responsabilitățile sunt separate**. Intrarea primește cererea, serviciul de aplicație coordonează pașii, domeniul păstrează regulile, iar infrastructura se ocupă de persistență și de implementările tehnice ale porturilor. Avantajul este simplitatea operațională: există o singură aplicație de pornit, testat și livrat. Limita este că, pe măsură ce sistemul crește, granițele dintre responsabilități trebuie păstrate disciplinat, altfel monolitul devine greu de modificat.

```
// App.java monolit
import java.util.*;

// totul intr-un singur fisier pentru simplitate demo
public class App {
    // (re)definim minimal tipurile ca sa ruleze standalone in exemplul monolit
    static final class Result {
        boolean ok;
        String orderId;
        String status;
        String message;
        Result(boolean ok, String orderId, String status, String message) {
            this.ok = ok;
            this.orderId = orderId;
            this.status = status;
            this.message = message;
        }
        @Override
        public String toString() {
          return "ok=" +ok+ " id=" +orderId+ " status=" +status+ " msg=" +message;
        }
    }
    static final class Order {
        final String id = UUID.randomUUID().toString();
        final int quantity;
        final int unitPrice;
        String status = "NEW";
        Order(int q, int p) {
            if (q <= 0 || p <= 0) throw new IllegalArgumentException("bad input");
            this.quantity = q;
            this.unitPrice = p;
        }
        int total() {
```

```
        return quantity * unitPrice;
    }
}

// "persistenta" in memorie
static final class OrderRepo {
    final Map<String, Order> db = new HashMap<>();
    void save(Order o) {
        db.put(o.id, o);
    }
    Optional<Order> find(String id) {
        return Optional.ofNullable(db.get(id));
    }
}

static final class StockLocal {
    int available;
    StockLocal(int initial) {
        this.available = initial;
    }
    boolean reserve(int qty) {
        if (qty <= 0 || available < qty) return false;
        available -= qty;
        return true;
    }
    void release(int qty) {
        if (qty > 0) available += qty;
    }
}

interface PaymentLocal {
    boolean charge(String orderId, int amount);
}
static final class FakePayment implements PaymentLocal {
    @Override
    public boolean charge(String orderId, int amount) {
        // regula demo: accepta sume <= 100
        return amount <= 100;
    }
}

static final class OrderService {
    final OrderRepo orders;
    final StockLocal stock;
    final PaymentLocal pay;
    OrderService(OrderRepo o, StockLocal s, PaymentLocal p) {
        this.orders = o;
        this.stock = s;
        this.pay = p;
    }

    // contract: intrari simple, rezultat minimal
    Result placeOrder(int qty, int unitPrice) {
        // fail fast pe argumente invalide
        if (qty <= 0 || unitPrice <= 0) {
            return new Result(false, null, "REJECTED", "invalid input");
        }
        Order o = new Order(qty, unitPrice);
        orders.save(o);

        // 1) rezervare stoc
        if (!stock.reserve(qty)) {
            o.status = "REJECTED";
```

```
                return new Result(false, o.id, o.status, "no stock");
            }
            o.status = "RESERVED";

            // 2) plata
            if (!pay.charge(o.id, o.total())) {
                // compensatie locala: elibereaza stocul rezervat
                stock.release(qty);
                o.status = "REJECTED";
                return new Result(false, o.id, o.status, "payment fail");
            }
            o.status = "PAID";

            // 3) confirmare
            o.status = "CONFIRMED";
            return new Result(true, o.id, o.status, "ok");
        }
    }
    // "controller" minimal: ruleaza trei scenarii
    public static void main(String[] args) {
        OrderRepo orders = new OrderRepo();
        StockLocal stock = new StockLocal(10);
        PaymentLocal pay = new FakePayment();
        OrderService svc = new OrderService(orders, stock, pay);

        System.out.println(svc.placeOrder(2, 30));   // succes
        System.out.println(svc.placeOrder(5, 50));   // esec plata (250>100)
        System.out.println(svc.placeOrder(20, 10));  // esec stoc
    }
}
```

O rulare posibilă afișează aceleași **trei scenarii** pe care le vom urmări și în variantele următoare (identificatorii `UUID` diferă la fiecare execuție):

```
ok=true id=<uuid-1> status=CONFIRMED msg=ok
ok=false id=<uuid-2> status=REJECTED msg=payment fail
ok=false id=<uuid-3> status=REJECTED msg=no stock
```

Codul fixează un **contract** de intrare și ieșire ușor de verificat. Metoda `placeOrder(qty, unitPrice)` primește două valori întregi și întoarce un obiect `Result` cu patru câmpuri: `ok`, `orderId`, `status`, `message`. Acest rezultat este suficient pentru a exprima clar succesul sau eșecul, dar destul de simplu pentru a rămâne stabil când vom schimba arhitectura. Testele pot verifica direct `ok/status/message` fără să inspecteze implementarea.

Stratul de domeniu este reprezentat de clasa `Order`, care concentrează identitatea, starea și regulile locale: validarea intrărilor, calculul totalului și tranzițiile de stare (`NEW` → `RESERVED` → `PAID` → `CONFIRMED` sau `REJECTED`). Modificarea stării se face numai prin pașii orchestrarți de serviciu, astfel regulile sunt într-un singur loc, iar testele devin previzibile. Identificatorul unic generat local (`UUID`) simplifică urmărirea în jurnal și va fi util și când vom separa procesele.

Stratul de persistență este reprezentat de un `OrderRepo` în memorie și de un `StockRepo` care modelează simplu stocul disponibil. Ideea nu este să demonstrăm o bază de date, ci să izolăm responsabilități: salvarea și regăsirea comenzii într-un loc, regulile de stoc în altul. Această separare va permite, în pașii următori, înlocuirea ușoară a acestor clase cu implementări „reale” (de exemplu, bază de date sau API extern) fără a atinge logica de orchestrare.

Stratul de aplicație (orchestrare) este `OrderService`, care orchestrează pașii. Se aplică verificări rapide la intrare, *Fail Fast*, pentru a respinge date invalide cât mai devreme și pentru a

produce mesaje explicite. Apoi se rulează pașii în ordine: rezervarea stocului, plata, confirmarea. În caz de eșec la stoc, fluxul se oprește imediat cu un mesaj explicit; în caz de eșec la plată, se execută o **compensație** locală (eliberarea stocului), apoi comanda este marcată `REJECTED`. La succes, stările avansează până la `CONFIRMED`. Acest șablon introduce, într-un context simplu, ideile pe care le vom folosi ulterior la tranzacții distribuite.

Stratul de prezentare este redus la un `main()` care joacă rolul unui *controller* minimal (rulează câteva scenarii): construiește dependențele, invocă `placeOrder()` pe câteva scenarii și afișează rezultatele. Într-o aplicație web, acest rol ar fi jucat de un *endpoint* HTTP, aici rămâne doar o intrare unică și lizibilă pentru execuția de probă.

Contractul rezultatelor permite teste centrate pe comportament: pentru intrări invalide, pentru lipsă stoc, pentru plată eșuată, pentru succes. Dependențele sunt injectate prin constructor, astfel se pot introduce ușor dubluri (de exemplu, un `PaymentLocal` care eșuează forțat) pentru a acoperi cazuri negative și de margine. De aici, evoluția poate merge natural în două direcții: (1) monolit modular, în care scoatem clasele în module separate păstrând aceeași semnătură `placeOrder()`, sau (2) separare pe servicii, în care *repo*-urile și plățile devin API-uri externe; în ambele, granițele deja evidente fac tranziția predictibilă, iar testele bazate pe contract reduc riscul de regresie [5], [13].

3.3. Monolit modular: aceleași reguli, granițe interne clare

După varianta stratificată, următorul pas este monolitul modular: aceeași aplicație, același proces, dar cu **granițe interne explicite între module**. Ideea este să păstrăm contracte interne, adică **porturi**, și adaptoare la margine, astfel încât logica de domeniu să nu depindă direct de detalii volatile, precum persistența sau integrarea. Contractul `placeOrder()` rămâne neschimbat, însă organizarea internă devine mai disciplinată [37], [42].

În această variantă, modulul ***Orders*** conține logica principală a cazului de utilizare și orchestrează colaborarea prin porturi, de exemplu `StockPort` și `PaymentPort`. Modulele ***Stock*** și ***Payments*** oferă implementări concrete pentru aceste contracte, încă locale, în același proces. Avantajul este că testarea poate izola ușor modulul de aplicație, injectând dubluri la porturi, iar schimbările de infrastructură sau de integrare rămân locale, fără a afecta codul de domeniu. Figura 3.4 sintetizează această separare în module interne cu contracte explicite.

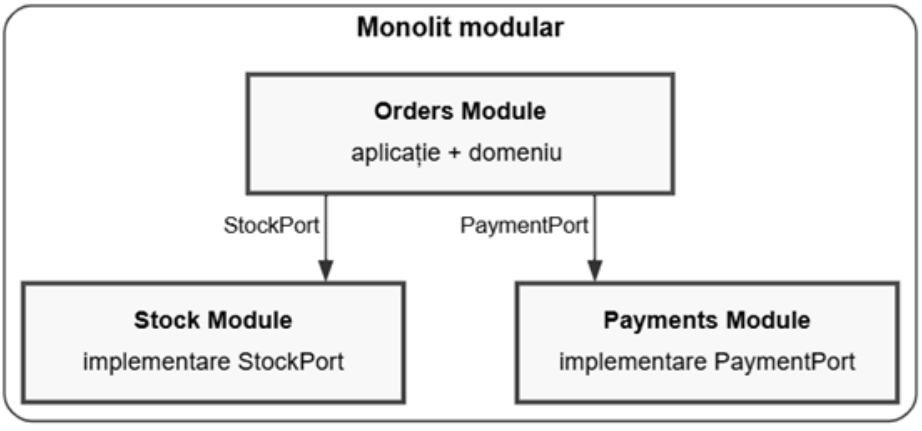


Figura 3.4. Monolit modular (module interne cu granițe explicite)

Monolitul modular păstrează **avantajul unui singur proces**, dar face granițele interne mai ferme. Modulele nu ar trebui să intre direct unele în detaliile celorlalte, ci să colaboreze prin contracte clare. Această variantă este adesea un pas foarte sănătos înainte de microservicii, deoarece obligă echipa să descopere granițele reale ale domeniului fără costul imediat al rețelei, al lansării în producție separate (*deployment*) și al observabilității distribuite [22], [42].

```
// AppModular.java
import java.util.*;

// scop: aceeasi logica, dar prin interfete (porturi) injectate
// ulterior putem inlocui StockPort/PaymentPort cu implementari peste HTTP,
// fara a modifica OrderService sau regulile de domeniu

// Porturi (contracte interne)

interface StockPort {
    boolean reserve(int qty);
    void release(int qty);
}

interface PaymentPort {
    boolean charge(String orderId, int amount);
}

// Tipuri minimale, identice ca intentie cu pasul anterior

final class Result {
    public final boolean ok;
    public final String orderId;
    public final String status;
    public final String message;

    Result(boolean ok, String id, String st, String msg) {
        this.ok = ok; this.orderId = id; this.status = st; this.message = msg;
    }
    @Override
    public String toString() {
        return "ok=" +ok+ " id=" +orderId+ " status=" + status + " msg=" + message;
    }
}

final class Order {
    public final String id = UUID.randomUUID().toString();
    public final int quantity;
    public final int unitPrice;
    public String status = "NEW";

    Order(int q, int p) {
        if (q <= 0 || p <= 0) throw new IllegalArgumentException("bad input");
        this.quantity = q; this.unitPrice = p;
    }
    int total() { return quantity * unitPrice; }
}

// Persistenta in memorie (adaptor local)

final class OrderRepo {
    final Map<String, Order> db = new HashMap<>();
    void save(Order o) { db.put(o.id, o); }
}
```

```
// Adaptoare concrete pentru porturi

final class StockInMemory implements StockPort {
    int available;

    StockInMemory(int init) { this.available = init; }

    @Override
    public boolean reserve(int qty) {
        if (qty <= 0 || available < qty) return false;
        available -= qty;
        return true;
    }
    @Override
    public void release(int qty) {
        if (qty > 0) available += qty;
    }
}

final class PaymentFake implements PaymentPort {
    @Override
    public boolean charge(String orderId, int amount) {
        // regula demo: accepta sume <= 100
        return amount <= 100;
    }
}

// Serviciul de aplicatie: orienteaza fluxul prin porturi

final class OrderService {
    final OrderRepo repo;
    final StockPort stock;
    final PaymentPort pay;

    OrderService(OrderRepo r, StockPort s, PaymentPort p) {
        this.repo = r; this.stock = s; this.pay = p;
    }
    Result placeOrder(int qty, int price) {
        // fail fast pe argumente invalide
        if (qty <= 0 || price <= 0)
            return new Result(false, null, "REJECTED", "invalid input");

        Order o = new Order(qty, price);
        repo.save(o);

        // 1) rezervare stoc
        if (!stock.reserve(qty)) {
            o.status = "REJECTED";
            return new Result(false, o.id, o.status, "no stock");
        }
        o.status = "RESERVED";

        // 2) plata
        if (!pay.charge(o.id, o.total())) {
            stock.release(qty);         // compensatie locala
            o.status = "REJECTED";
            return new Result(false, o.id, o.status, "payment fail");
        }
        o.status = "PAID";

        // 3) confirmare
        o.status = "CONFIRMED";
        return new Result(true, o.id, o.status, "ok");
```

```
    }
}

// "Controller" minimal pentru demo

public class AppModular {
    public static void main(String[] args) {
        var repo = new OrderRepo();
        var stock = new StockInMemory(10);
        var pay = new PaymentFake();
        var svc = new OrderService(repo, stock, pay);
        System.out.println(svc.placeOrder(2, 30));  // succes
        System.out.println(svc.placeOrder(5, 50));  // esec plata (250)
        System.out.println(svc.placeOrder(20, 10)); // esec stoc
    }
}
```

Rezultatul rămâne comparabil cu monolitul stratificat; se schimbă doar felul în care sunt legate dependențele:

```
ok=true id=<uuid-1> status=CONFIRMED msg=ok
ok=false id=<uuid-2> status=REJECTED msg=payment fail
ok=false id=<uuid-3> status=REJECTED msg=no stock
```

Parcurgerea pașilor este aceeași ca în monolitul stratificat (rezervare stoc, plată, confirmare). În monolitul stratificat, dependențele erau **clase concrete** locale; aici, dependențele sunt porturi (**interfețe**), iar ceea ce am numit „adaptoare” (implementări concrete) stau la margine. Câștigul este dublu: (1) testele de aplicație pot injecta dubluri simple ale porturilor, acoperind ușor cazuri de succes/eșec fără a schimba codul din `OrderService`; (2) evoluția infrastructurii devine locală—dacă pe viitor `PaymentPort` trebuie să apeleze un serviciu HTTP real, adăugăm un adaptor `PaymentHttp` care implementează interfața, fără a atinge logica de orchestrare sau regulile de domeniu.

Această formă „cu porturi și adaptoare” este **puntea naturală către servicii**. Când vom sparge în procese separate, `StockPort` și `PaymentPort` vor fi implementate peste rețea (RPC/REST), iar `OrderService` rămâne neschimbat. Contractul rezultatului rămâne stabil, ceea ce simplifică atât testarea, cât și „migrarea” spre microservicii.

3.4. Servicii REST și orchestrare

După monolitul modular, **păstrăm aceleași reguli de domeniu și aceleași porturidar mutăm două colaborări peste rețea**. Concret, ***Stock*** și ***Payment*** devin procese separate accesate prin HTTP, iar `OrderService` rămâne codul care aplică regulile: validează intrarea, rezervă stocul, inițiază plata, confirmă sau compensează. Pentru simplitate, folosim clasele standard din JDK: `com.sun.net.httpserver.HttpServer` pentru a porni servere HTTP minimaliste și `java.net.HttpURLConnection` pentru client. Regulile generale pentru contracte REST au fost discutate în secțiunea 2.7.2; aici urmărim doar cum se aplică ele în studiul de caz. Figura 3.5 sintetizează orchestrarea.

În această variantă, granițele interne devin granițe de rețea. ***Orders Service*** orchestrează fluxul și apelează ***Stock Service*** și ***Payment Service*** prin HTTP. Câștigul este autonomia mai mare a serviciilor, dar apar costuri noi: latență, depășirea timpului de așteptare (*timeout*), erori de transport, versionare de API și compatibilitate în timp, conform regulilor discutate în 2.7.2.

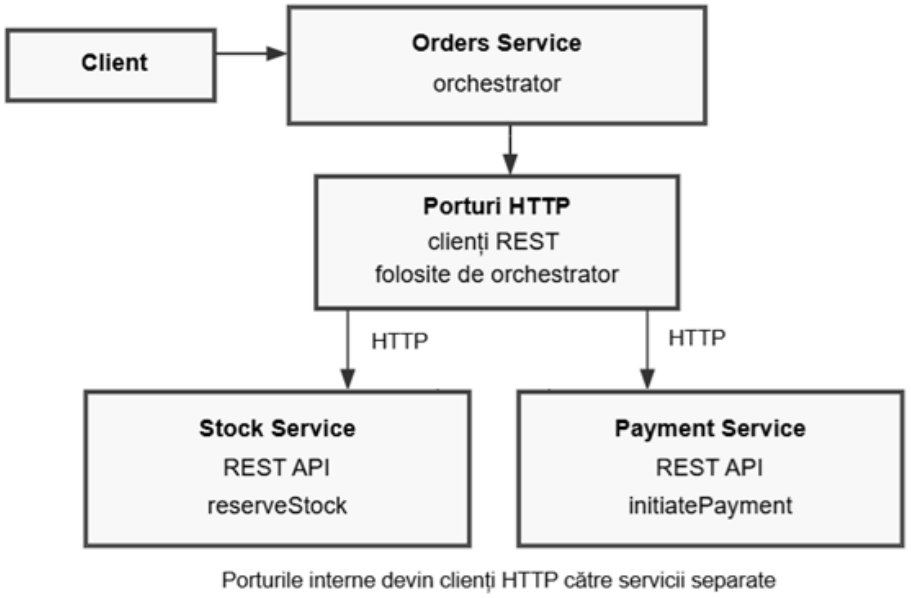


Figura 3.5. Servicii REST cu orchestrare (orchestratorul coordonează apelurile)

Serviciul *Stock* (server HTTP minimal) **expune două operații HTTP**:

- `POST /reserve` care primește cantitatea cerută și încearcă să o reserve
- `POST /release` primește cantitatea care trebuie eliberată, ca operație de compensare locală în caz de eșec la plată.

```
// StockServer.java
import com.sun.net.httpserver.*;
import java.io.*;
import java.net.*;
import java.nio.charset.StandardCharsets;

public class StockServer {
    // stoc in memorie, partajat in server
    static int available = 10;

    public static void main(String[] args) throws Exception {
        HttpServer srv = HttpServer.create(new InetSocketAddress(8081), 0);
        // POST /reserve?qty=Q
        srv.createContext("/reserve", ex -> {
            // validari simple
            var qStr = getQueryParam(ex.getRequestURI().getQuery(), "qty");
            int qty = Integer.parseInt(qStr);
            boolean ok = qty>0 && available>=qty;
            if(ok) available -= qty;
            write(ex, 200, ok ? "OK" : "NO");
        });
        // POST /release?qty=Q
        srv.createContext("/release", ex -> {
            var qStr = getQueryParam(ex.getRequestURI().getQuery(), "qty");
            int qty = Integer.parseInt(qStr);
            if(qty>0) available += qty;
            write(ex, 200, "OK");
        });
        srv.start();
        System.out.println("StockServer up on 8081");
    }
```

```
    static String getQueryParam(String q, String key){
        for(String p : q.split("&")){
            var kv = p.split("=");
            if(kv.length==2 && kv[0].equals(key)) return kv[1];
        }
        return "";
    }
    static void write(HttpExchange ex,int code,String body) throws IOException {
        byte[] b = body.getBytes(StandardCharsets.UTF_8);
        ex.sendResponseHeaders(code, b.length);
        try(OutputStream os = ex.getResponseBody()){ os.write(b); }
    }
}
```

În 3.3 apelam direct o interfață `StockPort` cu o implementare locală. Aici, același contract se „vede” ca HTTP+JSON: ***Stock*** devine un proces separat, are propriul ciclu de viață și propriile informații de diagnostic și monitorizare. Din punctul de vedere al ***Orchestrator***, metoda `reserve()` nu mai e un apel în memorie, ci o **cerere HTTP** cu latență, cod de răspuns și posibile depășiri ale timpului de așteptare. Câștigul este **izolarea** și **scalarea separată** a serviciului de stoc; costul este **latența rețelei** și nevoia de **gestionare explicită a erorilor de transport**.

Serviciul *Payment* (server HTTP minimal) **expune un *endpoint***:

- `POST /charge` cu corp JSON `{ "orderId": "...", "amount": <numar> }`. Răspunde cu `{ "ok": true|false }`, după o regulă simplă (de demo) care respinge sumele „mari”. Și aici folosim exclusiv `HttpServer`, răspunsuri JSON și statusuri `200` / `400` pentru a păstra stilul uniform cu ***Stoc***.

```
// PaymentServer.java
import com.sun.net.httpserver.*;
import java.io.*;
import java.net.*;
import java.nio.charset.StandardCharsets;

public class PaymentServer {
    public static void main(String[] args) throws Exception {
        // porneste un server HTTP minimal pe portul 8082
        HttpServer srv = HttpServer.create(new InetSocketAddress(8082), 0);
        // ruta: POST /charge?orderId=...&amount=...
        srv.createContext("/charge", ex -> {
            String query = ex.getRequestURI().getQuery();
            int amount = Integer.parseInt(getQueryParam(query, "amount"));
            // regula demo: platile <= 100 sunt acceptate
            boolean ok = amount <= 100;
            writeResponse(ex, 200, ok ? "OK" : "NO");
        });
        srv.start();
        System.out.println("PaymentServer up on 8082");
    }
    // extrage valoarea unui parametru din query string (ex. "a=1&b=2")
    static String getQueryParam(String query, String key) {
        if (query == null || query.isEmpty()) return "";
        for (String pair : query.split("&")) {
            String[] kv = pair.split("=", 2);
            if (kv.length == 2 && kv[0].equals(key)) {
                return kv[1];
            }
        }
        return "";
    }
```

```
    // trimite raspuns text simplu
    static void writeResponse(HttpExchange ex, int code, String body) throws
IOException {
        byte[] bytes = body.getBytes(StandardCharsets.UTF_8);
        ex.sendResponseHeaders(code, bytes.length);
        try (OutputStream os = ex.getResponseBody()) {
            os.write(bytes);
        }
    }
}
```

Față de `PaymentPort` din 3.3, acum ***Payment*** este **un serviciu** cu propria sa interfață HTTP. Contractul JSON devine parte din „limbajul comun" dintre procese, ceea ce facilitează înlocuirea implementării fără a atinge ***Orchestrator***. În schimb, trebuie să definim clar **idempotenta** (ce se întâmplă dacă retrimiți aceeași comandă de plată), **depășirile timpului de așteptare** și **codurile de eroare**.

Orchestrator (client HTTP + regulile existente) joacă același rol ca `OrderService` din 3.3, dar apelează ***Stock*** și ***Payment*** prin HTTP. Păstrează contractul extern `placeOrder(qty, unitPrice)` și rezultatul `{ ok, orderId, status, message }`.

Pas cu pas, ***Orchestrator***:

1. validează intrarea,
2. trimite `POST /reserve` la ***Stoc***,
3. dacă reușește, trimite `POST /charge` la ***Plăți***,
4. dacă plata eșuează, **compensează** prin `POST /release` la ***Stoc***,
5. în caz de succes marchează `CONFIRMED` și întoarce rezultatul.

```
// Orchestrator.java
import java.net.*;
import java.io.*;
import java.util.*;

// porturi pentru servicii externe
interface StockPort {
    boolean reserve(int qty);
    void release(int qty);
}
interface PaymentPort {
    boolean charge(String orderId, int amount);
}

// implementare HTTP pentru Stock
final class HttpStock implements StockPort {
    final String base = "http://localhost:8081";
    @Override
    public boolean reserve(int qty) {
        return "OK".equals(call(base + "/reserve?qty=" + qty));
    }
    @Override
    public void release(int qty) {
        call(base + "/release?qty=" + qty);
    }

    static String call(String url) {
        try {
            URL u = URI.create(url).toURL();
            HttpURLConnection c = (HttpURLConnection) u.openConnection();
            c.setRequestMethod("POST");
```

```
            c.setConnectTimeout(1000);
            c.setReadTimeout(1000);
            c.setDoOutput(true);
            try (OutputStream os = c.getOutputStream()) {
                os.write(new byte[0]); // post fara corp util, doar declansare
            }
            try (InputStream is = c.getInputStream()) {
                return new String(is.readAllBytes());
            }
        } catch (Exception e) {
            return "NO";
        }
    }
}

// implementare HTTP pentru Payment
final class HttpPayment implements PaymentPort {
    final String base = "http://localhost:8082";
    @Override
    public boolean charge(String orderId, int amount) {
        return "OK".equals(
           HttpStock.call(base+ "/charge?orderId=" +orderId+ "&amount=" +amount)
        );
    }
}
// tip rezultat minimal
final class Result {
    final boolean ok;
    final String orderId;
    final String status;
    final String message;
    Result(boolean ok, String id, String st, String msg) {
        this.ok = ok;
        this.orderId = id;
        this.status = st;
        this.message = msg;
    }

    @Override
    public String toString() {
        return "ok=" +ok+ " id=" +orderId+ " status=" + status + " msg=" + message;
    }
}

// entitate de domeniu
final class Order {
    final String id = UUID.randomUUID().toString();
    final int quantity;
    final int unitPrice;
    String status = "NEW";
    Order(int q, int p) {
        if (q <= 0 || p <= 0) throw new IllegalArgumentException("bad input");
        this.quantity = q;
        this.unitPrice = p;
    }
    int total() {
        return quantity * unitPrice;
    }
}

// "persistenta" in memorie
final class OrderRepo {
    final Map<String, Order> db = new HashMap<>();
```

```
    void save(Order o) {
        db.put(o.id, o);
    }
}

// orchestratorul regulilor de business
final class OrderService {
    final OrderRepo repo;
    final StockPort stock;
    final PaymentPort pay;
    OrderService(OrderRepo r, StockPort s, PaymentPort p) {
        this.repo = r;
        this.stock = s;
        this.pay = p;
    }
    Result placeOrder(int qty, int price) {
        if (qty <= 0 || price <= 0) {
            return new Result(false, null, "REJECTED", "invalid input");
        }
        Order o = new Order(qty, price);
        repo.save(o);
        // rezervare stoc
        if (!stock.reserve(qty)) {
            o.status = "REJECTED";
            return new Result(false, o.id, o.status, "no stock");
        }
        o.status = "RESERVED";
        // plata
        if (!pay.charge(o.id, o.total())) {
            stock.release(qty); // compensare
            o.status = "REJECTED";
            return new Result(false, o.id, o.status, "payment fail");
        }
        // confirmare
        o.status = "PAID";
        o.status = "CONFIRMED";
        return new Result(true, o.id, o.status, "ok");
    }
}

// punctul de intrare
public class Orchestrator {
    public static void main(String[] args) {
        var repo = new OrderRepo();
        var stock = new HttpStock();
        var pay = new HttpPayment();
        var svc = new OrderService(repo, stock, pay);

        System.out.println(svc.placeOrder(2, 30));  // succes
        System.out.println(svc.placeOrder(5, 50));  // esec plata
        System.out.println(svc.placeOrder(20, 10)); // esec stoc
    }
}
```

Cu `StockServer` și `PaymentServer` pornite local, ieșirea urmărește aceleași scenarii ca în exemplele anterioare:

```
ok=true id=<uuid-1> status=CONFIRMED msg=ok
ok=false id=<uuid-2> status=REJECTED msg=payment fail
ok=false id=<uuid-3> status=REJECTED msg=no stock
```

Regulile din `OrderService` rămân **identice** (*fail-fast*, rezervare, plată, compensație, confirmare), dar acum **depind de rețea** [22], [46]. Asta înseamnă că:

- trebuie setate ***time-out*-uri (durate de așteptare)** la client (`HttpURLConnection`),
- trebuie mapate **coduri HTTP** în mesaje de rezultat,
- trebuie gândită **idempotenta** (dacă „dai *retry*”, să nu dublezi efectele),
- trebuie ca `orderId` să apară în mesajele de diagnostic ale tuturor serviciilor implicate.

Câștigul este că putem porni, opri, scala ***Stock*** și ***Payment*** independent și putem schimba implementările lor fără a atinge orchestrarea. Costul este complexitatea operațională: rețea, latență, intermitență, necesitatea unor **politici explicite de retransmitere** și **tratare a erorilor**.

Folosind regulile REST discutate, contractul API trebuie scris clar: ce *endpoint*-uri există, ce parametri cer, ce răspunsuri trimit. Când evoluează API-ul, se adaugă câmpuri noi (opționale) și se evită schimbarea sensului celor existente, pentru a nu afecta clienții [40], [46], [47].

Testele se fac pe două niveluri:

1. **Unitare** pe ***Orchestrator***, folosind clienți „fake” care răspund cu „`OK`”/„`NO`”, pentru a valida logica fără rețea.
2. **Integrare cap-la-cap**, cu serverele ***Stock*** și ***Payment*** pornite local, pentru a verifica rutele, parametrii și fluxul complet.

Observabilitatea ajută la depanare: `orderId` se scrie în jurnalul fiecărui serviciu, iar pentru fiecare apel se notează dacă a reușit sau nu și cât a durat. Dacă ulterior se introduce un broker de mesaje, același identificator poate fi folosit pentru urmărirea parcursului complet al comenzii [43].

Avem aceeași orchestrare din varianta modulară, dar acum ca **servicii REST simple**, cu granițe clare și contracte stabile, pregătite pentru pasul următor (evenimente).

3.5. Evenimente și coregrafie

În loc ca o componentă să o cheme direct pe alta și să aștepte răspuns, trecem la **un model în care fiecare pas emite evenimente**, iar **celelalte componente se abonează la evenimentele relevante**. Practic, `OrderPlaced` devine un mesaj publicat pe un canal. Serviciul de stoc îl consumă și încearcă rezervarea, apoi publică un eveniment de rezultat, de exemplu `StockReserved`. Serviciul de plăți reacționează la confirmarea stocului și încearcă încasarea, iar la final publică un rezultat, de exemplu `PaymentCompleted`. Un serviciu de confirmare sau notificare poate transforma aceste rezultate în starea finală vizibilă pentru comandă sau pentru utilizator [21], [22].

Toate componentele comunică printr-un ***bus* de mesaje**, nu direct între ele. Astfel, părțile sunt decuplate în timp, nu trebuie să fie disponibile simultan, în spațiu, pot rula în procese sau noduri diferite, și în ritm, pot procesa evenimentele în propriul ritm. Pentru demo folosim un *bus* simplu în memorie, cu o interfață care poate fi înlocuită ulterior cu un broker real, fără a modifica logica de *business*.

Figura 3.6 arată **diferența față de orchestrarea REST din figura precedentă**. ***Orders Service* nu mai controlează direct toți pașii**, ci **publică un eveniment inițial**. Serviciile interesate consumă evenimentele relevante, aplică propriile reguli locale și publică la rândul lor evenimente noi. Astfel, fluxul global rezultă din reacțiile serviciilor, nu dintr-un coordonator central. Câștigul este reducerea cuplării directe și posibilitatea de a adăuga consumatori noi fără modificarea producătorului. Costul este că traseul complet al unei comenzi devine mai greu de urmărit. Această variantă cere contracte de evenimente clare, idempotentă, *outbox* și identificatori de corelație care apar în mesajele de diagnostic ale serviciilor implicate [22], [43], [50].

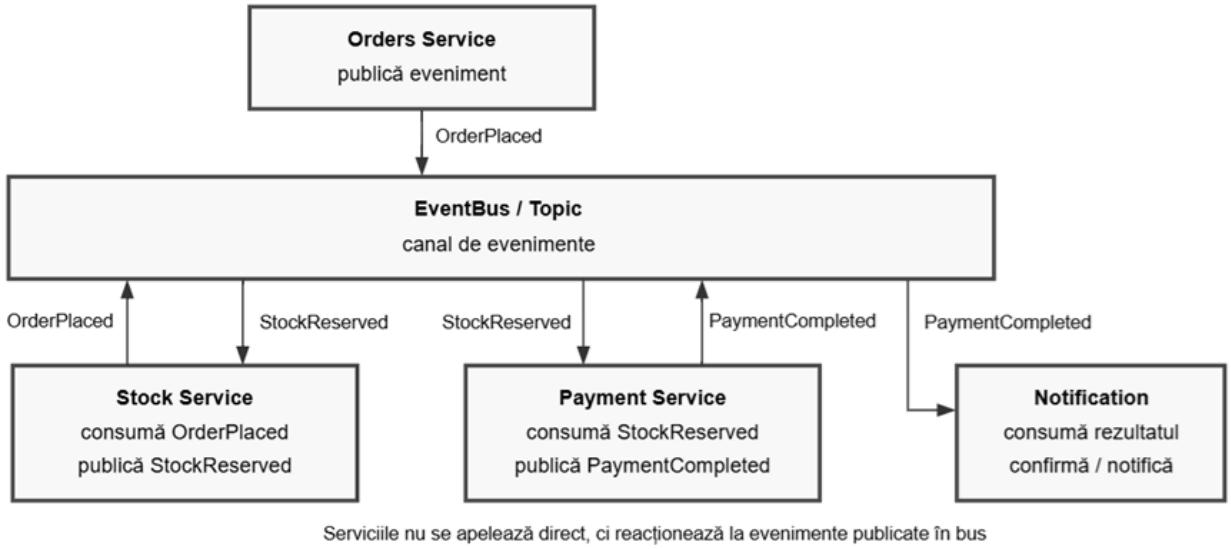


Figura 3.6. Evenimente și coregrafie (servicii care reacționează la evenimente)

`MiniBus` este o **implementare minimală de bus *publish/*subscribe** în memorie, potrivită pentru a înțelege mecanismul fără infrastructură externă. Expune două operații: publicare de evenimente pe un topic și abonare la acel topic cu o funcție (*handler*) care va fi chemată când apare un mesaj. Nu are persistență, nu asigură ordonare strictă și procesează sincron în același proces, dar păstrează aceeași interfață mentală ca un broker real (*RabbitMQ*, NATS). Ideea este să izolăm „cum comunicăm" de „ce logică rulăm", astfel încât, mai târziu, să putem înlocui acest *bus* cu unul de producție fără a rescrie componentele. Avantajul major este că **păstrăm interfața** (*plug-and-play*) **`EventBus`** **și putem schimba implementarea cu una reală** (*RabbitMQ*/NATS) **fără a modifica logica domeniului** [21], [68].

```
// MiniBus.java
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.function.Consumer;

// bus minimal in memorie, sincron, pentru demo-uri si teste rapide
public class MiniBus implements EventBus {
    // mapare topic -> lista de handler-e
    private final Map<String, List<Consumer<String>>> subs =
            new ConcurrentHashMap<>();
    @Override
    public void publish(String topic, String payload) {
        // apeleaza toti abonatii topicului, in acelasi thread (sincron, demo)
        List<Consumer<String>> handlers = subs.getOrDefault(topic, List.of());
        for (Consumer<String> h : handlers) {
            try {
                h.accept(payload);
            } catch (Exception e) { // mesaj simplu de diagnostic
                System.err.println("error on " + topic + ": " + e.getMessage());
            }
        }
    }
    @Override
    public void subscribe(String topic, Consumer<String> handler) {
        subs.computeIfAbsent(topic, t -> new ArrayList<>()).add(handler);
    }
}
```

`AppEvents` pune la lucru *bus*-ul: componenta `Orders` publică „`order.placed`” când utilizatorul face o comandă; componenta `Stock` se abonează la acest eveniment și, dacă poate, publică „`stock.reserved`” sau, altfel, „`stock.rejected`”; componenta `Payments` se abonează la „`stock.reserved`” și publică „`payment.ok`” sau „`payment.ko`”; componenta `Confirmation` se abonează la evenimentele de rezultat și actualizează starea comenzii. Astfel, pașii fluxului nu se mai apelează direct unii pe alții, ci „se prind” de evenimentele relevante. Acest lucru clarifică granițele, permite evoluția independentă a componentelor și pregătește terenul pentru trecerea la un broker real (fără să schimbăm logica internă).

```
// AppEvents.java
import java.util.*;

interface EventBus {
    void publish(String topic, String payload);
    void subscribe(String topic, java.util.function.Consumer<String> handler);
}

// stare minimalista pentru comenzi (doar pentru demo, fara DB)
final class OrdersState {
    final Map<String, String> status = new HashMap<>();
    void set(String id, String s) { status.put(id, s); }
    String get(String id) { return status.get(id); }
}
// componenta care plaseaza comanda si emite evenimentul initial
final class Orders {
    final OrdersState state;
    final EventBus bus;
    Orders(OrdersState s, EventBus b) { state = s; bus = b; }
    String place(int qty, int price) {
        // validari simple
        if (qty<=0 || price <= 0) throw new IllegalArgumentException("bad input");
        // genereaza id local (util pentru corelare)
        String id = java.util.UUID.randomUUID().toString();
        // marcheaza starea initiala
        state.set(id, "NEW");
        // publica evenimentul initial; payload simplu (id, qty, total)
        bus.publish("order.placed", id + "," + qty + "," + (qty * price));
        return id;
    }
}
// componenta de stoc: asculta order.placed si decide rezervarea
final class Stock {
    int available;
    final EventBus bus;
    Stock(int init, EventBus b) { available = init; bus = b; }
    void onPlaced(String payload) {
        // payload in format: id, qty, total
        var parts = payload.split(",");
        String id = parts[0];
        int qty = Integer.parseInt(parts[1]);
        int total = Integer.parseInt(parts[2]);
        // regula simpla de rezervare
        if (qty > 0 && available >= qty) {
            available -= qty;
            bus.publish("stock.reserved", id + "," + qty + "," + total);
        } else {
            bus.publish("stock.rejected", id);
        }
    }
}
```

```
// componenta de plati: asculta stock.reserved si decide plata
final class Payments {
    final EventBus bus;
    Payments(EventBus b) { bus = b; }
    void onReserved(String payload) {
        // payload in format: id, qty, total (ne folosim doar de total)
        var p = payload.split(",");
        String id = p[0];
        int total = Integer.parseInt(p[2]);
        // regula demo: platile <= 100 sunt acceptate
        boolean ok = total <= 100;
        bus.publish(ok ? "payment.ok" : "payment.ko", id);
    }
}
// componenta de confirmare: traduce evenimentele in stari de business
final class Confirmation {
    final OrdersState state;
    Confirmation(OrdersState s) { state = s; }
    void onStockRejected(String id) { state.set(id, "REJECTED(STOCK)"); }
    void onPaymentOk(String id)     { state.set(id, "CONFIRMED"); }
    void onPaymentKo(String id)     { state.set(id, "REJECTED(PAYMENT)"); }
}
public class AppEvents {
    public static void main(String[] args) {
        EventBus bus = new MiniBus();      // usor de inlocuit cu un broker real
        OrdersState state = new OrdersState();
        // instantele „serviciilor" noastre
        Orders orders   = new Orders(state, bus);
        Stock stock     = new Stock(10, bus);
        Payments pay    = new Payments(bus);
        Confirmation cf = new Confirmation(state);
        // legare prin abonari (coregrafie, nu orchestrare centralizata)
        bus.subscribe("order.placed",   stock::onPlaced);
        bus.subscribe("stock.reserved", pay::onReserved);
        bus.subscribe("stock.rejected", cf::onStockRejected);
        bus.subscribe("payment.ok",     cf::onPaymentOk);
        bus.subscribe("payment.ko",     cf::onPaymentKo);
        // trei scenarii de proba
        String a = orders.place(2, 30);   // succes
        String b = orders.place(5, 50);   // esec la plata (total 250)
        String c = orders.place(20, 10);  // esec la stoc
        System.out.println("a -> " + state.get(a));
        System.out.println("b -> " + state.get(b));
        System.out.println("c -> " + state.get(c));
    }
}
```

Pentru că `MiniBus` livrează sincron mesajele în acest demo, stările finale sunt deja vizibile imediat după cele trei apeluri `place`:

```
a -> CONFIRMED
b -> REJECTED(PAYMENT)
c -> REJECTED(STOCK)
```

Am trecut astfel de la apeluri directe la **coregrafie** pe **evenimente**. Orchestratorul dispare, iar fiecare componentă își joacă rolul local: `Orders` emite evenimentul „`order.placed`", `Stock` răspunde fie cu „`stock.reserved`", fie cu „`stock.rejected`", `Payments` publică „`payment.ok`" sau „`payment.ko`", iar `Confirmation` traduce aceste rezultate în stări finale ale comenzii. Nu mai

avem dependențe de tip „A cheamă B”, ci o succesiune de reacții la evenimente care descriu ceea ce s-a întâmplat.

Apare **decuplarea în timp**. Emițătorul nu așteaptă răspunsul consumatorilor. În demo execuția este sincronă pentru simplitate, dar într-un mediu real evenimentele intră în cozi, iar consumatorii procesează când pot. Acest lucru aduce elasticitate (scalare independentă pe pași) și toleranță la căderi locale (dacă un consumator este oprit temporar, recuperează la repornire).

Contractul se mută pe evenimente. Numele topicurilor și schema încărcăturii utile (*payload*) devin contractele dintre părți. Evoluția compatibilă se face prin adăugarea de câmpuri opționale fără schimbarea semnificațiilor celor existente. În practică sunt folosite formate standardizate (JSON, etc) și este versionată explicit schema. În demo am folosit șiruri simple doar pentru claritate [47], [66].

Observabilitatea se bazează pe corelare. `orderId` devine „firul roșu" care unește evenimentele unui flux. În producție, identificatorii de corelație se includ în înregistrări structurate de jurnal și se transmit mai departe împreună cu cererea sau evenimentul, astfel încât traseul cap-la-cap unei comenzi să poată fi reconstruit rapid, cu latențele pe fiecare pas [43].

În demo nu avem baze de date, însă în sisteme reale fiecare componentă își protejează invarianta locală cu tranzacții ACID, iar propagarea către ceilalți se face prin *pattern*-ul *Outbox* (mesajele se salvează în aceeași tranzacție cu modificarea locală, apoi sunt publicate). Consumatorii păstrează un *Inbox* și marchează mesaje procesate, astfel *handler*-ele devin idempotente (reluarea aceluiași eveniment nu strică starea) [22], [50].

Migrarea poate fi graduală: fluxul de *business* rămâne același, schimbăm doar granițele și modul de cuplare. Se poate începe cu `MiniBus` în testare, iar când e totul pregătit se înlocuiește implementarea `EventBus` cu *RabbitMQ* sau NATS, fără să se atingă logica din `Orders`, `Stock`, `Payments`, `Confirmation`. Interfața stabilă permite schimbarea „motorului” de mesaje fără refactorizări ample. Pe scurt, păstrăm același contract de *business*, dar trecem de la apeluri sincrone la colaborare prin evenimente. Codul rămâne lizibil și didactic, iar interfața `EventBus` ne permite să schimbăm ulterior infrastructura de mesagerie fără a rescrie logica domeniului.

3.6. Concluzii

În acest capitol am urmărit aceeași funcționalitate simplă, gestionarea unei comenzi, și am trecut-o prin mai multe forme de organizare, de la monolit stratificat, la monolit modular, apoi la servicii REST cu orchestrare și, în final, la integrare prin evenimente și coregrafie [21], [22], [37]. Tabelul 3.1 rezumă evoluția studiului de caz din perspectiva granițelor arhitecturale, a tipului de comunicare și a costului principal introdus de fiecare variantă.

Tabelul 3.1. Comparație între variantele studiului de caz (granițe, comunicare și costuri)

Varianta	**Graniță principală**	**Comunicare**	**Cost principal**
Monolit stratificat	straturi interne	apel local	cuplare internă
Monolit modular	module interne	porturi locale	disciplină modulară
Servicii REST	procese separate	HTTP sincron	rețea, *timeouts*, versionare
Evenimente	servicii decuplate	mesaje asincrone	observabilitate, consistență eventuală

Tabelul arată **creșterea treptată a autonomiei**, împreună cu **costuri tot mai mari de integrare și operare**. Monolitul stratificat este simplu de operat, însă cere disciplină pentru a evita cuplarea internă. Monolitul modular păstrează simplitatea unui singur proces, dar clarifică dependențele prin porturi. Serviciile REST introduc autonomie de lansare în producție (*deployment*), cu prețul problemelor de rețea. Varianta bazată pe evenimente reduce cuplarea directă, dar cere observabilitate mai bună și reguli clare pentru consistență și idempotentă.

Evoluția arhitecturală nu înseamnă doar „mai multe servicii", ci schimbarea granițelor și a mecanismelor de integrare. **Pe măsură ce granițele devin mai puternice, crește autonomia**, dar **cresc și costurile de integrare, testare și operare**.

Varianta **monolit stratificat** a fixat termenii, un singur proces, tranzacții locale, latență minimă, granițe clare între stratul de aplicație, domeniu și persistență. Monolitul modular a păstrat aceleași reguli, dar a făcut granițele interne mai ferme, prin porturi și adaptoare. Efectul practic este că logica de domeniu nu depinde de detalii volatile, iar înlocuirea unei dependențe se face local, cu teste pe contract. Această disciplină pregătește terenul pentru pașii următori, deoarece contractele deja există și pot deveni interfețe de proces sau de rețea.

Trecerea la servicii REST cu orchestrare a scos dependențele în afara procesului. Am înlocuit apeluri locale cu HTTP, am păstrat contracte simple și am introdus nevoia explicită de a trata erori de transport, depășirea timpului de așteptare (*timeout*) și versiuni. Câștigul este scalarea și independența echipelor, costul este latența, complexitatea infrastructurii și cerința de diagnostic real în producție: înregistrări structurate de jurnal, metrici la nivel de punct de acces și corelare pe identificatorul comenzii. Orchestrarea oferă control și vizibilitate, dar concentrează și riscuri, de aceea este important ca interfețele să fie idempotente și compatibile în timp.

Integrarea prin evenimente și coregrafie a decuplat pașii în timp și în ritm. În loc de apel direct, fiecare componentă reacționează la evenimente, iar fluxul se reconstituie din mesaje. Câștigul este extensibilitatea și reziliența, se pot adăuga consumatori noi fără a atinge sursa, iar vârfurile de trafic se absorb mai ușor. Costul este disciplina contractelor de evenimente, schema, versiuni și reguli de livrare, plus nevoia de a asigura consistența. Monitorizarea și diagnosticul devin esențiale: identificatorul comenzii se transmite între servicii și apare în mesajele de diagnostic ale fiecăruia, pentru a face vizibil parcursul cap-la-cap al comenzii.

În ansamblu, **am ilustrat comparativ două familii de abordări**. Pe de o parte**, client server cu orchestrare**, potrivit când există pași clari și dependențe controlate. Pe de altă parte, **evenimente și coregrafie**, potrivit când vrei echipe independente, integrare ușoară și toleranță mai bună la căderi locale. Aceleași principii din primele două capitole rămân valabile, granițe explicite, contracte stabile, testare pe interfață, compatibilitate în timp și operabilitate. Alegerea stilului se face după context și constrângeri, iar migrarea se poate face gradual, păstrând contractele și mutând încet granițele acolo unde aduc cel mai mult beneficiu.

Alegerea arhitecturii nu este, așadar, o alegere între „vechi" și „nou", ci între forme diferite de control al complexității. O arhitectură bună este aceea care pune granițele acolo unde schimbarea, testarea, operarea și responsabilitatea echipelor o cer.

În loc de încheiere

În locul unei concluzii definitive, această ultimă secțiune propune o perspectivă asupra evoluției paradigmelor și arhitecturilor software. Cartea a urmărit o lectură evolutivă: de la programarea structurată, unde controlul complexității începe cu secvențe clare, decizii, iterații, funcții și structuri de date, până la servicii, integrare, mesagerie, evenimente, containerizare și cloud. Firul comun nu a fost istoria tehnologiilor în sine, ci întrebarea practică: cum împărțim un sistem software în părți suficient de clare, stabile și testabile?

Textul nu își propune să epuizeze toate paradigmele și toate stilurile arhitecturale. El reprezintă perspectiva autorului, formată după o experiență de peste 30 de ani în domeniul ingineriei software și al tehnologiilor de programare în Internet. Din această perspectivă, paradigmele nu trebuie privite ca mode trecătoare care se înlocuiesc complet una pe alta. Programarea structurată rămâne necesară în interiorul metodelor OO. Principiile OO rămân importante în componente și *framework*-uri. Contractele și separarea responsabilităților rămân esențiale în servicii, microservicii și arhitecturi cloud. Tehnologiile se schimbă, dar problemele fundamentale revin în forme noi.

O idee centrală este că arhitectura software nu înseamnă alegerea celei mai noi tehnologii, ci plasarea corectă a granițelor. Uneori granița potrivită este o funcție. Alteori este o clasă, un modul, o componentă, un serviciu, o coadă de mesaje sau un container. O graniță bună ascunde detalii, expune un contract clar și permite schimbarea locală fără efecte necontrolate în restul sistemului. O graniță prost aleasă produce cuplare, duplicare, diagnostic dificil și alte costuri.

Capitolele au arătat și că nu există o soluție universală. Monolitul stratificat poate fi excelent pentru aplicații mici și medii. Monolitul modular poate fi o alegere foarte sănătoasă când granițele de domeniu trebuie clarificate fără costul distribuției. Serviciile REST oferă autonomie, dar aduc latență, depășirea timpului de așteptare, versionare și cerințe mai mari de monitorizare și diagnostic între servicii. Evenimentele decuplează și permit extensii elegante, dar cer idempotentă, corelație, consistență eventuală și diagnostic atent. Cloudul și containerele simplifică unele aspecte operaționale, dar introduc alte responsabilități.

Tendințele recente, *low-code*, *no-code*, *AI-assisted development*, agenții software, pot accelera dezvoltarea, pot deschide programarea către categorii noi de utilizatori și pot automatiza părți importante din efort. Totuși, cu cât codul este produs mai repede, asistat de AI, cu atât devine mai importantă înțelegerea contractelor, testării, securității, observabilității și evoluției în timp. Instrumentele pot genera cod, dar responsabilitatea arhitecturală rămâne la oameni.

În final, arhitectura bună este o formă de luciditate tehnică. Ea nu complică sistemul pentru eleganță teoretică, dar nici nu ignoră complexitatea reală. Alege granițe, contracte și mecanisme de comunicare potrivite contextului. Păstrează codul suficient de simplu pentru a fi înțeles, suficient de modular pentru a fi schimbat și suficient de observabil pentru a fi operat. Aceasta este, poate, lecția comună a tuturor paradigmelor discutate în carte.

Bibliografia a fost selectată, pe cât posibil, din surse accesibile online, având în vedere specificul cititorilor acestei lucrări. Pentru conceptele fundamentale au fost folosite lucrări consacrate. Pentru tehnologii, protocoale și platforme au fost preferate specificații, documentații oficiale sau pagini stabile, pentru ca cititorii să poată consulta și aprofunda rapid sursele.

Abrevieri principale

ACID - Atomicitate, Consistență, Izolare, Durabilitate / *Atomicity, Consistency, Isolation, Durability*
ACK - *Acknowledgement* / confirmare
AI - Inteligență artificială / *Artificial Intelligence*
AMQP - *Advanced Message Queuing Protocol*
API - *Application Programming Interface*
BFF - *Backend for Frontend*
BPEL - *Business Process Execution Language*
CDC - *Change Data Capture*
CLI - *Command-Line Interface*
CORBA - *Common Object Request Broker Architecture*
CPU - *Central Processing Unit*
CQRS - *Command-Query Responsibility Segregation*
CRM - *Customer Relationship Management*
CQS - *Command-Query Separation*
DAO - *Data Access Object*
DCE - *Distributed Computing Environment*
DDD - *Domain-Driven Design*
DI - *Dependency Injection*
DIP - *Dependency Inversion Principle*
DP - *Design Pattern*
DTO - *Data Transfer Object*
EE - *Enterprise Edition*
ETag - *Entity Tag*
EJB - *Enterprise JavaBeans*
ESB - *Enterprise Service Bus*
FF - *Fail Fast*
FIFO - *First In, First Out*
GoF - *Gang of Four*
gRPC - *gRPC Remote Procedure Calls*
HATEOAS - *Hypermedia as the Engine of Application State*
HCLC - *High Cohesion, Low Coupling*
HTML - *HyperText Markup Language*
HTTP - *Hypertext Transfer Protocol*
IaaS - *Infrastructure as a Service*
IDL - *Interface Definition Language*
IIOP - *Internet Inter-ORB Protocol*
IMM - *Immutability*
IoC - *Inversion of Control*
IoT - *Internet of Things*
IPC - *Inter-Process Communication*
ISP - *Interface Segregation Principle*
JAR - *Java Archive*
JAX-WS - *Java API for XML Web Services*
JDBC - *Java Database Connectivity*
JDK - *Java Development Kit*
JMS - *Java Message Service*
JNDI - *Java Naming and Directory Interface*
JPA - *Java Persistence API*

JRMP - *Java Remote Method Protocol*
JSON - *JavaScript Object Notation*
JSON-RPC - *JSON Remote Procedure Call*
JSP - *Jakarta Server Pages* / istoric *JavaServer Pages*
LoD - *Law of Demeter*
LSP - *Liskov Substitution Principle*
MDB - *Message-Driven Bean*
MOM - *Message-Oriented Middleware*
MQTT - *Message Queuing Telemetry Transport*
MVC - *Model-View-Controller*
MVP - *Model-View-Presenter*
MVVM - *Model-View-ViewModel*
OCP - *Open-Closed Principle*
ONC - *Open Network Computing*
OO - Orientare spre obiecte / *Object-Oriented*
ORB - *Object Request Broker*
OS - *Operating System* / sistem de operare
P2I - *Program to an Interface*
PaaS - *Platform as a Service*
POA - *Portable Object Adapter*
POJO - *Plain Old Java Object*
POSIX - *Portable Operating System Interface*
QoS - *Quality of Service*
REST - *Representational State Transfer*
RMI - *Remote Method Invocation*
RPC - *Remote Procedure Call*
SaaS - *Software as a Service*
SAML - *Security Assertion Markup Language*
SLA - *Service-Level Agreement*
SOA - *Service-Oriented Architecture*
SOAP - *Simple Object Access Protocol*
SPI - *Service Provider Interface*
SQL - *Structured Query Language*
SRP - *Single Responsibility Principle*
TCP - *Transmission Control Protocol*
TDA - *Tell, Don't Ask*
TDD - *Test-Driven Development*
UAP - *Uniform Access Principle*
UDDI - *Universal Description, Discovery and Integration*
UDP - *User Datagram Protocol*
UI - *User Interface* / interfață cu utilizatorul
UML - *Unified Modeling Language*
URI - *Uniform Resource Identifier*
URL - *Uniform Resource Locator*
UUID - *Universally Unique Identifier*
VM - Mașină virtuală / *Virtual Machine*
WS-* - familia standardelor *Web Services*
WSDL - *Web Services Description Language*
XML - *Extensible Markup Language*
XML-RPC - *XML Remote Procedure Call*
XSD - *XML Schema Definition*

Bibliografie

[1] E. W. Dijkstra, „Notes on Structured Programming", în O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, *Structured Programming*, Academic Press, 1972, pp. 1-82. Disponibil online: https://research.tue.nl/en/publications/notes-on-structured-programming/

[2] C. Böhm, G. Jacopini, „Flow diagrams, Turing machines and languages with only two formation rules", *Communications of the ACM*, vol. 9, nr. 5, pp. 366-371, 1966. DOI: 10.1145/355592.365646. Disponibil online: https://cacm.acm.org/research/flow-diagrams-turing-machines-and-languages-with-only-two-formation-rules/

[3] D. L. Parnas, „On the criteria to be used in decomposing systems into modules", *Communications of the ACM*, vol. 15, nr. 12, pp. 1053-1058, 1972. DOI: 10.1145/361598.361623. Disponibil online: https://cacm.acm.org/research/on-the-criteria-to-be-used-in-decomposing-systems-into-modules/

[4] B. Meyer, *Object-Oriented Software Construction*, ed. a 2-a, Prentice Hall, 1997. ISBN 0-13-629155-4. Disponibil online: https://www.informit.com/store/object-oriented-software-construction-book-cd-rom-9780136291558

[5] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley Professional, 2007. ISBN 978-0131495050. Disponibil online: https://www.oreilly.com/library/view/xunit-test-*patterns*/9780131495050/

[6] cppreference.com, „assert", documentație pentru macro-ul `assert` din `<assert.h>`. Disponibil online: https://en.cppreference.com/w/c/error/assert

[7] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen, K. A. Houston, *Object-Oriented Analysis and Design with Applications*, ed. a 3-a, Addison-Wesley Professional, 2007. ISBN 978-0201895513. Disponibil online: https://www.oreilly.com/library/view/object-oriented-analysis-and/9780201895513/

[8] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2002. ISBN 978-0135974445. Disponibil online: https://www.pearson.com/subject-catalog/p/agile-software-development-principles-*pattern*s-and-practices/P200000003363

[9] B. Liskov, J. M. Wing, „Behavioral Subtyping Using Invariants and Constraints", *ACM Transactions on Programming Languages and Systems*, vol. 16, nr. 6, pp. 1811-1841, 1994. DOI: 10.1145/197320.197383. Disponibil online: https://www.cs.cmu.edu/afs/cs/project/calder/www/fmdp.html

[10] K. J. Lieberherr, I. M. Holland, „Assuring Good Style for Object-Oriented Programs", *IEEE Software*, vol. 6, nr. 5, pp. 38-48, 1989. DOI: 10.1109/52.35588.

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994. ISBN 978-0201633610. Disponibil online: https://www.oreilly.com/library/view/design-*pattern*s-elements/0201633612/

[12] Object Management Group, *OMG Unified Modeling Language (OMG UML), Version 2.5.1*, formal/17-12-05, 2017. Disponibil online: https://www.omg.org/spec/UML/2.5.1

[13] K. Beck, *Test Driven Development: By Example*, Addison-Wesley Professional, 2002. ISBN 978-0321146533. Disponibil online: https://www.oreilly.com/library/view/test-driven-development/0321146530/

[14] JUnit Team, *JUnit User Guide*, versiunea curentă online. Disponibil online: https://junit.org/junit5/docs/current/user-guide/

[15] J. Hughes, „Why Functional Programming Matters", *The Computer Journal*, vol. 32, nr. 2, pp. 98-107, 1989. DOI: 10.1093/comjnl/32.2.98. Disponibil online: https://www.cse.chalmers.se/~rjmh/Papers/whyfp.html

[16] Oracle, „Package java.util.stream", *Java SE 21 & JDK 21 API Specification*. Disponibil online: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/stream/package-summary.html

[17] Android Developers, „Intents and intent filters", *Android Developers Documentation*. Disponibil online: https://developer.android.com/guide/components/intents-filters

[18] Android Developers, „WorkManager", *Android Developers API Reference*. Disponibil online: https://developer.android.com/reference/androidx/work/WorkManager

[19] D. C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, Wiley, 2000. ISBN 978-0471606956. Disponibil online: https://www.oreilly.com/library/view/*pattern*-oriented-software-architecture/9781118725177/

[20] D. Alur, J. Crupi, D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies*, ed. a 2-a, Prentice Hall, 2003. ISBN 978-0131422469. Disponibil online: https://www.oreilly.com/library/view/core-j2eetm-*pattern*s/0131422464/

[21] G. Hohpe, B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley Professional, 2003. ISBN 978-0321200686. Disponibil online: https://www.enterpriseintegration*pattern*s.com/

[22] C. Richardson, *Microservices Patterns: With examples in Java*, Manning, 2018. ISBN 978-1617294549. Disponibil online: https://www.manning.com/books/microservices-*pattern*s

[23] B. Burns, *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*, O'Reilly Media, 2018. ISBN 978-1491983645. Disponibil online: https://www.oreilly.com/library/view/designing-distributed-systems/9781491983638/

[24] C. Szyperski, D. Gruntz, S. Murer, *Component Software: Beyond Object-Oriented Programming*, ed. a 2-a, Addison-Wesley, 2002. ISBN 978-0201745726. Disponibil online: https://www.cs.ox.ac.uk/publications/publication6058-abstract.html

[25] T. Preston-Werner, „Semantic Versioning 2.0.0". Disponibil online: https://semver.org/

[26] Oracle, *JavaBeans 1.01 Specification*. Disponibil online: https://www.oracle.com/java/technologies/javase/*JavaBeans*-spec.html

[27] Oracle, „Package java.beans", *Java SE 21 & JDK 21 API Specification*. Disponibil online: https://docs.oracle.com/en/java/javase/21/docs/api/java.desktop/java/beans/package-summary.html

[28] Eclipse Foundation, *Jakarta Enterprise Beans, Core Features, Version 4.0*, 2020. Disponibil online: https://jakarta.ee/specifications/enterprise-beans/4.0/jakarta-enterprise-beans-spec-core-4.0

[29] M. Fowler, „Inversion of Control Containers and the *Dependency Injection pattern*", 2004. Disponibil online: https://www.martinfowler.com/articles/injection.html

[30] Android Developers, „The activity lifecycle", *Android Developers Documentation*. Disponibil online: https://developer.android.com/guide/components/activities/activity-lifecycle

[31] T. M. H. Reenskaug, „Models--Views--Controllers", Xerox PARC working paper, 1979. DOI: 10.5281/zenodo.3676092. Disponibil online: https://zenodo.org/records/3676092

[32] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2002. ISBN 978-0321127426. Disponibil online: https://www.oreilly.com/library/view/*pattern*s-of-enterprise/0321127420/

[33] Eclipse Foundation, *Jakarta Servlet Specification, Version 6.0*, 2022. Disponibil online: https://jakarta.ee/specifications/servlet/6.0/jakarta-servlet-spec-6.0

[34] Eclipse Foundation, *Jakarta Pages*. Disponibil online: https://jakarta.ee/specifications/pages/

[35] M. Fowler, „GUI Architectures" și „Presentation Model", 2004-2006. Disponibil online: https://www.martinfowler.com/eaaDev/uiArchs.html și https://martinfowler.com/eaaDev/PresentationModel.html

[36] S. Newman, „*Pattern*: Backends For Frontends", 2015. Disponibil online: https://samnewman.io/*pattern*s/architectural/bff/

[37] A. Cockburn, „Hexagonal Architecture", 2005. Disponibil online: https://alistair.cockburn.us/hexagonal-architecture

[38] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*, Prentice Hall, 2005. ISBN 978-0131858589. Disponibil online: https://www.oreilly.com/library/view/service-oriented-architecture-concepts/0131858580/

[39] W3C, *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, W3C Recommendation, 2007. Disponibil online: https://www.w3.org/TR/wsdl/

[40] OpenAPI Initiative, *OpenAPI Specification v3.1.0*, 2021. Disponibil online: https://spec.openapis.org/oas/v3.1.0.html

[41] D. A. Chappell, *Enterprise Service Bus*, O'Reilly Media, 2004. ISBN 978-0596006754. Disponibil online: https://www.oreilly.com/library/view/enterprise-service-bus/0596006756/

[42] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley Professional, 2003. ISBN 978-0321125217. Disponibil online: https://www.oreilly.com/library/view/domain-driven-design-tackling/0321125215/

[43] OpenTelemetry, *Documentation*. Disponibil online: https://opentelemetry.io/docs/

[44] E. Jonas et al., „Cloud Programming Simplified: A Berkeley View on Serverless Computing", UC Berkeley Technical Report UCB/EECS-2019-3, 2019. Disponibil online: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[45] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, doctoral dissertation, University of California, Irvine, 2000. Disponibil online: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm

[46] R. Fielding, M. Nottingham, J. Reschke, *HTTP Semantics*, RFC 9110, IETF, 2022. DOI: 10.17487/RFC9110. Disponibil online: https://www.rfc-editor.org/rfc/rfc9110

[47] T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 8259, IETF, 2017. DOI: 10.17487/RFC8259. Disponibil online: https://www.rfc-editor.org/rfc/rfc8259

[48] Spring, *Spring Boot Reference Documentation*. Disponibil online: https://docs.spring.io/spring-boot/reference/index.html

[49] React, „useState", *React Documentation*. Disponibil online: https://react.dev/reference/react/useState

[50] MDN Web Docs, „Idempotency-Key *header*". Disponibil online: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/*Header*s/Idempotency-Key

[51] A. Bucaioni, A. Cicchetti, F. Ciccozzi, „Modelling in low-code development: a multi-vocal systematic review", *Software and Systems Modeling*, vol. 21, pp. 1959-1981, 2022. DOI: 10.1007/s10270-021-00964-0. Disponibil online: https://link.springer.com/article/10.1007/s10270-021-00964-0

[52] R. Sabetto, S. Kandwal, D. Agarwal, *Software Engineering with Generative Artificial Intelligence Tools*, MITRE, 2024. Disponibil online: https://www.mitre.org/news-insights/publication/software-engineering-generative-artificial-intelligence-tools

[53] N. A. Otoum, N. Elkhalili, „Methods and Techniques of Agentic Software Engineering: A Systematic Literature Review", *IEEE Access*, 2026. DOI: 10.1109/ACCESS.2026.3652325.

[54] A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, ed. a 4-a, Pearson Higher Education, 2015. ISBN 978-0133591620. Disponibil online: https://research.vu.nl/en/publications/modern-operating-systems-4th-edition/

[55] The Open Group, „fork - create a new process", *The Open Group Base Specifications*. Disponibil online: https://pubs.opengroup.org/onlinepubs/9799919799/functions/fork.html

[56] Oracle, „Class Thread", *Java SE 21 & JDK 21 API Specification*. Disponibil online: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Thread.html

[57] D. R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997. ISBN 978-0201633924. Disponibil online: https://openlibrary.org/works/OL2637209W/Programming*with*POSIX_threads

[58] The Open Group, „shm*open" și „mmap", The Open Group Base Specifications. Disponibil online: https://pubs.opengroup.org/onlinepubs/009695099/functions/shm*open.html și https://pubs.opengroup.org/onlinepubs/9799919799/functions/mmap.html

[59] The Open Group, „pipe" și „mkfifo", *The Open Group Base Specifications*. Disponibil online: https://pubs.opengroup.org/onlinepubs/9799919799/functions/pipe.html și https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/functions/mkfifo.html

[60] Linux man-pages project, „unix(7) - sockets for local interprocess communication". Disponibil online: https://man7.org/linux/man-pages/man7/unix.7.html

[61] The Open Group, „mq*open - open a message queue", The Open Group Base Specifications. Disponibil online: https://pubs.opengroup.org/onlinepubs/9799919799/functions/mq*open.html

[62] Linux man-pages project, „sendfile(2) - transfer data between file descriptors". Disponibil online: https://man7.org/linux/man-pages/man2/sendfile.2.html

[63] Linux man-pages project, „splice(2) - splice data to/from a pipe". Disponibil online: https://man7.org/linux/man-pages/man2/splice.2.html

[64] *SQLite* Project, *SQLite Documentation*, incluzând „Appropriate Uses For SQLite" și „Isolation In SQLite". Disponibil online: https://www.sqlite.org/docs.html

[65] Oracle, *Java Remote Method Invocation Specification*. Disponibil online: https://docs.oracle.com/javase/8/docs/platform/rmi/spec/rmiTOC.html

[66] gRPC Project, *gRPC Documentation*, și Google, *Protocol Buffers Documentation*. Disponibil online: https://grpc.io/docs/ și https://protobuf.dev/

[67] Object Management Group, *Common Object Request Broker Architecture (CORBA), Version 3.4*. Disponibil online: https://www.omg.org/spec/CORBA/3.4

[68] Eclipse Foundation, *Jakarta Messaging 3.1*; AMQP Working Group, *AMQP 0-9-1*; OASIS, *MQTT Specifications*; NATS, *NATS Docs*; ZeroMQ Project, *ZeroMQ Documentation*. Disponibil online: https://jakarta.ee/specifications/messaging/3.1/, https://www.amqp.org/specification/0-9-1/amqp-org-download, https://mqtt.org/mqtt-specification/, https://docs.nats.io/ și https://zeromq.org/

[69] Docker, *Docker Docs*, și Kubernetes, *Kubernetes Documentation*. Disponibil online: https://docs.docker.com/ și https://kubernetes.io/docs/

[70] P. Mell, T. Grance, *The NIST Definition of Cloud Computing*, NIST Special Publication 800-145, 2011. DOI: 10.6028/NIST.SP.800-145. Disponibil online: https://www.nist.gov/publications/nist-definition-cloud-computing

Notă privind utilizarea instrumentelor de inteligență artificială

În procesul de elaborare a acestui material au fost utilizate instrumente de inteligență artificială generativă, inclusiv ChatGPT, dezvoltat de OpenAI, ca sprijin pentru identificarea preliminară a unor referințe și detalii relevante pentru temele tratate, pentru semnalarea eventualelor repetiții excesive sau formulări neclare, precum și pentru verificarea unor exemple didactice, a coerenței redactării și a utilizării corecte și consecvente a diacriticelor în limba română. Utilizarea acestor instrumente s-a realizat punctual, pe fragmente de lucru și pe aspecte editoriale sau tehnice specifice, nu prin delegarea redactării lucrării către un sistem automat. Informațiile, sugestiile și observațiile obținute cu sprijinul acestor instrumente au fost analizate critic, verificate și integrate selectiv de autor.

Utilizarea acestor instrumente nu a substituit contribuția intelectuală a autorului. Conceperea lucrării, elaborarea și redactarea conținutului, selecția temelor, organizarea capitolelor, interpretarea conceptelor, validarea explicațiilor tehnice, verificarea exemplelor de cod și forma finală a textului aparțin autorului. Instrumentele de inteligență artificială nu sunt considerate autori sau coautori ai lucrării, iar responsabilitatea pentru acuratețea, originalitatea și integritatea materialului revine integral autorului. Instrument utilizat: ChatGPT, OpenAI, https://chat.openai.com/, utilizat în perioada 2025–2026.

wwww.matrixrom.ro

9786062510701