# *Proiectarea arhitecturala (Proiectarea de sistem)*

*Prof. univ. dr. ing. Florica Moldoveanu*

# Proiectarea arhitecturala

- Modelul de analiza ofera o vedere externa asupra sistemului, independenta de implementare. El serveste ca mijloc de comunicare intre client si dezvoltator.
- **Proiectarea este etapa in care se construieste** solutia problemei: un **model fizic**.
- Cerintele din Documentul Cerintelor Software sunt alocate unor componente fizice ale viitorului sistem.
- **Proiectarea arhitecturala, numita si proiectare de sistem, deoarece stabileste si configuratia hardware a sistemului, transforma modelul de analiza intr-un model al arhitecturii sistemului.**
- **Principalele activitati:**

  - Definirea obiectivelor proiectarii

  - Descompunea in subsisteme care pot fi realizate de echipe diferite

  - Alegerea strategiilor de construire a sistemului: strategii hardware/software, managementu datelor persistente, fluxul global al controlului, politici de control al accesului, tratarea conditiilor limita.

# De la modelul de analiza la modelul de proiectare

- Cerinte nefunctionale → Obiectivele proiectarii
- Modelul functional → Descompunerea in subsisteme
- Modelul obiect → Distributia subsistemelor pe hardware, managementul datelor persitente
- Modelul dinamic → Activitati concurente, fluxul global al controlului

❖ **Modelul de proiectare arhitecturala cuprinde:**

1. **Obiectivele proiectarii.** Descriu calitati ale sistemului care trebuie urmarite in dezvoltare sau restrictii manageriale. Ele trebuie sa ghideze deciziile dezvoltatorilor atunci cand sunt necesare compromisuri.
2. **Arhitectura software.** Descrie descompunerea sistemului in subsisteme, responsabilitatile subsistemelor si dependentele intre subsisteme, alocarea subsistemelor pe hardware si principalele decizii, cum ar fi: fluxul controlului, controlul accesului la sistem si stocarea datelor persistente.
3. **Condiţiile limită** in functionarea sistemului: configurarea sistemului, pornirea, initializarea, oprirea si tratarea exceptiilor.

# Definirea obiectivelor proiectarii

- **Obiectivele proiectarii sunt derivate din cerintele nefunctionale**, care in etapa de proiectare devin **criterii de proiectare.**

De exemplu, un **obiectiv de proiectare** poate fi un **criteriu de performanta**, cum ar fi timpul de raspuns scurt sau capacitatea sistemului de a executa un numar mare de taskuri. Ambele cerinte sunt mai greu de realizat. Un alt obiectiv de proiectare poate fi asigurarea unei capacitati a memoriei care sa permita optimizarea vitezei de procesare.

- **Criteriile de proiectare** pot fi organizate in 5 grupuri:

  - *performanta*
  - *încrederea in sistem*
  - *costul*
  - *mentenenţa*
  - *criteriile utilizatorului final*

# Criterii de proiectare (1)

- **Criterii de performanţă**

— Timpul de raspuns: în cât timp raspunde sistemul la o cerere a utilizatorului

— Throughput: câte taskuri poate realiza sistemul intr-o perioada de timp fixa

— Memoria: care este spatiul de memorie necesar pentru rularea sistemului

- **Criterii de încredere (Dependability) -** determina efortul care trebuie alocat in dezvoltare pentru a minimiza caderile sistemului si consecintele lor.

— Robustetea (Robustness): capacitatea de a rezista la intrari utilizator nevalide.

— Fiabilitatea (Reliability): capacitatea de a functiona conform comportamentului specificat.

— Disponibilitatea (Availability): procentul de timp in care sistemul poate fi folosit.

— Toleranta la defecte (Fault tolerance): capacitatea de a functiona in conditii de eroare.

— Securitatea (Security): capacitatea de a rezista la atacuri.

— Siguranta (Safety): capacitatea de a nu pune in pericol vieti omenesti chiar si in prezenta erorilor de operare sau a caderilor.

# Criterii de proiectare (2)

- **Criterii de cost**

— Costul dezvoltarii: costul dezvoltarii sistemului initial

— Costul tranzitiei la utilizatori (Deployment): costul instalarii sistemului si instruirii utilizatorilor

— Costul actualizarii (upgrade) sistemului: costul modificarii sistemului existent pentru a obtine sistemul actualizat – costul asigurarii compatibilitatii cu sistemul inlocuit (backward compatibility)

— Costul mentenantei: costul necesar pentru repararea defectelor in timpul operarii sistemului si efectuarea de imbunatatiri

— Costul administrarii: costul necesar administrarii sistemului

- Nu toate criteriile de cost pot deveni obiective de proiectare. Trebuie facute compromisuri intre diferite tipuri de costuri, cum ar fi: costul dezvoltarii <-> costul tranzitiei si costurile de mentenanta.

# Criterii de proiectare (3)

- **Criterii de mentenanţă -** determina dificultatea de a modifica sistemul dupa livrare.

— Extensibilitatea: usurinta de a adauga functionalitati sistemului

— Usurinta de modificare a sistemului

— Adaptabilitatea: usurinta de a adapta sistemul la diferite domenii de operare

— Portabilitatea:usurinta de a porta sistemul pe diferite platforme

— Claritatea codului (Readability): usurinta de a intelege sistemul prin citirea codului sursa

— Trasabilitatea cerintelor: usurinta de a face corespondenţa între codul sursa şi cerinţe specifice.

- **Criterii ale utilizatorului final**

— Utilitatea: cat de mult il ajuta sistemul pe utilizator in munca lui

— Usurinta de utilizare (Usability): cat de usor este pentru utilizator sa foloseasca sistemul

# Compromisuri în stabilirea obiectivelor de proiectare (1)

Numai un numar mic dintre criteriile de proiectare pot fi realizate simultan. De exemplu, fiabilitatea si siguranta in functionare sunt in contradictie cu un cost de dezvoltare redus.

- In definirea obiectivelor proiectarii trebuie facute anumite compromisuri. Exemple:

- **Memorie <–> viteză**:
  - daca se impun cerinte de timp de raspuns sau throughput atunci memoria trebuie sa fie mai mare pentru cresterea vitezei;
  - daca se impune spatiu de memorie mic, atunci datele pot fi comprimate in detrimentul vitezei de procesare.
- **Timp de livrare <–> funcţionalitate**: dacă timpul de livrare prevazut nu permite implementarea functionalitatii specificate, atunci sistemul poate fi livrat cu functionalitate incompletă, sau poate fi livrat mai tarziu cu functionalitatea completă.

# Compromisuri in stabilirea obiectivelor de proiectare (2)

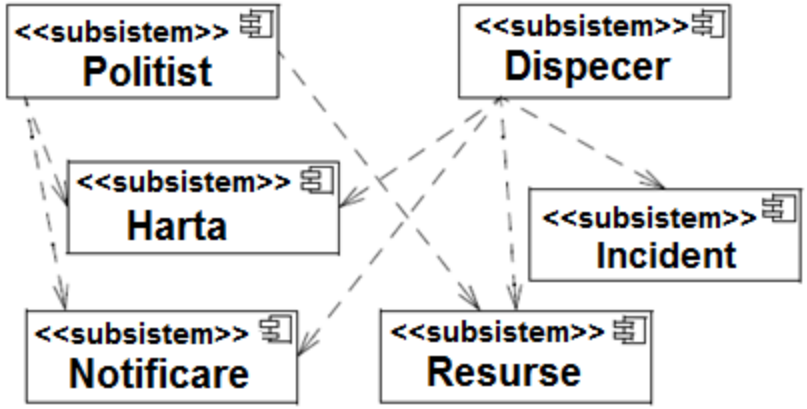
- **Timp de livrare <–> calitate**: daca sistemul nu poate fi testat suficient pentru a fi livrat la data planificata, sistemul poate fi livrat la timp dar cu defecte cunoscute, sau sistemul poate fi livrat mai tarziu dar cu mai putine defecte.
- **Timp de livrare <–> resurse umane**: daca dezvoltarea este în urma celei planificate, managerul de proiect poate aloca resurse suplimentare pentru a mari productivitatea. Dar, adaugarea de personal nou poate necesita instruire conducand la scaderea productivitatii. De asemenea, adaugarea de personal duce la cresterea costului dezvoltarii.

# Proiectarea arhitecturii software (1)

- Arhitectura software rezulta printr-un **proces iterativ** de **descompunere a cerintelor functionale si alocarea lor unor subsisteme ale viitorului sistem**.
- Scopul descompunerii: reducerea complexitatii (“divide et impera”).
- Un **subsistem:**
  - contine un set de clase corelate;
  - este o parte substituibila a unui sistem, cu interfete bine definite, care încapsulează comportamentul claselor pe care le conţine;
  - este dezvoltat de o singură persoană sau de o echipă de dezvoltare independenta.
- Descompunand sistemul în subsisteme relativ independente, echipe independente le pot dezvolta cu un minimum de comunicare.
- In cazul sistemelor complexe, descompunerea se aplica în mod recursiv, descompunând subsistemele în subsisteme mai simple.

# Proiectarea arhitecturii software (2)

## Descompunerea cerintelor functionale pe subsisteme

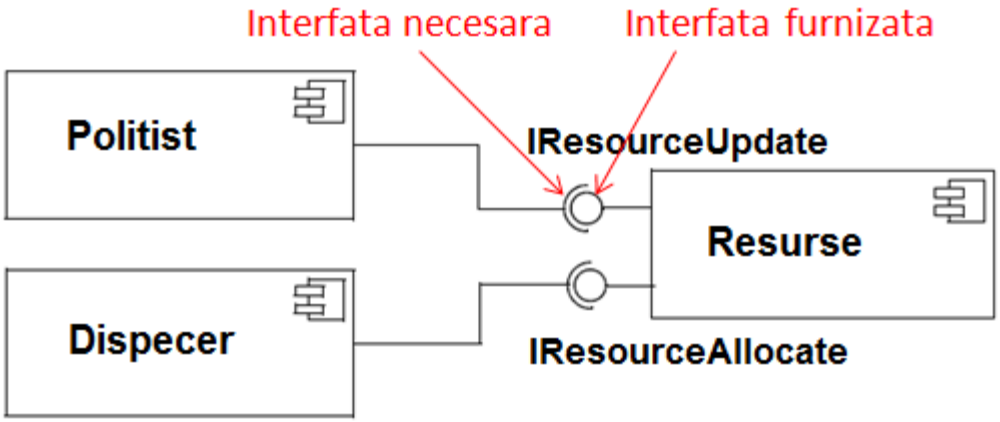



Intre subsistemele rezultate din descompunere se stabilesc relatii de dependenţă.

- Unele limbaje de programare, cum ar fi Java, furnizeaza constructii pentru modelarea subsistemelor (pachetele Java).
- In alte limbaje (C, C++), subsistemele nu sunt explicit modelate, dar ele pot fi grupate intr-un pachet UML.
- Indiferent daca limbajul ofera sau nu posibilitatea de modelare explicita a subsistemelor, acestea trebuie documentate cu grija deoarece sunt dezvoltate de echipe diferite.

# Servicii şi interfeţe de subsisteme

- Un subsistem se caracterizeaza prin **serviciile pe care le furnizeaza** altor subsisteme.
- **Un serviciu este un set de operaţii corelate care au un scop comun.**
- **Setul de operaţii furnizate de un subsistem formeaza interfaţa/interfeţele subsistemului**, conţinând: numele operatiilor, parametrii şi tipul parametrilor, valorile întoarse.
- **Proiectarea de sistem se focalizeaza pe definirea serviciilor pe care le furnizeaza fiecare subsistem.**



# Principii de descompunere în subsisteme

- **Subsistemele trebuie sa fie cat mai independente unul de altul**:
  - O modificare a unui subsistem trebuie sa aibă influenta minima asupra altor subsisteme.
  - O schimbare mica a cerintelor nu trebuie sa conduca la modificari majore ale arhitecturii software.
  - Efectul unei conditii de eroare trebuie sa fie izolat în subsistemul care a generat-o.
  - Un subsistem trebuie sa poata fi inteles ca o entitate de sine-statatoare.

- **Subsistemele trebuie sa "ascunda" continutul lor** (*information hiding*): se separa interfaţa/interfetele de implementare.

# Cuplare şi coeziune

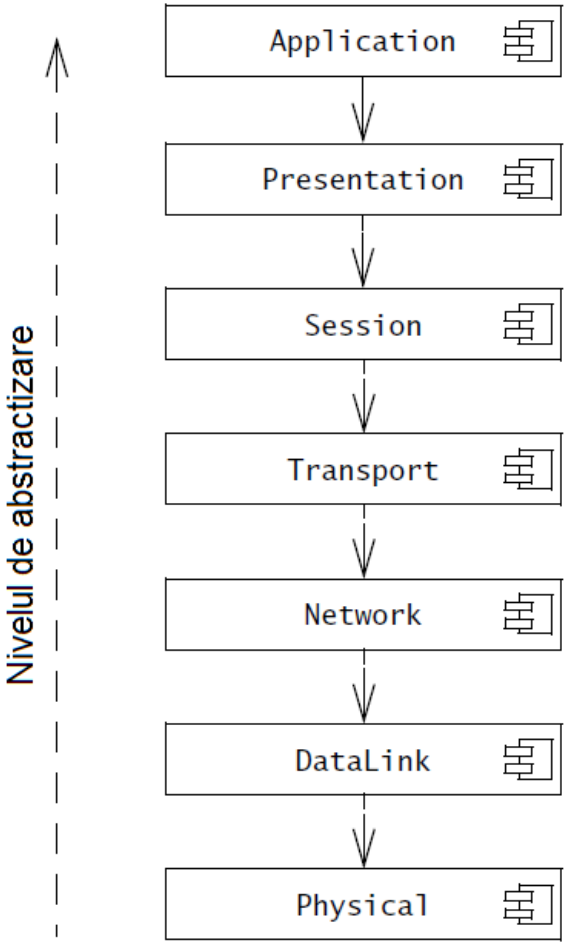
- **Cuplarea a doua subsisteme** este data de numarul de dependenţe dintre ele. Daca sunt slab cuplate ele sunt relative independente, modificarea unuia va avea impact redus asupra celuilalt.
  - In proiectarea de sistem se doreste ca subsistemele sa fie slab cuplate. Aceasta minimizează impactul pe care erorile sau schimbarile viitoare într-un subsistem îl are asupra celorlalte subsisteme
  - Reducerea cuplarii nu trebuie sa conduca la cresterea complexitatii prin introducerea de nivele suplimentare de abstractizare care consuma timp de dezvoltare si de procesare.
- **Coeziunea unui subsistem** este determinata de numarul de dependenţe dintre elementele sale.
  - O proprietate dorita a proiectarii de sistem este obtinerea de subsisteme cu coeziune mare.
  - Pentru a mari coeziunea unui subsistem se poate recurge la o descompunere a sa in alte subsisteme →poate conduce la cresterea cuplarii caci creste numarul de interfete.
- **In general trebuie facut un compromis între coeziune si cuplare**. O euristica buna este ca la fiecare nivel de abstractizare sa nu fie mai mult de 7+/- 2 subsisteme. De asemenea, numarul de niveluri nu trebuie sa fie mai mare de 7+/- 2. In general 3 niveluri de decompozitie sunt suficiente.

# Descompunere ierarhică

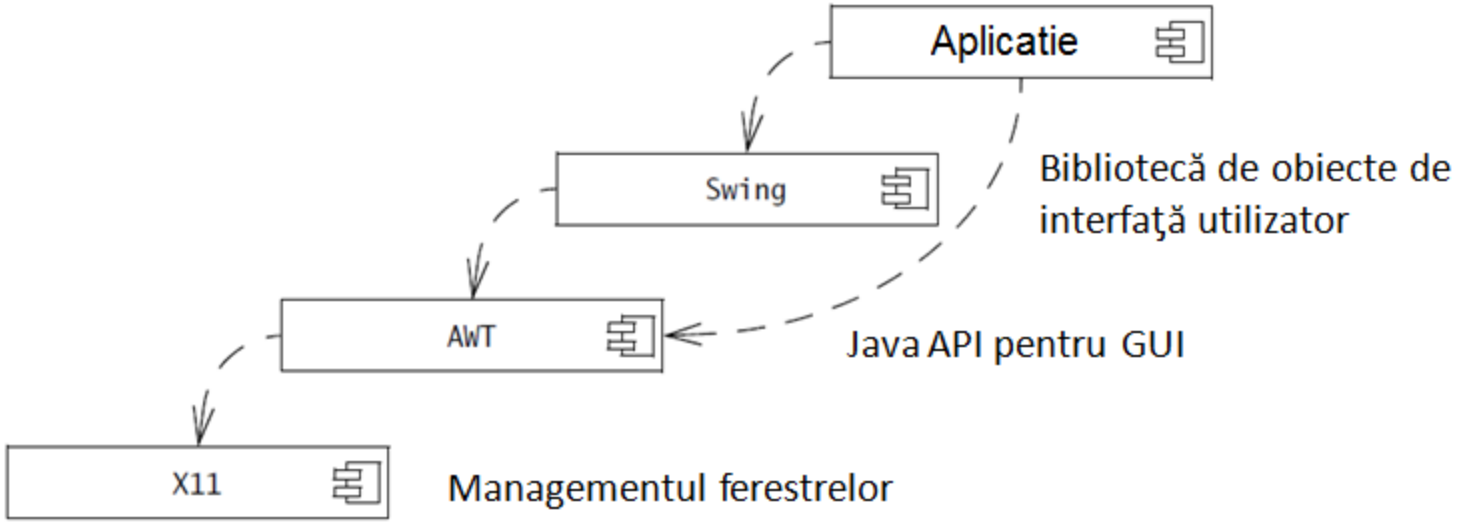
- O descompunere ierarhica a unui sistem produce un set ordonat de niveluri.
  - **Un nivel** este o grupare de subsisteme care furnizeaza servicii corelate, posibil realizate utilizand servicii din alt nivel.
  - Nivelurile sunt ordonate in sensul ca **fiecare nivel depinde numai de nivelurile de sub el** si nu are cunoastinta despre nivelurile de deasupra lui.
- Intr-o **arhitectură închisă**, fiecare nivel poate accesa numai nivelul aflat imediat sub el.
- Intr-o **arhitectură deschisă**, un nivel poate accesa si alte niveluri aflate sub el.
  - Un exemplu de arhitectură închisă este modelul OSI, care este compus din 7 niveluri. Fiecare nivel furnizeaza un set de servicii utilizand serviciile oferite numai de nivelul de sub el.

# Arhitectură ierarhică închisă

**Modelul OSI**
(Open Systems Interconnection)



# Arhitectură ierarhică deschisă




**Avantajul unei arhitecturi ierarhice inchise:**

-cuplare slaba intre subsisteme, subsistemele pot fi integrate si testate incremental

**Dezavantaje**:

-fiecare nivel introduce un overhead de viteza si memorie (scaderea vitezei si cresterea necesarului de memorie)

-adaugarea de noi functionalitati este dificila, daca nu a fost prevazuta

❖ **In mod uzual, descompunerea are 3-5 niveluri.**

# Partiţionare

- O alta abordare în reducerea complexitaţii constă în **partitionarea sistemului** in **subsisteme relativ independente, care apartin aceluiasi nivel de abstractizare**, fiecare subsistem fiind responsabil pentru o clasa diferita de servicii.
- **Fiecare subsistem depinde slab de celelalte dar poate opera adesea singur.**

Exemplu: un sistem “on-board” pentru un autovehicul este descompus in:

- subsistemul de conducere, care directioneaza in timp real soferul,
- subsistemul care ofera servicii legate de preferintele soferului (radio, pozitia scaunului),
- subsistemul de urmarire a consumului de combustibil si planificarea intretinerii.

**In general, descompunerea se face atat prin partitionare cat si ierarhic:**

➢ Mai intai se face descompunerea in subsisteme de nivel inalt, care ofera functionalitati specifice sau ruleaza pe noduri hardware specifice.

➢Fiecare subsistem este descompus, daca se justifica, in niveluri din ce in ce mai joase pana ce se obtin subsisteme suficient de simple pt a fi dezvoltate de o singura persoana.

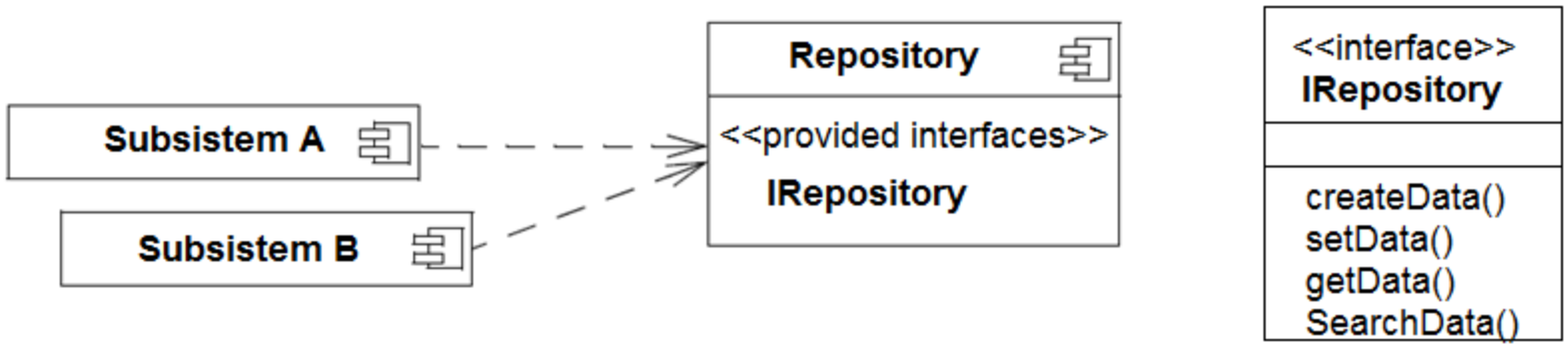
# Stiluri arhitecturale

❖ **Un stil arhitectural este un şablon pentru descompunerea in subsisteme.**

Exemple de stiluri arhitecturale:

- Repository
- Model - View - Controller (MVC)
- Client - Server
- Peer - to - peer
- Three - tier
- Four - tier
- Pipe and filter

# Arhitectura *Repository* (1)

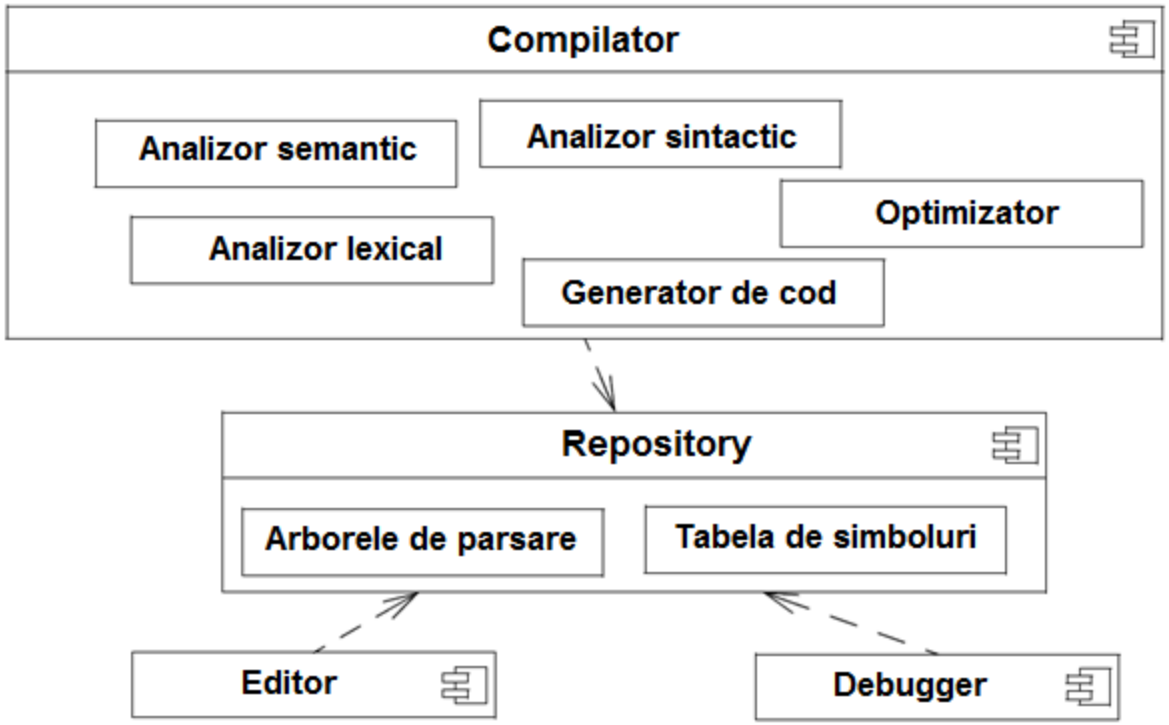



- Subsistemul Repository pastreaza o structura de date centrala, numita “repository” (depozit) si ofera servicii de creare, acces si modificare a datelor.
- Celelalte subsisteme sunt relativ independente între ele si interactioneaza numai prin Repository.
- Fluxul controlului poate fi dictat fie de repository fie de subsisteme (fluxuri de control independente si sincronizare pentru accesul la Repository)

❖ Este **arhitectura tipica pentru sistemele care folosesc baze de date**.

❖ Compilatoarele si sistemele de dezvoltare software folosesc, de asemenea, acest stil architectural.

# Arhitectura *Repository* (2)




- Componentele Compilator, Debugger si Editor sunt apelate in mod independent de utilizator.

- Componenta Repository asigura serializarea acceselor concurente.

- **Avantajul arhitecturii**: pot fi adaugate cu usurinta noi subsisteme care lucreaza cu Repository.
- **Dezavantaje:**

  - accesul la Repository poate conduce la o “strangulare”→ scaderea performantei;

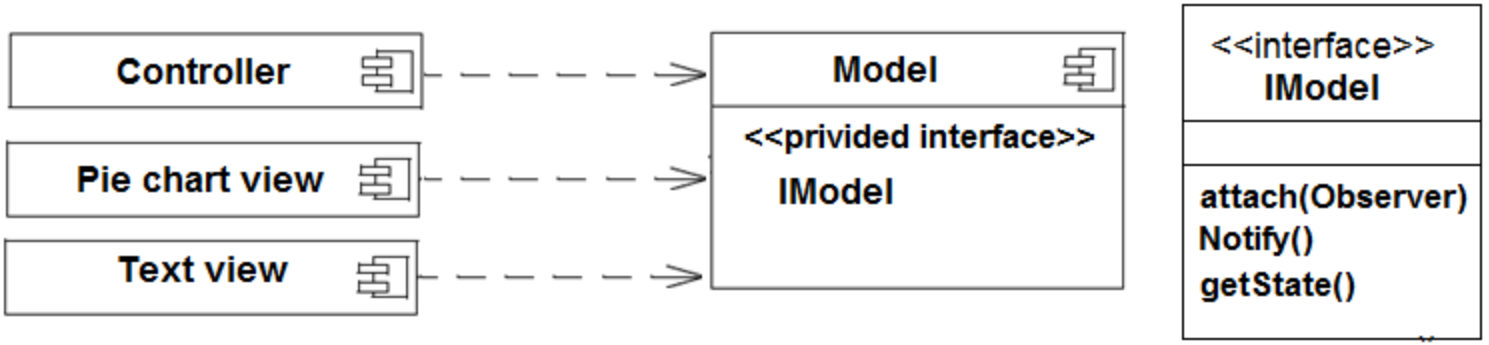
  - cuplarea intre Repository si fiecare alt subsistem este mare →modificarea Repsitory-ului afecteaza toate celelalte subsisteme.

# Arhitectura *Model-View-Controller* (1)

❖ O arhitectura MVC este alcatuita din 3 tipuri de subsisteme:

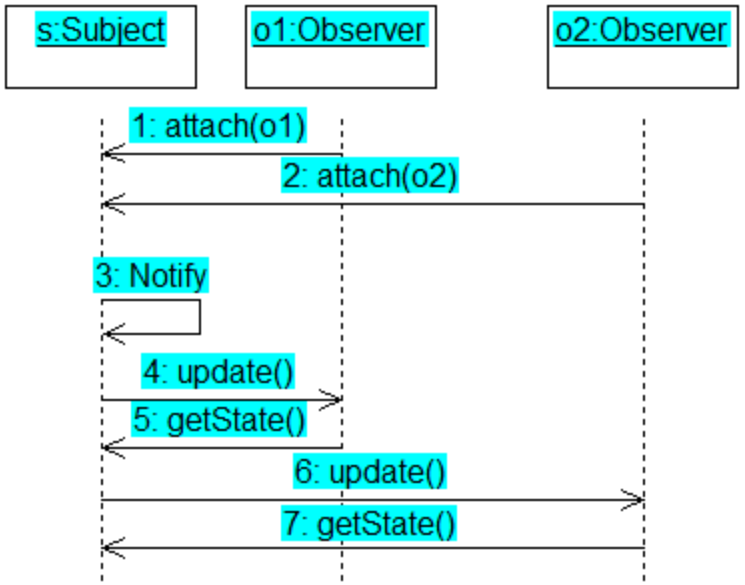
- Subsisteme **Model**: contin o reprezentare a datelor specifice aplicatiei, impreuna cu:
  - Operatiile de modificare a datelor conform logicii aplicatiei
  - Operatiile de acces la date
  - Operatiile de actualizare a datelor
  - Operatia de notificare a obiectelor **View**, la orice schimbare a datelor
- Subsisteme **View**: responsabile cu vizualizarea modelului
- Subsisteme **Controller**: gestioneaza interactiunea cu utilizatorul

❖ MVC este un caz particular de Repository, unde subsistemele **Model** implementeaza Repository (structura de date centrala) iar subsistemele **Controller** dicteaza fluxul controlului.



# Arhitectura *Model-View-Controller* (2)

- Subsistemele Model trebuie sa fie independente de subsistemele View si Controller.
- Subsistemele Controller trimit mesaje subsistemelor Model.
- Subsistemele **Model** notifica subsistemele **View** atunci cand se modifica modelul, printr-un protocol de tip “subscribe/notify”, implementat folosind sablonul “Observer”: un obiect **Model** este un **subiect** (memoreaza datele) iar obiectele **View** sunt **observatori**.




❖ Arhitectura MVC este adecvata sistemelor interactive, mai ales atunci cand sunt necesare mai multe vederi ale modelului.

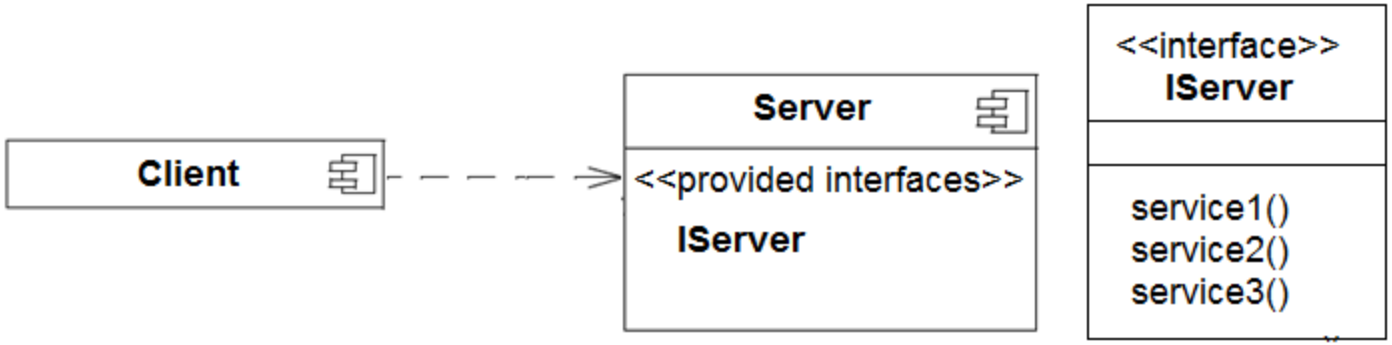
❖ Motivatia: interfeţele utilizator (View si Controller) sunt mult mai supuse schimbarilor decat modelul (datele specifice aplicatiei).

# Arhitectura *Client- Server* (1)

❖ Unul sau mai multe subsisteme numite **servere** furnizeaza servicii instantelor altor subsisteme, numite **clienti.**

- Clientii cunosc interfata serverului.
- Serverul nu trebuie sa cunoasca interfetele clientilor.

❖ Clientii sunt responsabili de iteractiunea cu utilizatorii.




- Cererea pentru un serviciu se face in mod uzual printr-un mechanism “remote procedure call” sau un obiect broker (ex. CORBA, Java RMI, HTTP).
- Fluxul controlului in client si in server sunt independente, exceptand sincronizarea pentru a raspunde la cereri si a primi rezultate.

# Arhitectura *Client- Server* (2)

- Este des folosita in sistemele informatice cu o baza de date centrala:

Clientii:

- primesc intrari de la utilizatori, efectueaza verificarea datelor introduse si apeleaza servicii ale serverului.

Serverul:

- raspunde la cererile clientului efectuand tranzatii cu baza de date;
- asigura integritatea si securitatea datelor.

- Un client poate accesa serviciile oferite de mai multe servere. Ex: un client poate accesa sute/mii de servere existente în Internet.

- Arhitecturile client-server sunt adecvate sistemelor distribuite care gestioneaza volume mari de date.

# Arhitectura *Client- Server* (3)

❖ **Criterii de proiectare urmarite in arhitecturile client-server**

- Portabilitate:
  - Serverul sa poata fi accesat din orice sistem de operare şi independent de mediul de comunicare
  - Clientul sa poata rula pe diverse dispozitive si sisteme de operare
- Transparenta locatiei – chiar daca serverul este distribuit
- Performanţă înaltă:
  - Client optimizat pentru taskuri de interactiune (actualizarea rapida a informatiei afisate, primita de la server)
  - Serverul optimixat pentru operatii CPU – intensive
- Scalabilitate: serverul sa poata trata un numar mare de clienti
- Fiabilitate

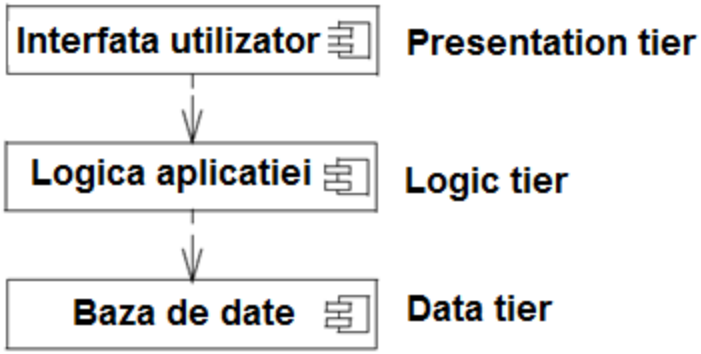
# Arhitectura *Peer-to-peer*

- Arhitectura peer-to-peer este o generalizare a arhitecturii client-server, in care subsistemele pot juca atat rolul de server cat si rolul de client.
- Fiecare subsistem poate cere si furniza servicii celorlalte: un peer poate fi atat server cat si client.
- Fluxul controlului in fiecare subsistem este independent, exceptand sincronizarile necesare pentru tratarea cererilor.
- Exemplu: un server de baza de date care accepta cereri de la aplicatii, dar totodata notifica aplicatiile atunci cand se fac modificari in baza de date.
- Arhitectura peer-to-peer este mai greu de proiectat caci poate introduce posibilitatea de deadlock si complica fluxul controlului la nivelul fiecarui subsistem (peer).

# Arhitectura *Three-tier (1)*

- **Arhitectura ierarhica in 3 niveluri, închisă**.
- Specifica sistemelor care includ o baza de date.
- Subsistemele sunt organizate în 3 niveluri ierarhice:

- *Nivelul interfaţă utilizator – Presentation tier* (interactiunea prin ferestre, form-uri, pagini web, s.a)
- *Nivelul aplicaţie (application logic ) – Logic tier*, numit si *Middleware* – include prelucrarile specifice aplicatiei si comunicarea dintre nivelul *interfaţă* şi nivelul *storage*
- *Nivelul stocare (storage) - Data tier* – sistemul de gestiune a bazei de date – asigura stocarea si regasirea obiectelor persistente.



# Arhitectura *Three-tier (2)*

- *Nivelul storage*, analog unui subsistem *Repository*, poate fi partajat de diferite aplicatii care utilizeaza aceeasi baza de date.
- Separarea nivelului *interfata* de nivelul *aplicatie* permite dezvoltarea (existenta) mai multor interfete utilizator pentru subsistemul/subsistemele care implementeaza logica aplicatiei.
- Cele 3 niveluri sunt de regula alocate pe noduri hardware distincte.
- Arhitectura este des folosita in sistemele bazate pe Web:
  - Web Browers implementeaza nivelul Interfata utilizator
  - Web server – trateaza cererile provenite de la web browser: implementeaza nivelul aplicatie
  - Baza de date – asigura gestiunea datelor persistente

Avantajul arhitecturii:

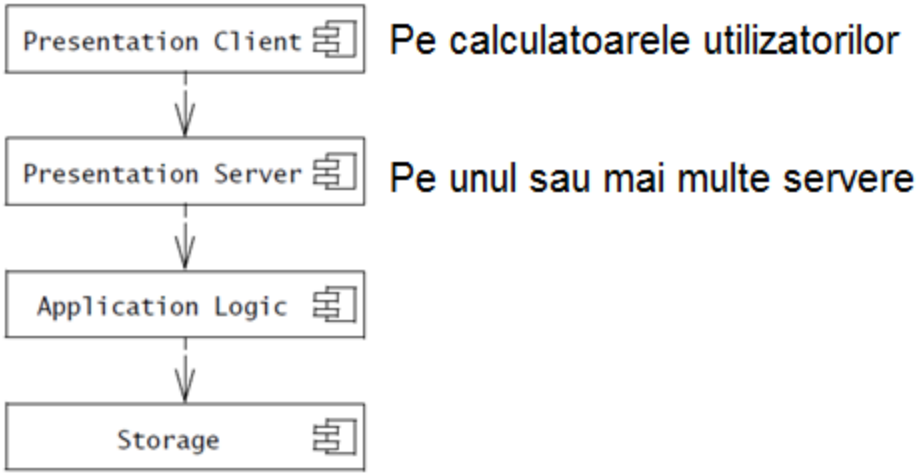
- Fiecare dintre cele 3 niveluri poate fi îmbunatatit sau înlocuit independent, în cazul unei schimbari a cerintelor sau a unei schimbari tehnologice.

# Arhitectura *Four-tier (1)*

- Este o arhitectura de tip Three-tier in care nivelul *Interfata utilizator* este descompus in:

  - nivelul *Prezentare Client* (Presentation Client Layer) si

  - nivelul *Prezentare Server* (Presentation Server Layer)

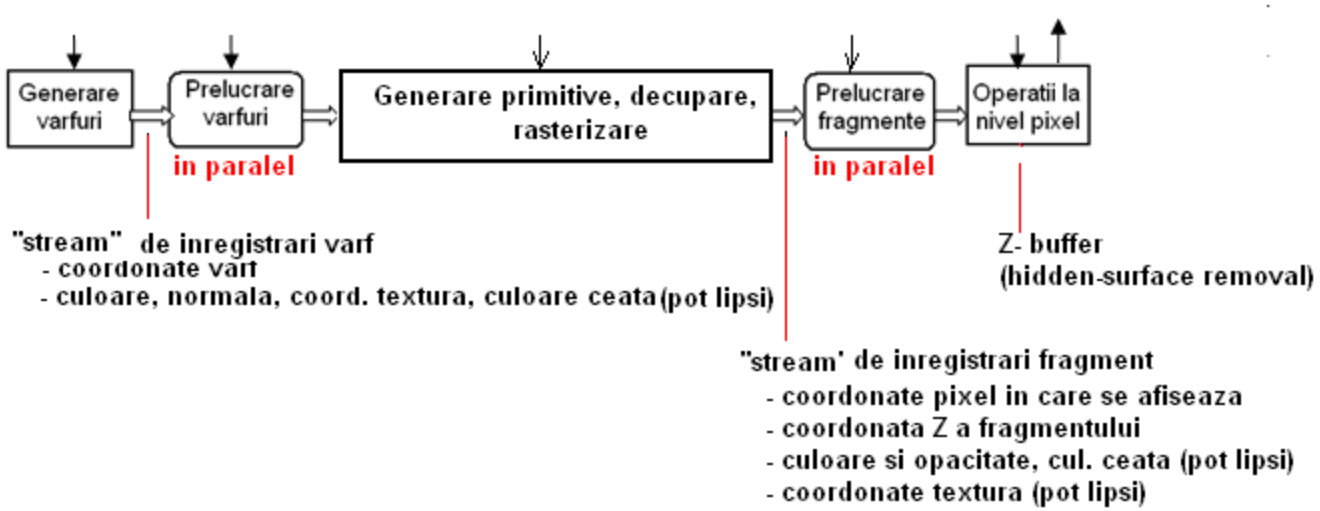



- **Avantajul**: nivelul *Presentation Client* poate oferi o gama larga de interfete utilizator care pot partaja elemente ale nivelului *Presentation server*.

# Arhitectura *Pipe and filter (1)*

- ❖ Filtru: un subsistem care efectueaza un pas de procesare
- ❖ Pipe: conexiunea între doi pasi de procesare
- ❖ Filtrele proceseaza fluxuri de date (data streams)

- Exemplu: Graphics pipeline!



# Exercitiu

1. Alegeti un stil arhitectural potrivit pentru aplicatia pe care o dezvoltati in cadrul proiectului. Justificati alegerea. Specificati scopul aplicatiei.
2. Care sunt principalele obiective de proiectare?

# Identificarea subsistemelor (1)

❖ Descompunerea initiala in subsisteme pleaca de la clasele identificate in modelul de analiza:

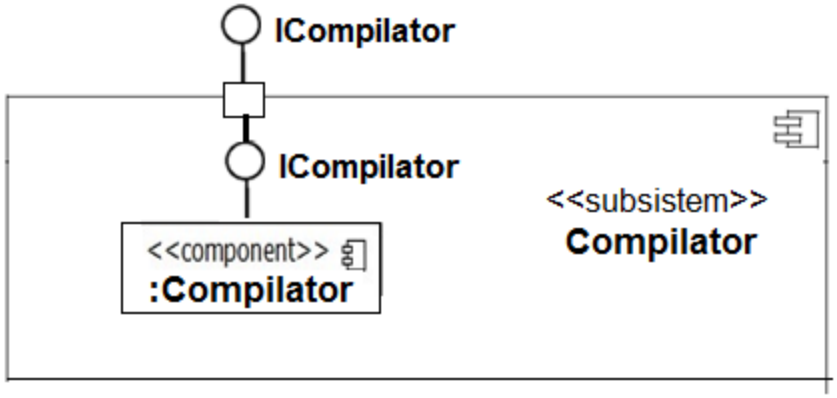
-se grupeaza in acelasi subsistem clasele corelate functional;

-numarul de asocieri dintre clase din subsisteme diferite trebuie sa fie cat mai mic.

❖ O alta euristica de identificare a subsistemelor:

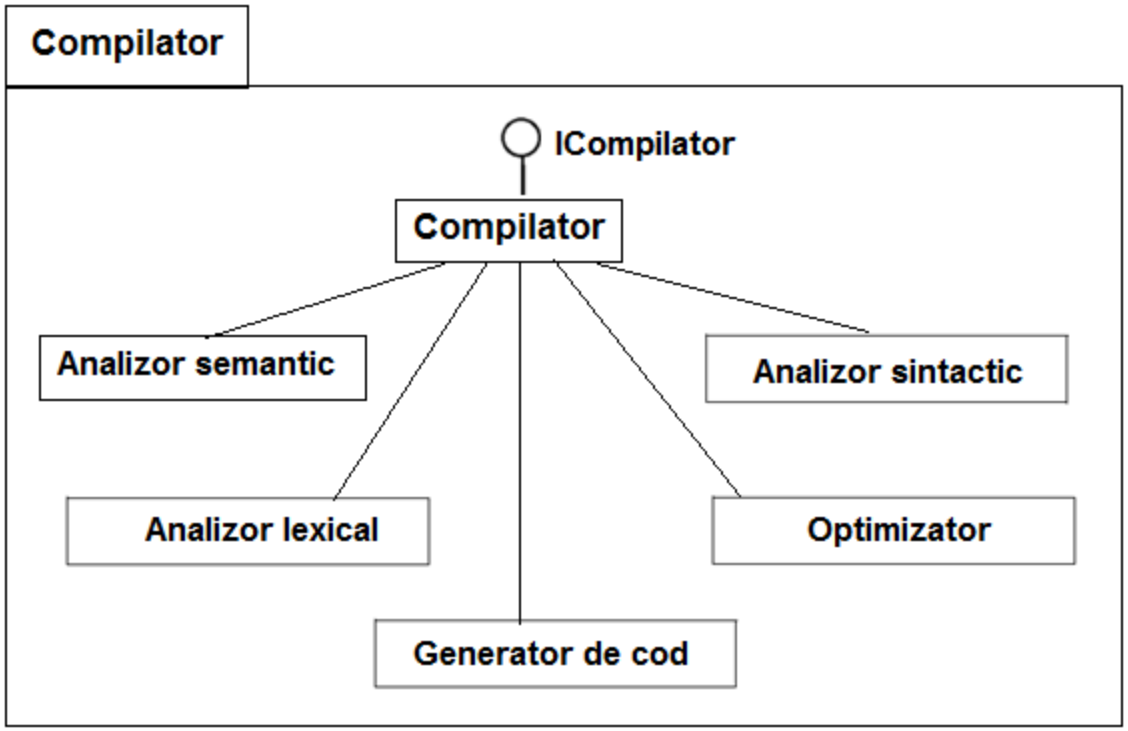
- se grupeaza într-un subsistem clasele obiectelor care participa într-un caz de utilizare.

❖ **Sablonul de proiectare Façade** ofera un mecanism adecvat pentru gruparea claselor in subsisteme şi reducerea cuplarii între subsisteme. Exemplu:




**Subsistemul Compilator** comunica prin interfata furnizata de o instanta a subcomponentei **Compilator.**

# Identificarea subsistemelor (2)




Gruparea claselor care alcatuiesc subsistemul Compilator, într-un pachet.

**Şablonul Façade**

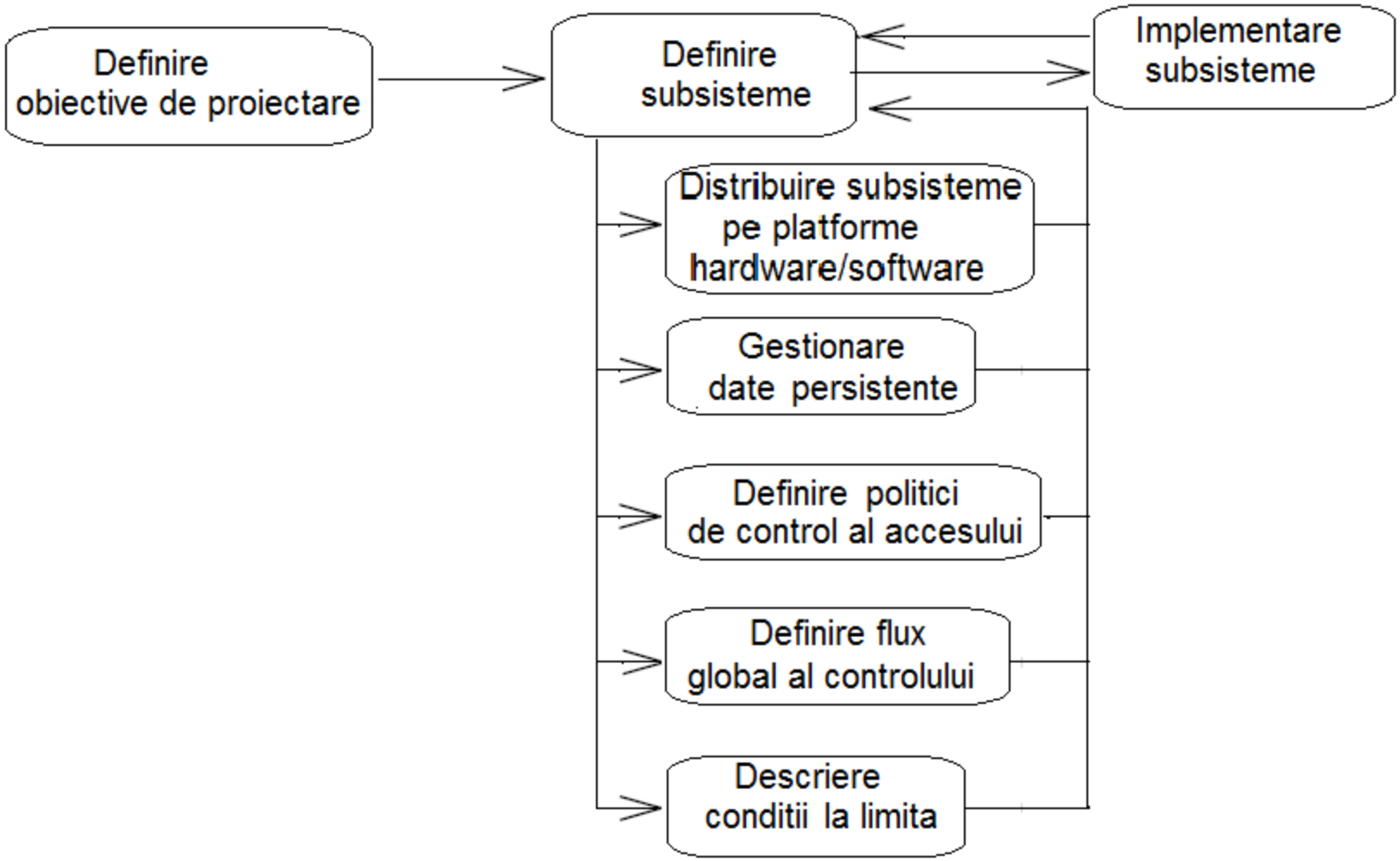
Clasele grupate in subsistemul **Compilator** comunica prin interfata **ICompilator** furnizata de clasa **Compilator.**

# Rafinarea descompunerii in sisteme

Descompunerea initiala este ajustata iterativ in timpul celorlalte activitati ale proiectarii de sistem, urmarindu-se realizarea obiectivelor de proiectare:

- Utilizarea de componente existente (“Off-the-shelf”): descompunerea initiala este ajustata in acest scop. Astfel de componente pot realiza servicii complexe mai economic decat daca ar fi dezvoltate. Exemple: pachete de interfata utilizator, sisteme de baze de date, s.a.
- Alocarea subsistemelor pe hardware: atunci cand un sistem este distribuit pe mai multe noduri pot sa apara necesare si alte subsisteme pentru rezolvarea unor aspecte de fiabilitate si performanta.
- Managementul datelor persistente: pot fi necesare unul sau mai multe subsisteme de management al datelor persistente.
- Politica de control al accesului utilizatorilor la resurse poate influenta distributia acestora in subsisteme.
- Fluxul global al controlului are impact asupra interfetelor subsistemelor.
- Conditiile limita: pornirea/oprirea sistemului, tratatea conditiilor speciale – pot adauga subsisteme

# Activitati în proiectarea de sistem



# Distribuirea subsistemelor pe platforme hardware/software (1)

- Multe sisteme se executa pe mai multe dispozitive de calcul si depind de accesul la Internet.
- Alocarea subsistemelor pe noduri hardware are consecinte importante asupra performantei si complexitatii sistemului: trebuie efectuata la inceputul proiectarii de sistem.
- Selectarea configuratiei hardware include si selectarea masinii virtuale pe care se va executa sistemul: sistemul de operare si alte componente necesare (sistemul de gestiune a bazei de date, sistemul de comunicare, s.a.)
- Selectia configuratiei hardware si a masinii virtuale poate fi restrictionata de client: hardware existent, considerente de cost.
- Alte criterii: anumite componente trebuie sa se execute in locatii specifice (de ex., software pentru un bancomat), trebuie achizitionate echipamente de la un anumit producator, trebuie asigurate anumite conditii de comunicare, etc.
- Pentru reprezentarea alocarii subsistemelor pe echipamente si platforme software se folosesc **diagrame UML de distributie**.

# Distribuirea subsistemelor pe platforme hardware/software (2)

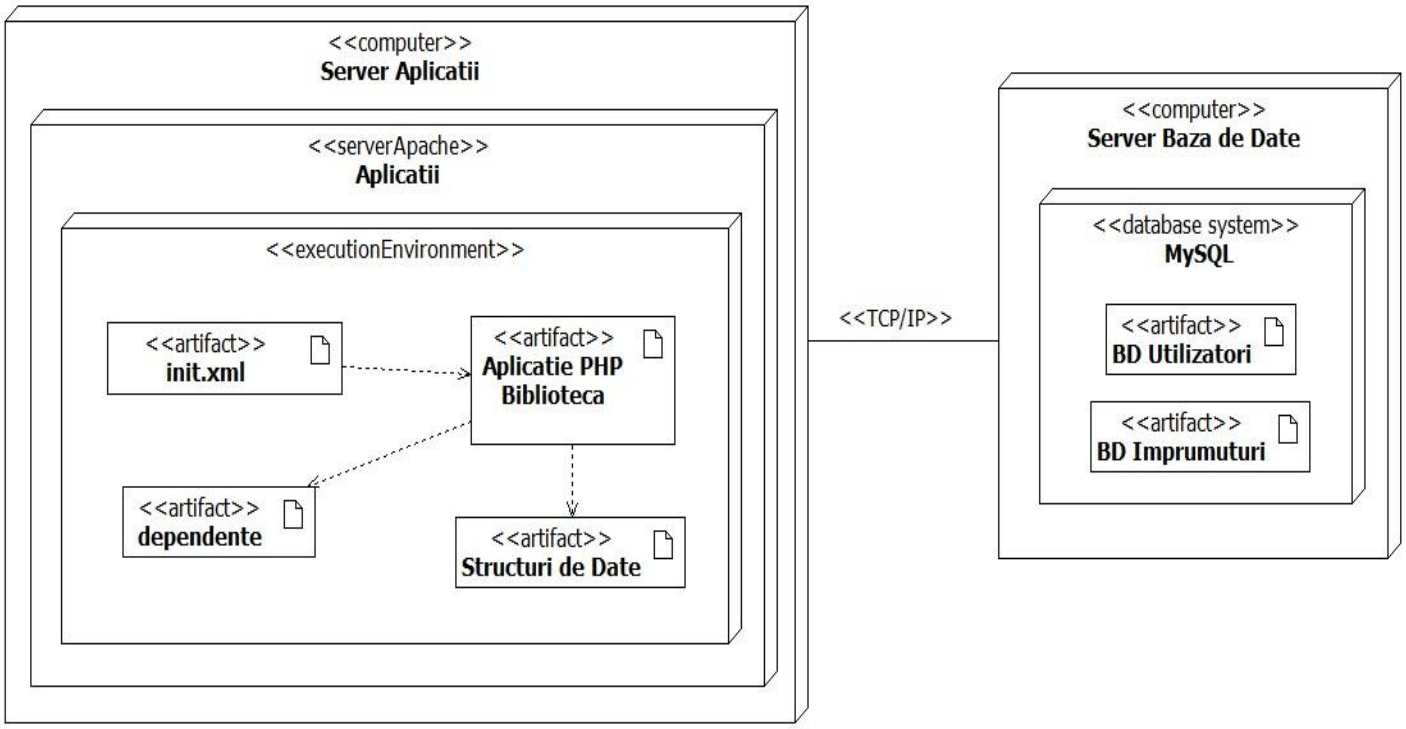



**Diagrama de distributie in UML 2.x.**

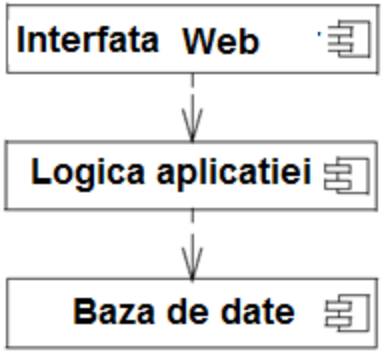
# Identificarea şi gestionarea datelor persistente

**Date persistente**:

- Date care nu se distrug la terminarea executiei aplicatiei care le-a creat
- Pot fi regasite si actualizate in cursul mai multor executii si posibil de mai multe aplicatii

**Mecanisme pentru asigurarea persistentei**:

- Sistem de fisiere: ieftin, simplu de implementat, gestiune de nivel coborat
- Baza de date: flexibil, scalabil, portabil, suporta scrieri/citiri concurente




Date utilizatori si carti

**Arhitectura SGCB** (Sistemul de gestiune a cartilor din mai multe biblioteci)

# Gestionarea datelor persistente

- **Intrebari care influenteaza proiectarea unei baze de date**
  - Cat de des este accesata baza de date?
  - Care este rata de cereri estimata?
  - Care este volumul tipic al datelor transferate la o cerere?
  - Trebuie sa fie o baza de date distribuita?
  - Datele trebuie sa fie arhivate?

- **Maparea modelului obiect UML pe o baza de date relationala:**
  - Datele se memoreaza in tabele alcatuite din mai multe randuri
  - Fiecare coloana a unei tabele reprezinta un atribut
  - Atributele unei clase corespund coloanelor unei tabele
  - Un rand din tabela corespunde valorilor atributelor unei instante a clasei
  - Asocierile dintre doua clase se implementeaza prin relatii intre tabele.

# Controlul accesului

- ❖ Intr-un sistem multi-utilizator fiecare tip de utilizatori (actor) are anumite drepturi de acces la resursele sistemului.
- Se determina obiectele partajate de actori.
- Se defineste mecanismul de control al accesului la obiectele partajate.
- In functie de cerintele de securitate se stabilesc regulile de autentificare a utilizatorilor si necesitatile de criptare a unor date.
- Accesul diferitilor actori la instantele claselor poate fi modelat prin **matricea de acces**:

| **Actor** | **Clasa A** | **Clasa B** | **Clasa C** |
|---|---|---|---|
| Actor 1 | Oper1A()<br>Oper2A() | | Oper1C()<br>Oper3C() |
| Actor 2 | Oper1A()<br>Oper2A()<br>Oper3A() | Oper1B()<br>Oper2B()<br>Oper3B() | |
| Actor 3 | Oper3A() | | Oper1C() |

Drepturile de acces

# Fluxul global al controlului (1)

**Fluxul controlului**: secventierea actiunilor la executia sistemului.

Sunt 3 mecanisme de control al fluxului operatiilor intr-un sistem:

— Dirijat procedural (procedure-driven control)

— Dirijat de evenimente (event driven control)

— Bazat pe fire de executie (thread-uri)

**Control dirijat procedural**

- Operatiile asteapta intrarile de la un actor atunci cand le sunt necesare.
- Este folosit in sistemele mai vechi si cele implementate in limbaje procedurale (cum sunt C, Pascal, Basic).
- Dificil de utilizat in limbajele orientate obiect, secventierea operatiilor fiind distribuita in seturi mari de obiecte.

# Fluxul global al controlului (2)

**Exemplu de flux al controlului dirijat procedural (Java)**: sistemul afiseaza mesaje si asteapta introducerea datelor de catre utilizator.

```
Stream in, out;
String userid, passwd;
out.println(“Login:”);
in.readln(userid);
out.println(“Password:”);
in.readln(passwd);
```

**Control dirijat de evenimente**

- Sistemul contine o bucla principala in care se asteapta un eveniment extern.
- Atunci cand se produce un eveniment, el este transferat obiectului corespunzator, pe baza informatiei asociate evenimentului.
- Este o structura de control simpla, care centralizeaza toate intrarile in bucla principala.
- Este uzuala in sistemele bazate pe evenimente generate de interfata grafica utilizator
- Nu este adecvata imlementarii secventelor in mai multi pasi.

# Fluxul global al controlului (3)

**Exemplu de flux al controlului dirijat de evenimente (Java):**

```
Iterator subscribers, eventStream;
Subscriber subscriber;
Event event;
EventStream eventStream;
while (eventStream.hasNext()) {
/* se extrage evenimentul din eventStream si se transmite obiectelor interesate in acel eveniment */
event = eventStream.next();
subscribers = dispatchInfo.getSubscribers(event);
while (subscribers.hasNext()) {
subscriber = subscribers.next()) {
subscriber.process(event);
}}
```

Obs: Urmatorul eveniment se extrage din eventStream numai dupa ce evenimentul curent a fost procesat de toate obiectele interesate.

# Fluxul global al controlului (4)

**Control bazat pe fire de executie**

- Pentru fiecare eveniment sistemul creaza un thread (evenimentele sunt tratate in paralel);
- Intr-un thread se pot astepta intrari de la un actor.

**Exemplu (Java):**

```
Thread thread;
Event event;
EventHandler eventHandler;
boolean done;

while (!done) {
event = eventStream.getNextEvent();
eventHandler = new EventHandler(event)
thread = new Thread(eventHandler);
thread.start(); /  * executa operatia run()   */
}
```

- Sistemele bazate pe fire de executie sunt mai greu de depanat si testat.

# Identificarea conditiilor limita

- In proiectare se considera, in principal, comportamentul sistemului in starea sa stabila.
- Este necesar sa se examineze si conditiile limita in functionarea sistemului: cum este pornit, initializat si oprit, cum se trateaza caderile majore care pot duce la pierderea datelor si intreruperile in comunicatia prin retea – caderi cauzate de un defect software sau de o intrerupere a sistemului electric.
- Cazurile de utilizare care descriu aceste situatii se numesc cazuri de utilizare limita (**boundary use cases)** sau **administrative use cases** – deoarece actorul in aceste cazuri de utilizare este de regula administratorul de sistem.

**Pornirea, oprirea si configurarea**

- Pentru fiecare componenta (ex. un server Web) se adauga 3 cazuri de utilizare: pornirea, oprirea si configurarea componentei.

**Tratarea exceptiilor**

- Pentru fiecare tip de cadere prevazuta se scrie un caz de utilizare care extinde unul dintre cazurile de utilizare descrise in faza de extragere a cerintelor. Tratarea acestor exceptii este decisa in faza de proiectare de detaliu.
- Anumite exceptii pot fi tolerate de sistem si incluse in proiectarea unor componente.

# Documentul de proiectare arhitecturala (1)
# - Arhitectural Design Document (ADD) -

**Cuprins**

**1. Introducere**

1.1 Scopul sistemului

1.2 Obiective de proiectare

1.3 Definitii, acronime, abrevieri

1.4 Referinte la alte documente

*(de ex. Doc de specificare a cerintelor, alte sisteme, etc.)*

**2. Arhitectura sistemului existent** *(daca noul sistem inlocuieste unul existent)*

sau **Arhitecturi ale unor sisteme similare**

# Documentul de proiectare arhitecturala (2)

**3. Arhitectura propusa**

3.1 Prezentare generala a arhitecturii sistemului

*(cu prezentarea sumara a functionalitatilor alocate fiecarui subsistem)*

3.2 Decompozitia in subsisteme si responsabilitatile fiecarui subsistem

3.3 Distributia subsistemelor pe echipamente hardware

3.4 Managementul datelor persistente

*(datele persistente, sistemul de baze de date folosit, schema conceptuala a bazei de date)*

3.5 Controlul accesului utilizatorilor la sistem

3.6 Fluxul global al controlului

3.7 Conditiile limita

Glosar de termeni