Proiectarea arhitecturala -1 (Proiectarea de sistem)

Prof. univ. dr. ing. Florica Moldoveanu

Curs *Ingineria programelor* – UPB, Automatică şi Calculatoare
2020-2021

Proiectarea arhitecturala

- **Modelul de analiza** ofera o vedere externa asupra sistemului, independenta de implementare. El serveste ca mijloc de comunicare intre dezvoltator şi client.
- **Proiectarea este etapa în care se construieste modelul fizic al sistemului**.
- Cerintele functionale specificate 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 într-un model al arhitecturii sistemului.**
- **Principalele activitati:**

  - Definirea obiectivelor proiectarii
  - Definirea subsistemelor software ale viitorului sistem
  - Alegerea strategiilor de construire a sistemului: strategii hardware/software, managementul datelor persistente, fluxul global al controlului, politici de control al accesului la sistem, tratarea conditiilor limita.

De la modelul de analiza → la modelul de proiectare

- Cerinte nefunctionale → Obiectivele proiectarii
- Modelul functional (cazuri de utilizare) + Modelul obiect (diagrame de clase) → Definire subsisteme, distributia subsistemelor pe hardware, managementul datelor persistente
- Modelul dinamic (diagrame de interactiune, de stari, de activitate) → Activitati concurente, fluxul global al controlului

Modelul de proiectare arhitecturala cuprinde:

1. **Obiectivele proiectarii.**
2. **Arhitectura software.** Descrie structurarea sistemului in subsisteme, responsabilitatile subsistemelor si dependențele dintre ele, alocarea subsistemelor pe hardware, fluxul controlului, controlul accesului la sistem si stocarea datelor persistente.
3. **Tratarea condiţiilor limită** la functionarea sistemului: configurarea sistemului, pornirea, initializarea, oprirea si tratarea exceptiilor.

Obiectivele proiectarii

- **Descriu calități ale sistemului sau restricții manageriale, care trebuie urmarite în toate perioadele urmatoare ale dezvoltării.**
- **Se decid analizând cerintele nefunctionale.**

- Exemple de **obiective de proiectare:**
  - **Îndeplinirea cerințelor de performanță**: timp de raspuns, throughput.
  - **Îndeplinirea cerințelor de calitate:** fiabilitatea, securitatea, siguranta în functionare, ş.a.
  - **Îndeplinirea cerințelor de operare cu sistemul**: comunicarea cu operatorii umani, usurinta de operare, usurinta de invatare a modului de operare, ş.a.
  - **Îndeplinirea cerințelor de mentenanță:** adaptabilitatea, extensibilitatea, portabilitatea
  - **Îndeplinirea cerințelor de planificare a proiectului**: termene, produse livrate, costuri.
- **Obiectivele de proiectare pot fi prioritizate, atunci cand nu toate pot fi îndeplinite, tinand cont de prioritizarea cerințelor nefunctionale.**

Criterii de proiectare (1)

Obiective de proiectare: cum pot fi îndeplinite? → Criterii de proiectare

- Se stabilesc criteriile de proiectare care pot asigura îndeplinirea obiectivelor.
- Pentru fiecare obiectiv, se stabilesc:
  - Criterii primare (esentiale), care trebuie aplicate pentru atingerea obiectivului
  - Criterii secundare (de dorit), care daca se aplica aduc un plus in realizarea obiectivului

❖ **Principalele criterii de proiectare** pot fi organizate in 5 grupuri:

- Criterii de performanţa
- Criterii de încredere in sistem
- Criterii de cost
- Criterii de mentenenţa
- Criterii ale utilizatorului final

Criterii de proiectare (2)

- **Criterii de performanţă**

— Asigurarea unui anumit timp de raspuns al sistemului la o cerere a utilizatorului

— Asigurarea unei anumite capacitati de procesare (throughput): numarul de taskuri realizate de sistem intr-o perioada de timp specificata.

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

Cadere = functionare neconforma cu specificatia.

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

— Robustetea (Robustness): capacitatea sistemului de a rezista (functiona) la intrari neprevazute.

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

— Toleranța la defecte (Fault tolerance): functionarea in conditii de cadere a unor componente.

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

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

Criterii de proiectare (3)

- **Criterii de mentenanţă – asigurarea unor calități ale sistemului de care depinde ușurinţa de a fi modificat după livrare.**

— Extensibilitatea: usurinta de a adauga functionalitati sistemului

— Usurinta de modificare a sistemului

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

— Portabilitatea: usurinta de a porta sistemul pe diferite platforme

— Claritatea codului (Readability): usurinta de intelegere a codului sursa

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

Extensibilitatea, usurinta de modificare, adaptabilitatea – se urmaresc la proiectarea arhitecturala a sistemului.

- **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

Criterii de proiectare (4)

- **Criterii de cost: asigurarea respectarii costurilor stabilite prin contractul cu clientul**

— 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, asigurandu-se compatibilitatea 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

Trebuie facute compromisuri intre diferite tipuri de costuri; de ex.: cost dezvoltare mic → cost mentenanta mare; daca se aloca mai mult timp dezvoltarii (cost mai mare), costul mentenantei poate fi mai mic.

Stabilirea si prioritizarea criteriilor de proiectare (1)

Intre criteriile de proiectare care pot contribui la realizarea obiectivelor de proiectare stabilite pot exista contradictii. Exemple:

Performanță <–> cost dezvoltare

Obiectivul prioritar	Criteriul de proiectare primar	Criteriul de proiectare secundar
Indeplinirea cerintelor de performanta	**Performanta** Se va achizitiona un hardware care poate asigura obtinerea performantelor cerute.	**Costul dezvoltarii** Costurile de achizitie prevazute pot fi depasite.
Indeplinirea cerințelor de planificare a proiectului	**Costul dezvoltarii** Se va achizitiona un hardware in limitele costurilor prevazute în contractul cu clientul.	**Performanta** Este posibil ca performanta ceruta sa nu poata fi realizata.

Stabilirea si prioritizarea criteriilor de proiectare (2)

Fiabilitatea <-> cost dezvoltare

Obiectivul prioritar	Criteriul de proiectare primar	Criteriul de proiectare secundar
Indeplinirea cerintelor de calitate ale sistemului	**Fiabilitatea** Se va urmari obtinerea unui sistem cu fiabilitatea ceruta, prin proiectarea unei arhitecturi uşor de testat si testarea de sistem până la obtinerea fiabilitatii cerute.	**Costul dezvoltarii** Poate creste prin depasirea termenului de livrare sau cresterea efortului de dezvoltare (alocarea de resurse umane suplimentare)
Indeplinirea cerințelor de planificare a proiectului: costul dezvoltarii	**Costul dezvoltarii** Testarea de sistem se va opri astfel încât sa nu se depaseasca costul dezvoltarii chiar daca nu s-a atins fiabilitatea ceruta.	**Fiabilitatea** Sistemul va fi livrat la timp sau la epuizarea resurselor financiare, dar cu defecte cunoscute.

Stabilirea si prioritizarea criteriilor de proiectare (3)

Cost dezvoltare <–> cost mentenanță

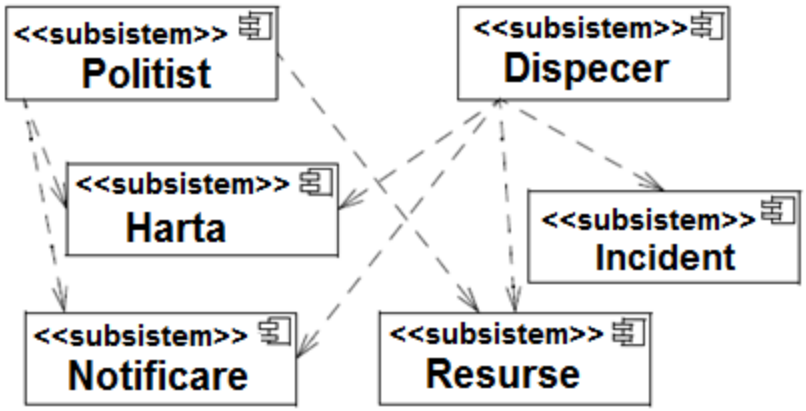
Obiectivul prioritar	Criteriul de proiectare primar	Criteriul de proiectare secundar
Dezvoltarea sistemului cu încadrarea cheltuielilor în costul de dezvoltare prevazut	**Costul de dezvoltare** Planificarea activitatilor de dezvoltare (proiectare, testare) cu un set de resurse umane, a.î. sa nu se depaseasca costul dezvoltarii prevazut in contractul cu clientul.	**Costul mentenantei** Prea putin timp alocat proiectarii sau testarea insuficientă pot creste costul mentenanței.
Sistemul dezvoltat sa fie usor de intretinut a.i. costul mentenantei sa nu-l depaseasca pe cel prevazut.	**Cost mentenanta** Se pune accent pe proiectarea unei arhitecturi usor de extins şi modificat. Se acorda suficient timp testării, a.î. produsul livrat sa contina cat mai putine defecte.	**Cost dezvoltare** Costul prevazut pentru dezvoltare poate fi depasit.

Proiectarea arhitecturii software (1)

- Arhitectura software este rezultatul unui **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, care realizeaza o functie/un set de functii.
  - Se doreste sa fie o parte substituibila a sistemului, cu interfețe bine definite, care încapsulează comportamentul claselor pe care le conţine.
  - Trebuie sa fie descris a.î. sa poata fi dezvoltat de o singură persoană sau de o echipă de dezvoltare, în mod independent.
- Descompunand sistemul în subsisteme relativ independente:
  - Echipe independente le pot dezvolta cu un minimum de comunicare
  - Mentenanta sistemului este mult uşurata.
- In cazul sistemelor complexe, descompunerea se aplica în mod recursiv, descompunând subsistemele în subsisteme mai simple.

Proiectarea arhitecturii software (2)

Reprezentarea arhitecturii software printr-o diagrama de componente UML

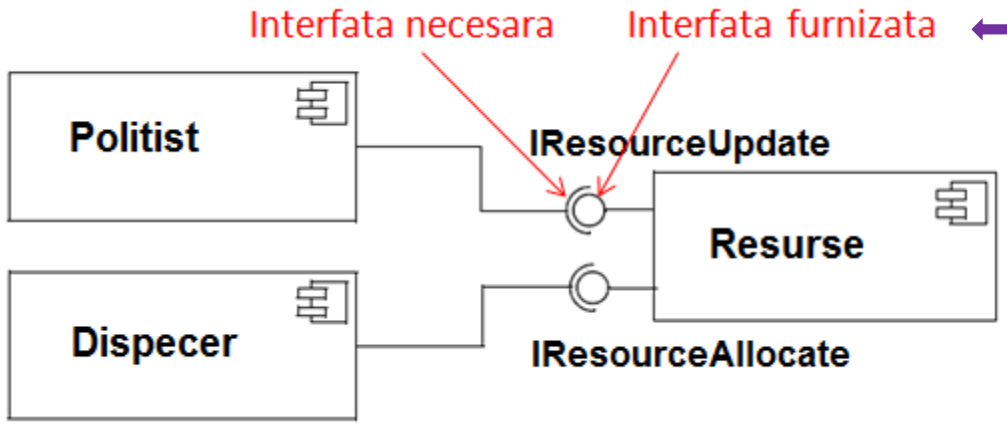


- Intre subsistemele rezultate din descompunerea cerintelor functionale se stabilesc relatii de dependenţă.
- Un subsistem A depinde de subsistemul B daca A foloseste operatii/servicii implementate de B.

- Unele limbaje de programare, cum ar fi Java, furnizeaza constructii pentru gruparea elementelor care fac parte dintr-un subsistem: Java package.
- In alte limbaje (C, C++), subsistemele nu sunt explicit modelate, dar elementele care fac parte dintr-un subsistem pot fi grupate intr-un pachet UML.
- Indiferent daca limbajul de implementare ofera sau nu posibilitatea de modelare explicita a subsistemelor, acestea trebuie documentate cu grija pentru a putea fi dezvoltate de echipe diferite.

Servicii şi interfeţe de subsisteme

- Relatiile de dependenţă dintre subsisteme se definesc prin intermediul interfeţelor subsistemelor.
- 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 servicii 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.**



Reprezentarea interfetelor în UML

Interfaces:
- provided
- required

Subsistemul Politist depinde de subsistemul Resurse deoarece utilizeaza interfata IResourceUpdate furnizata de subsistemul Resurse.

Principii si strategii 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 (sa afecteze cat mai putin relatiile dintre subsisteme).
  - Efectul unei conditii de cădere trebuie sa fie izolat în subsistemul care a generat-o.
  - Un subsistem trebuie sa poata fi înteles ca o entitate de sine-statatoare.
- **Subsistemele trebuie sa "ascunda" continutul lor** (*information hiding*): se separa interfaţa/interfetele de implementare.

❖ Strategii de descompunere in subsisteme:

- Descompunere ierarhica în niveluri de abstractizare (descompunere pe verticala)
- Partitionare la acelaşi nivel de abstractizare (descompunere pe orizontala)

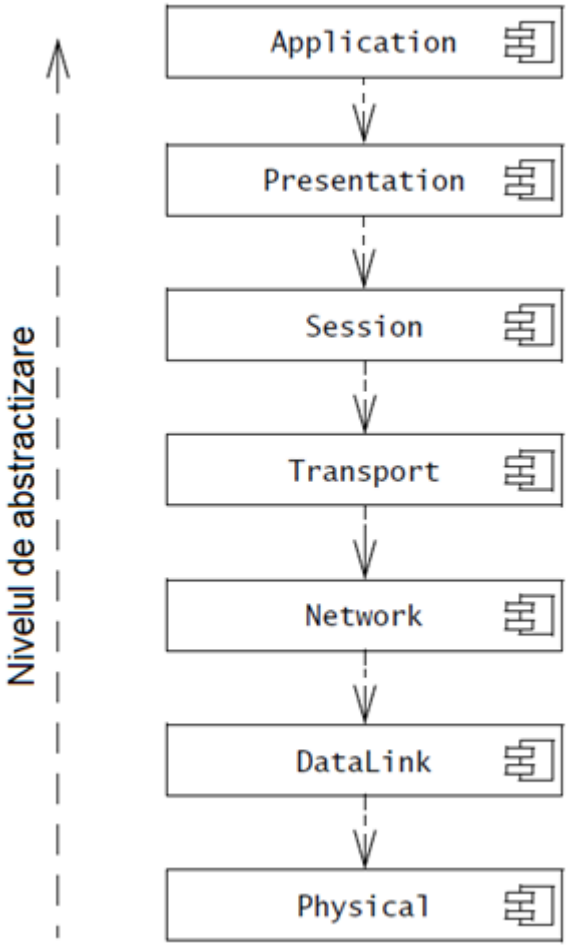
Descompunere ierarhică

- 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 cunostinta 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, pe care le implementeaza utilizand serviciile oferite numai de nivelul de sub el.

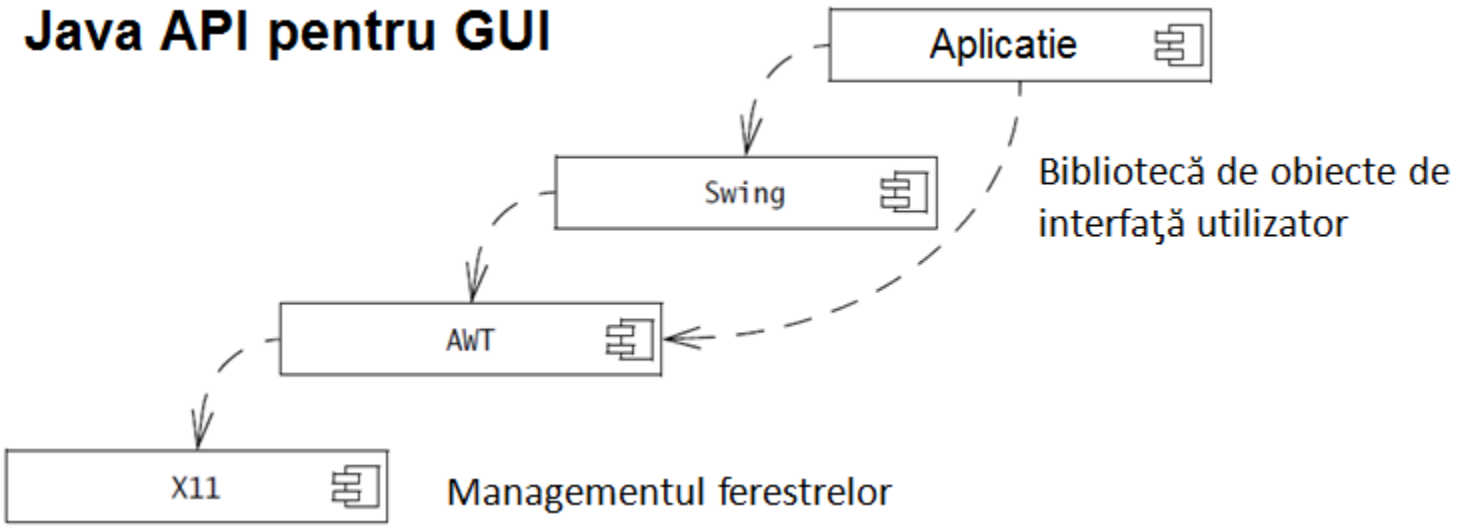
- **In mod uzual, descompunerea ierarhica are 3-5 niveluri.**

Arhitectură ierarhică închisă

Modelul OSI
(Open Systems Interconnection)



Arhitectură ierarhică deschisă



Avantajele unei arhitecturi ierarhice închise:

- cuplare slaba între 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 pe un nivel intermediar este dificila, daca nu a fost prevazuta.

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: sistemul de bord al unui autovehicul poate fi 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 o **partitionare in subsisteme de nivel inalt**, care ofera functionalitati specifice sau se execută pe noduri hardware diferite.

➢**Fiecare subsistem de nivel inalt este descompus ierarhic**, daca se justifica, in niveluri din ce in ce mai coborâte, până ce se obțin subsisteme suficient de simple pt a fi dezvoltate de o singura persoana/echipa, in mod independent.

Cuplare şi coeziune

- **Cuplarea a doua subsisteme** este data de numarul de dependenţe dintre ele. Daca sunt slab cuplate ele sunt relativ 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 caderile la executie sau schimbarile viitoare într-un subsistem îl are asupra celorlalte subsisteme.
- **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 interna mare**.
- Un subsistem cu o coeziune interna slaba poate fi descompus in alte subsisteme, mai coezive →poate conduce la cresterea cuplarii, caci creste numarul de interfeţe.
- **In general, trebuie facut un compromis între coeziune si cuplare**. O euristica buna este ca pe fiecare nivel de abstractizare sa nu fie mai mult de 7+/- 2 subsisteme. De asemenea, numarul de niveluri ierarhice sa fie de 7+/-2. In general 3 niveluri sunt suficiente.

Stiluri arhitecturale

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

Stiluri arhitecturale care vor fi tratate in cadrul cursului:

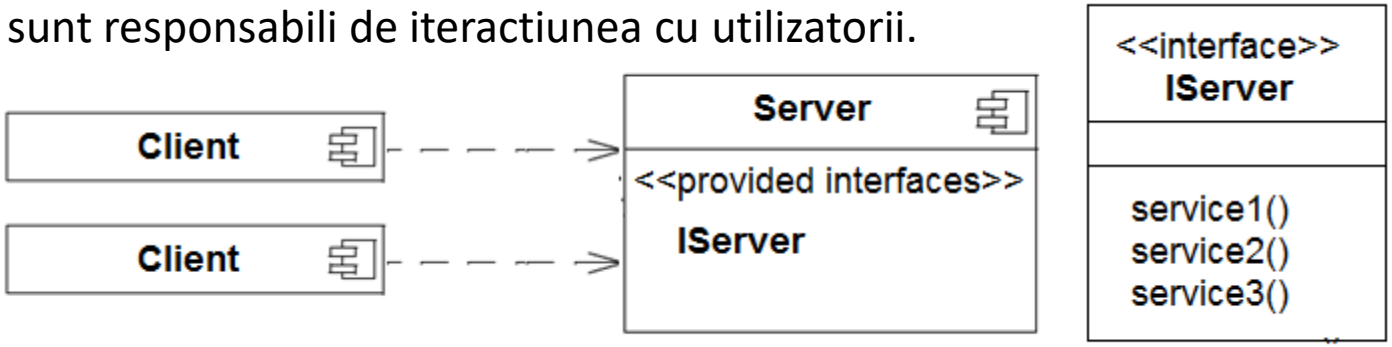
- Client - Server
- Peer - to – peer
- Arhitecturi ierarhice
  - -Three - tier
  - - Four - tier
- Arhitecturi centrate pe date
  - - Repository
- Arhitecturi centrate pe fluxul datelor
  - - Pipe and filter
- Arhitecturi orientate pe interactiune
  - - Model - View - Controller (MVC)
- Arhitecturi dirijate de evenimente

Arhitectura *Client- Server* (1)

❖ Unul sau mai multe subsisteme, care alcatuiesc **server-ul,** furnizeaza servicii 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 oferit de server se poate face printr-un mecanism “remote procedure call”, prin HTTP sau alte mecanisme.
- **La executie, clientul si serverul sunt procese independente** care ruleaza, de regula, pe calculatoare diferite, sincronizarea lor fiind necesara numai pentru a raspunde la cereri si a primi rezultate.

Arhitectura *Client- Server* (2)

❖ Este des folosita in sistemele cu o baza de date centrala: serverul gestioneaza BD

Clientii:

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

Serverul:

- raspunde la cererile clientului efectuand tranzactii cu baza de date
- asigura controlul accesului concurent la baza de date
- asigura integritatea si securitatea datelor
- asigura recuperarea datelor in cazul caderilor

❖ Arhitectura este adecvata sistemelor distribuite care gestioneaza volume mari de date.

Arhitectura *Client- Server* (3)

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

- Portabilitate:
  - Clientul sa poata rula pe diverse dispozitive si sisteme de operare
  - Serverul sa poata fi accesat din orice sistem de operare şi independent de mediul de comunicare
- Transparenta locatiei: clientul nu trebuie sa cunoasca locatia/locatiile serverului
- Performanţă înaltă:
  - Client optimizat pentru taskuri de interactiune (actualizarea rapida a informatiei afisate, primita de la server)
  - Serverul optimizat pentru operatii CPU – intensive/ operatii cu baze de date
- 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, în 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 (client), 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).

Arhitecturi centrate pe date: *Repository*

- Mai multe subsisteme comunica printr-un depozit de date, accesand si modificand datele.
- Subsistemele sunt relativ independente, ele comunicand numai prin depozitul de date.
- Scopul arhitecturii: **asigurarea integritatii datelor.**

Exemple: arhitectura unui sistem cu o baza de date accesata/modificata de mai multe subsisteme, o arhitectura web care mentine o structura de date ce poate fi accesata/modificata prin servicii bazate pe web.

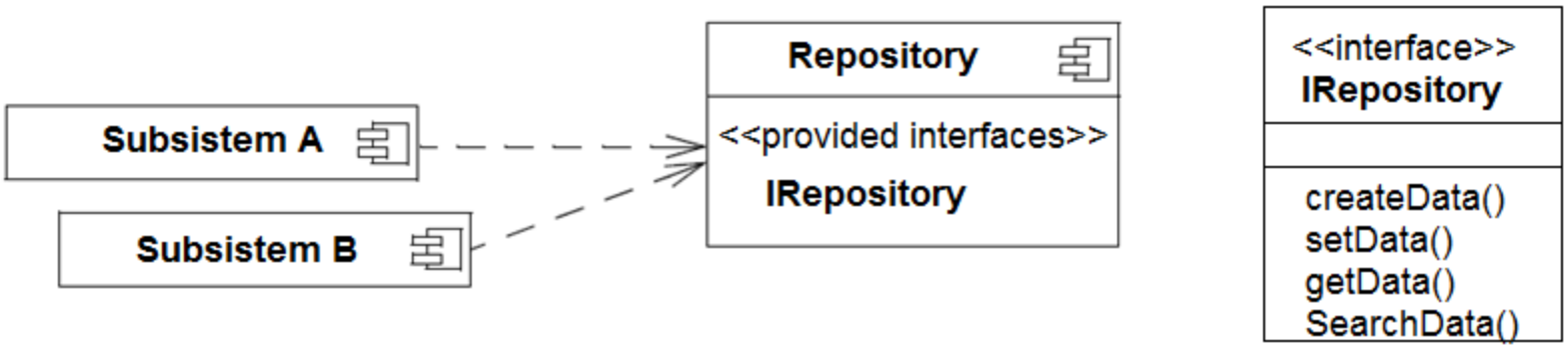
Arhitectura are 2 tipuri de subsisteme:

- Un subsistem Repository, responsabil cu persistenta datelor si accesul la depozitul de date.
- Mai multi „accesori” - subsisteme care comunica cu Repository pentru a accesa depozitul de date si a modifica datele din depozit; comunica intre ele numai prin Repository.

Fluxul controlului diferentiaza 2 tipuri de arhitecturi centrate pe date:

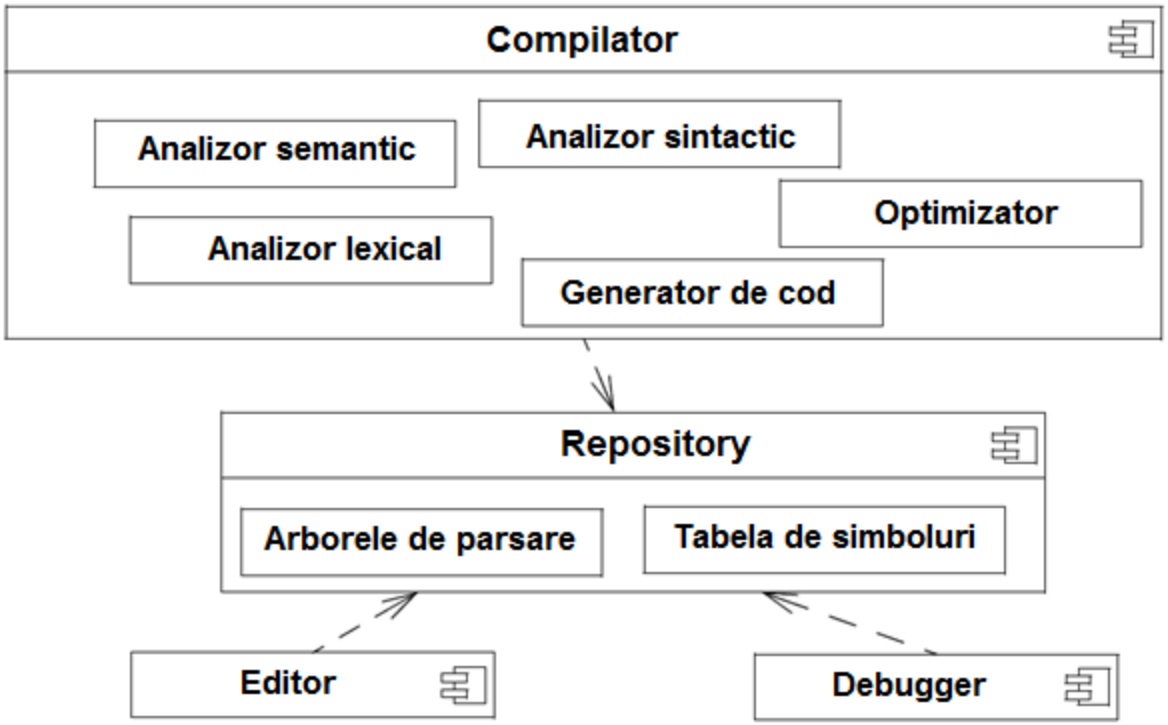
- Arhitecturi de tip “Repository pasiv”
- Arhitecturi de tip “Repository activ”.

Repository pasiv (1)



- Subsistemul Repository pastreaza o structura de date centrala, numita “repository” (depozit) si ofera servicii de creare, acces si modificare a datelor.
- Subsistemele accesor sunt relativ independente între ele si comunica numai prin Repository.
- Componenta Repository este pasiva iar accesorii sunt activi si controleaza fluxul prelucrarilor.
- Accesorii trimit cereri catre repository, ca de ex. „getData”, „setData”, etc.
- Procesarile atasate datelor sunt declansate de cererile accesorilor.
- Componenta Repository trebuie sa asigure serializarea accesurilor concurente.
- Este **arhitectura tipica pentru sistemele care folosesc baze de date, compilatoare si sisteme de dezvoltare software.**

Repository pasiv (2)



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

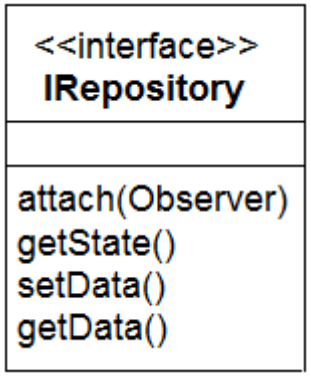
- Componenta Repository trebuie sa asigure serializarea accesurilor concurente.

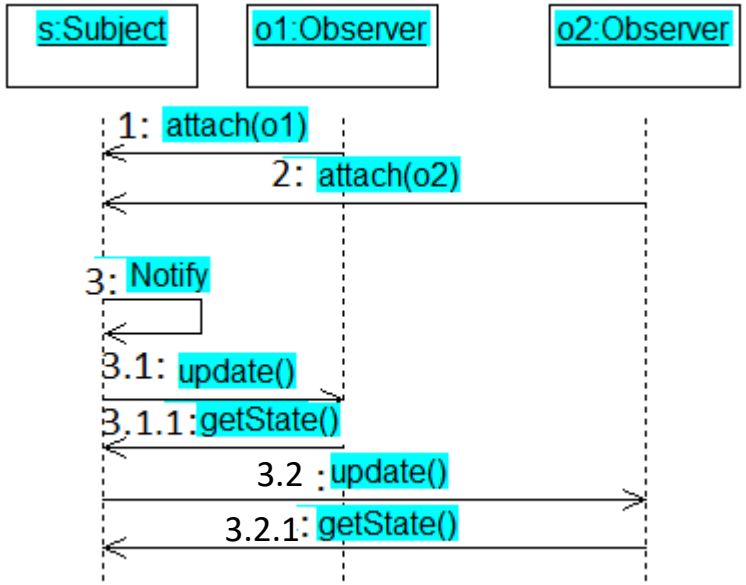
❖ **Avantajele arhitecturii**:
- Asigura integritatea datelor, operatii de backup si restaurare a datelor
- Asigura scalabilitate – pot exista oricâți accesori
- Reduce overhead-ul produs de transmisia datelor între componentele software

❖ **Dezavantaje:**
- Accesul la Repository poate conduce la o “strangulare”→ scaderea performantei;
- Cuplarea intre Repository si fiecare alt subsistem este mare →modificarea Repository-ului afecteaza toate celelalte subsisteme.

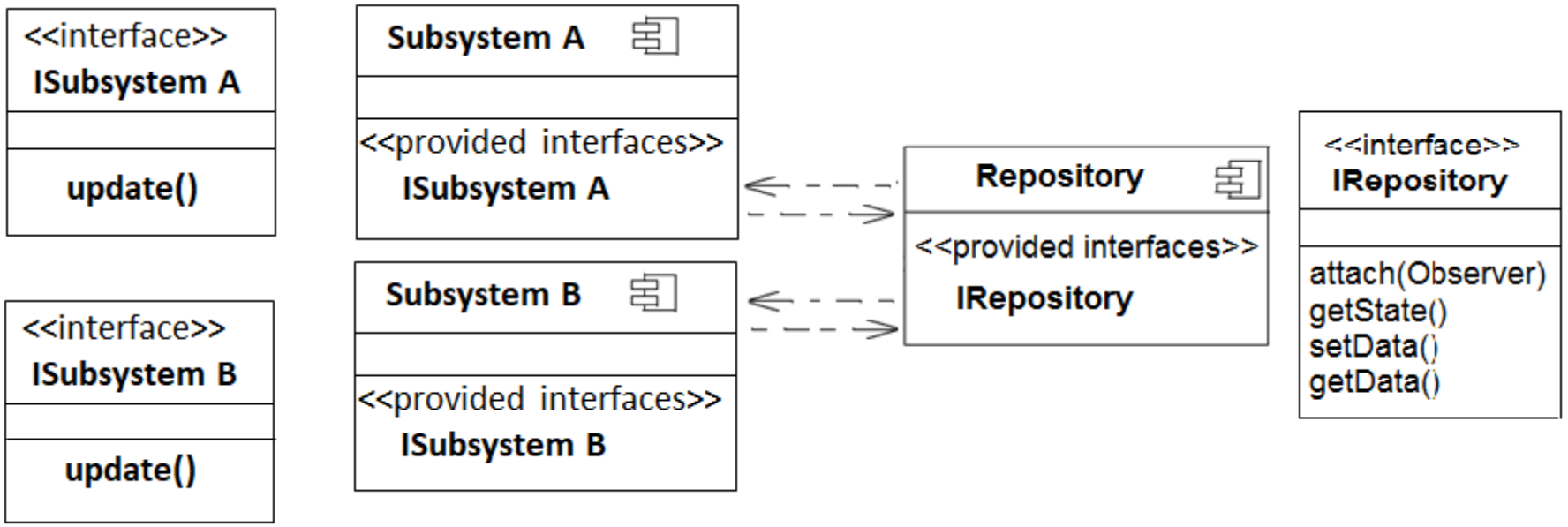
Repository activ (1)

- Componenta Repository este activă iar accesorii sunt pasivi.
- Fluxul prelucrarilor este determinat de starea curenta a depozitului de date.
- Componenta Repository notifica accesorii atunci cand datele sunt modificate.
- Prelucrarile in sistem sunt declansate de modificarea starii depozitului.
- Este o arhitectura de tip „subiect – observator” („subject – subscribers”), unde subiectul detine depozitul de date.





Repository activ (2)



Repository activ (3)

Avantajele arhitecturii:

- Accesorii se executa in paralel, fiind complet independenti intre ei
- Scalabilitate: pot fi adaugati oricati accesori
- Reutilizabilitatea codului accesorilor

Dezavantaje:

- Dependenta puternica intre componenta Repository si accesori: modificarile structurale ale depozitului pot afecta toti accesorii (operatia GetState())
- Probleme in sincronizarea accesorilor → dificultati in proiectare si testare