Analiza și proiectarea sistemelor software

Curs 8

PLAN CURS

Descrierea arhitecturii la execuție și a distribuirii

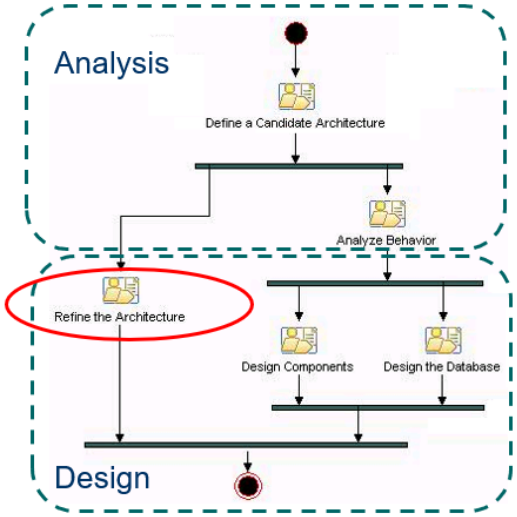
Proiectare clase (cont.)

Proiectare BD

Principiile practice ale proiectării software-lui

DESCRIEREA ARHITECTURII LA EXECUȚIE și a DISTRIBUIRII

- Analiză
  - Analiză arhitecturală (definire arhitectură candidat)
  - Analiza UC (analiză comportament)
- Proiectare
  - Identificare elemente de proiectare (rafinarea arhitecturii)
  - Identificare mecanisme de proiectare (rafinarea arhitecturii)
  - Proiectare clase (proiectare componente)
  - Proiectare subsisteme (proiectare componente)
  - Descrierea arhitecturii la execuţie şi a distribuirii (Rafinarea arhitecturii)
  - Proiectarea BD



Arhitecturi client/server

Alocarea proceselor la noduri

Consideraţii de proiectare

ARHITECTURI CLIENT/SERVER

- Client/server – modalitatea conceptuală de divizare a aplicaţiei în solicitanţi de servicii (clienţi) şi ofertanţi de servicii (servere).

- De obicei un client serveşte un singur utilizator şi gestionează serviciile de prezentare *end-user* din cadrul GUI.
    - Un sistem poate consta din mai multe tipuri de clienţi (ex staţii de lucru ale utilizatorilor, calculatoare din reţea, dispozitive mobile).
    - Serverul oferă de obicei servicii mai multor clienţi simultan. Aceste servicii sunt în mod tipic servicii de acces la baze de date, de securitate sau de imprimare.
    - Un sistem poate consta din mai multe tipuri de servere. De exemplu: *servere de baze de date* ce gestionează motoare de baze de date (ex. Oracle, DB2); *servere de imprimare*, gestionând logica driver-ului (ex. cum ar fi gestiunea cozii de aşteptare pentru o anumită imprimantă); *servere de comunicare* (ex.TCP/IP, ISDN, X.25); *servere pentru gestionarea ferestrelor* (ex. X); ***servere de fişiere*** (ex. NFS).

- Logica aplicaţiei şi în particular logica business sunt distribuite prin ***partiţionarea aplicaţiei între client şi server.***

ARHITECTURI CLIENT/SERVER

Aplicaţiile tipice includ: servicii client, servicii business, servicii de date.

Tipuri de arhitecturi funcţie de alocarea serviciilor la nodurile de procesare:

- **Arhitectură pe două straturi** (“Fat client”) : Cea mai mare parte a funcţionalităţii sistemului (serviciile client şi cele business) se execută la client.

- **Arhitectură în trepte**: Sistemul este divizat în trei părţi logice: servicii *client*, servicii *business*, servicii de *date.*

Părţile logice pot fi distribuite fizic pe trei sau mai multe noduri fizice.

Serviciile *client*, responsabile în principal cu elemente de prezentare şi interacţiune prin GUI, tind să se execute pe staţii de lucru sau pe dispozitive mobile.

Serviciile de *date* tind să fie implementate utilizând tehnologii din categoria server de baze de date, care se execută în mod normal pe unul sau mai multe noduri de performanţă ridicată şi cu lărgime mare de bandă, care servesc sute sau mii de utilizatori conectaţi într-o reţea.

Serviciile *business* sunt utilizate în mod tipic de mai mulţi utilizatori în comun, astfel că acestea tind să fie localizate, de asemenea, pe servere specializate, deşi ele pot fi amplasate şi pe aceleaşi noduri cu serviciile de date.

ARHITECTURI CLIENT/SERVER

Aplicaţiile tipice includ: servicii client, servicii business, servicii de date.

Tipuri de arhitecturi funcţie de alocarea serviciilor la nodurile de procesare:

- **Arhitectură pe două straturi** (“Fat client”) : =
- **Arhitectură în trei trepte**: Sistemul este divizat în trei părţi logice: servicii client, servicii business, servicii de date.

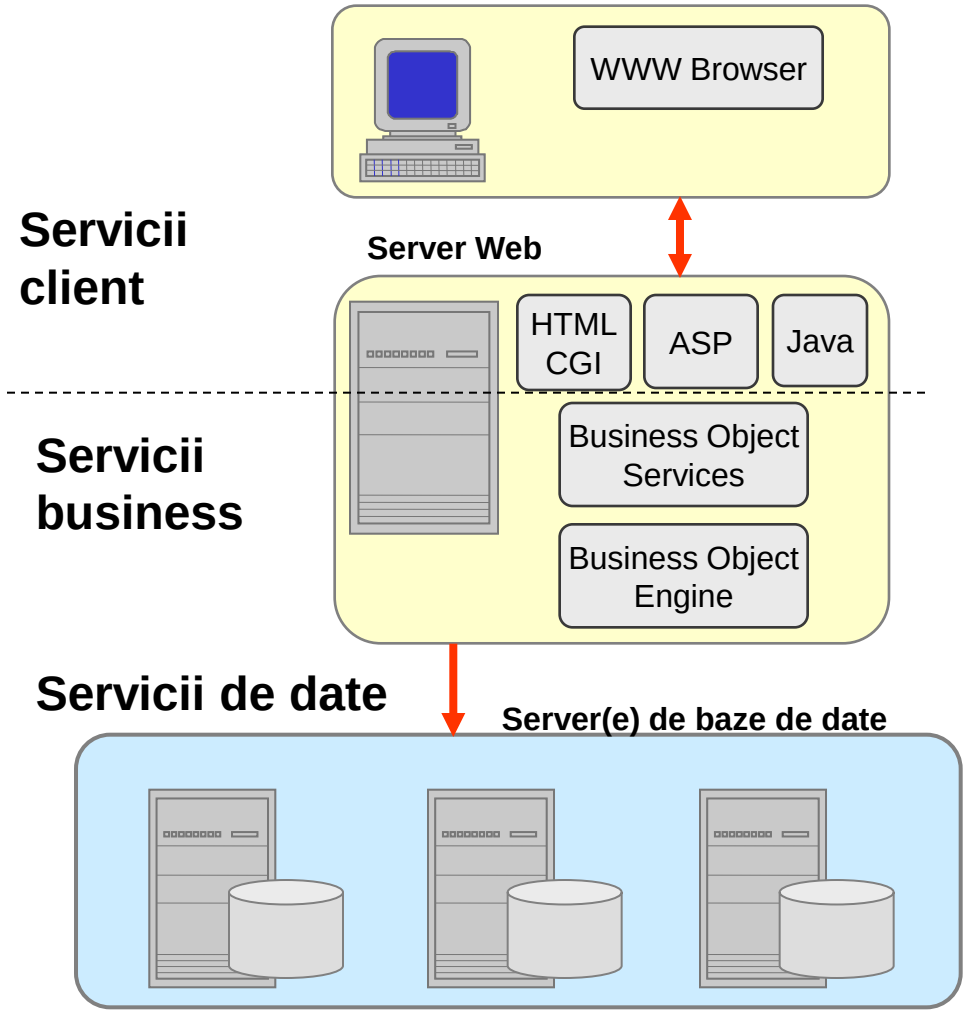
Acest mod de partiţionare a funcţionalităţii oferă un model de arhitectură relativ fiabil pentru *promovarea scalabilităţii*: prin *adăugare de servere* şi *reechilibrarea procesărilor* între serverele business şi între serverele de date se obţine un grad superior de scalabilitate.

- **Arhitectura de tip aplicaţie Web** poate fi caracterizată ca şi client “anorexic”. Deoarece clientul este un Web browser, serviciile client propriu-zise sunt de dimensiuni reduse. Aproape întreaga activitate are loc pe unul sau mai multe servere Web şi servere de date.

Aplicaţiile Web sunt *uşor de distribuit şi de modificat*. Ele sunt relativ ieftin de dezvoltat şi de întreţinut (deoarece cea mai mare parte a infrastructurii aplicaţiei este oferită de browser şi de serverul web). Totuşi, ele ar putea *să nu ofere gradul dorit de control asupra aplicaţiei* şi, de asemenea, tind *să satureze repede reteaua* dacă nu sunt bine proiectate (uneori chiar şi dacă sunt bine proiectate).

ARHITECTURI CLIENT/SERVER

- Exemplu de arhitectură de aplicaţie Web



ALOCARE PROCESE LA NODURI

Pentru distribuirea sistemului procesele trebuie asignate la dispozitive fizice și/sau virtuale.

CRITERII:

legate de cerințe

- Respectarea modelului arhitecturii client/server.
- Timpul de răspuns şi volumul informaţiilor prelucrate: procesele cu cerinţe de timp de răspuns rapid trebuie alocate celor mai rapide procesoare.
- Minimizarea traficului în reţea: procesele ce interacţionează puternic vor fi alocate pe acelaşi nod.
- Cerinţe de re-rutare (pentru redundanţă şi toleranţă la defecte).

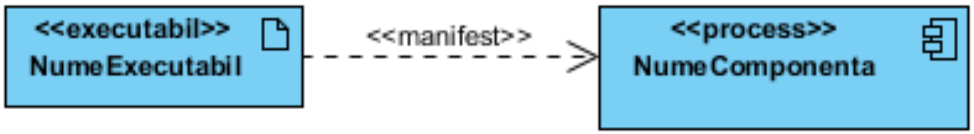
legate de resurse

- Capacitatea nodului (în termeni de capacitate de memorie şi putere de procesare).
- Lărgimea de bandă medie pentru realizarea comunicării (bus, LANs, WANs).
- Disponibilitatea de hardware şi de legături de comunicare.

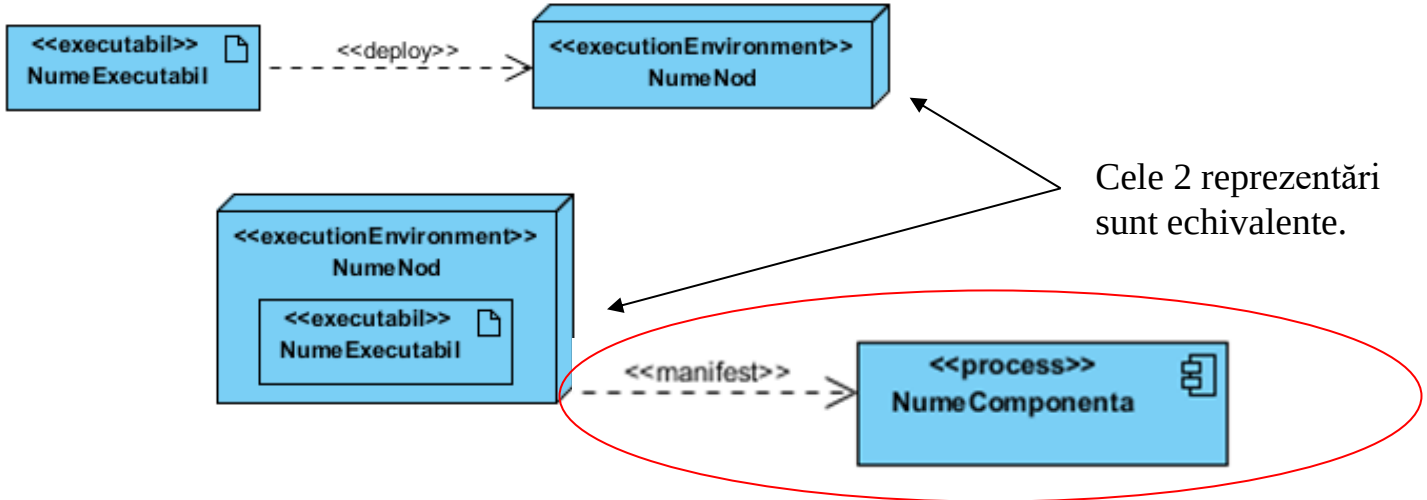
MODELAREA ALOCĂRII PROCESELOR LA NODURI

Procesele pot fi reprezentate cu componente stereotipate cu <<*process*>>.

Componentele sunt generate la execuție (runtime) din fișiere executabile reprezentate ca artefact stereotipat cu <<*executabil*>> (sau <<*artifact*>> sau <<*deployment spec.*>> , etc).

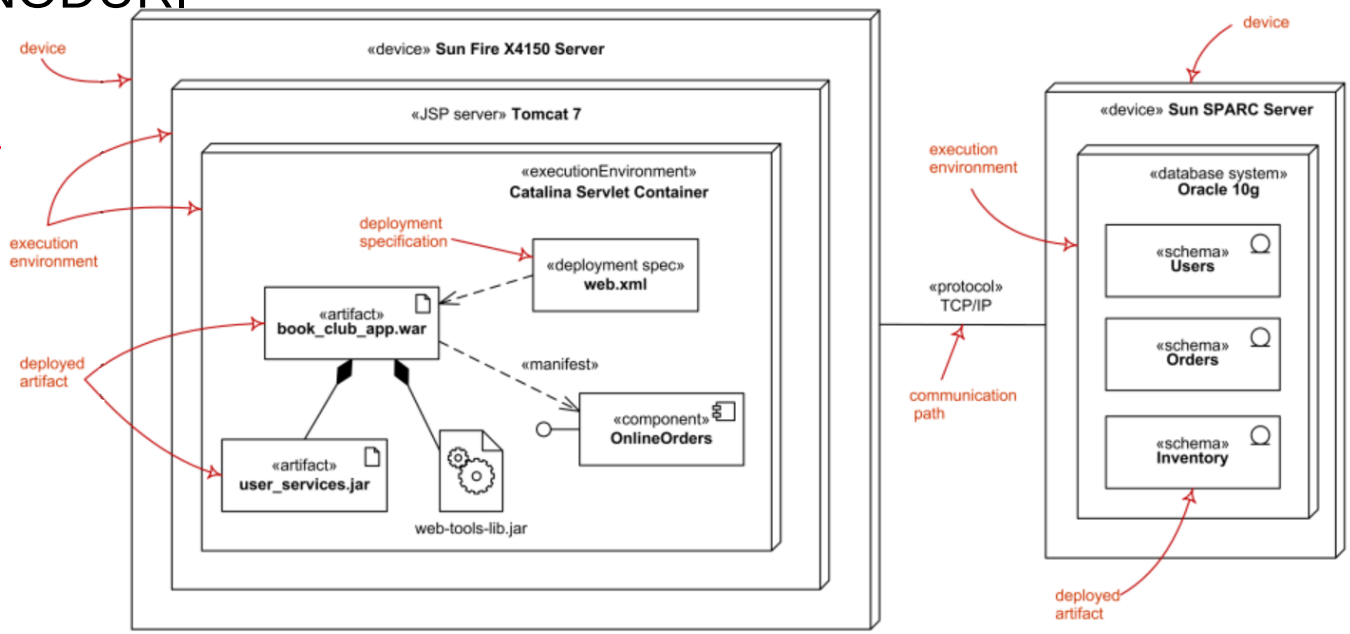


Executabilele vor fi instalate (deployed) pe nodurile de procesare.



MODELAREA ALOCĂRII PROCESELOR LA NODURI

https://www.uml-diagrams.org/deployment-diagrams-overview.html



- Diagramele de instalare (Deployment diagrams) permit capturarea topologiei nodurilor sistemului, incluzând şi asignarea la acestea a elementelor de execuţie.

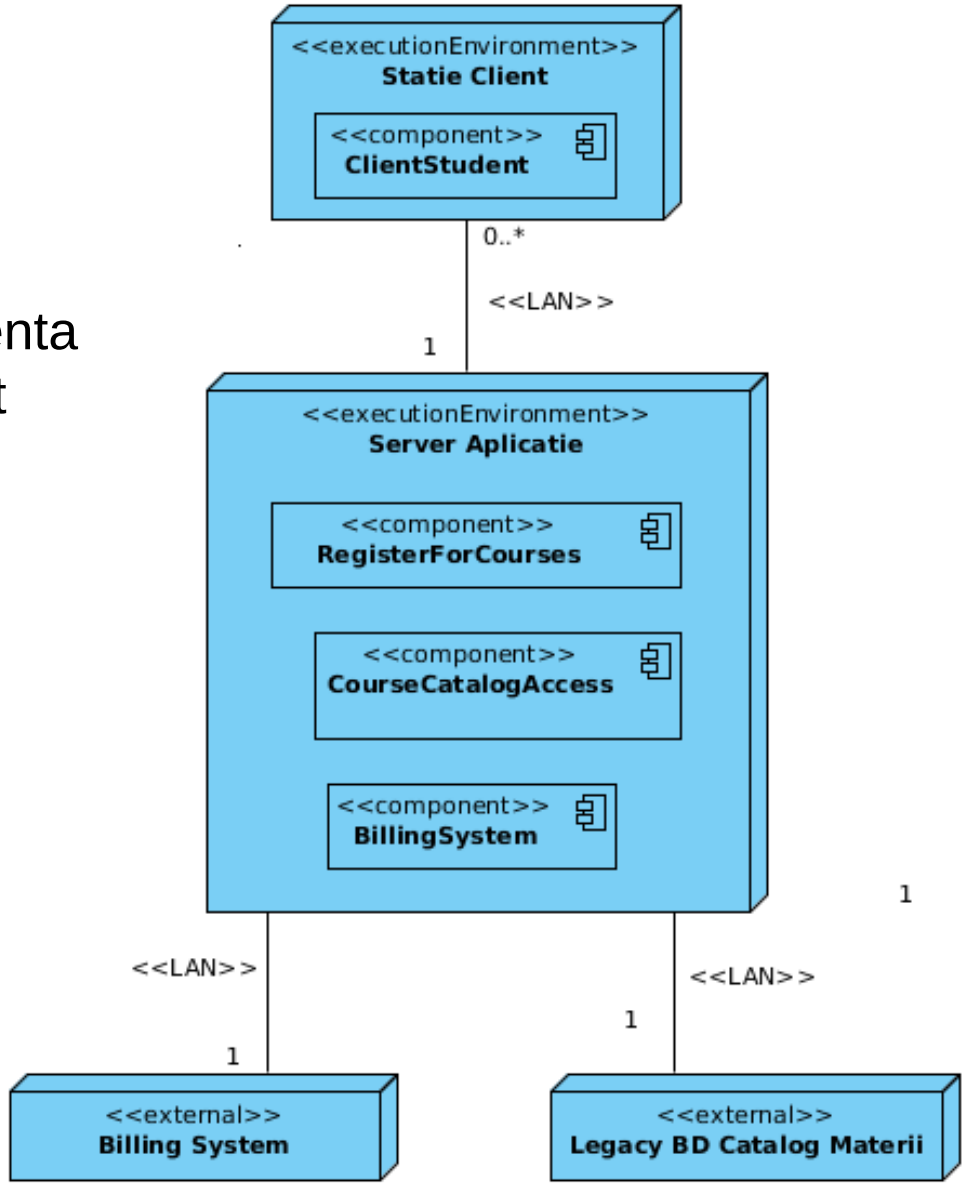
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-deployment-diagram/

- O diagramă de instalare conţine noduri conectate prin asocieri. Asocierile indică o cale de comunicare între noduri.
- Într-un nod putem reprezenta componente. Componentele conţinute într-un nod sunt cele care se execută pe nodul respectiv.
- Într-un nod putem reprezenta artefacte. Un artefact conţinut într-un anumit nod rezidă pe acel nod sau generează o componentă ce se execută pe acel nod.

MODELAREA ALOCĂRII PROCESELOR LA NODURI

Exemplu : diagramă de instalare (deployment)

În noduri se pot reprezenta componente, ca în acest exemplu



CONSIDERAȚII DE PROIECTARE

Exemplu : obiecte distribuite - mecanism de proiectare pentru distribuire

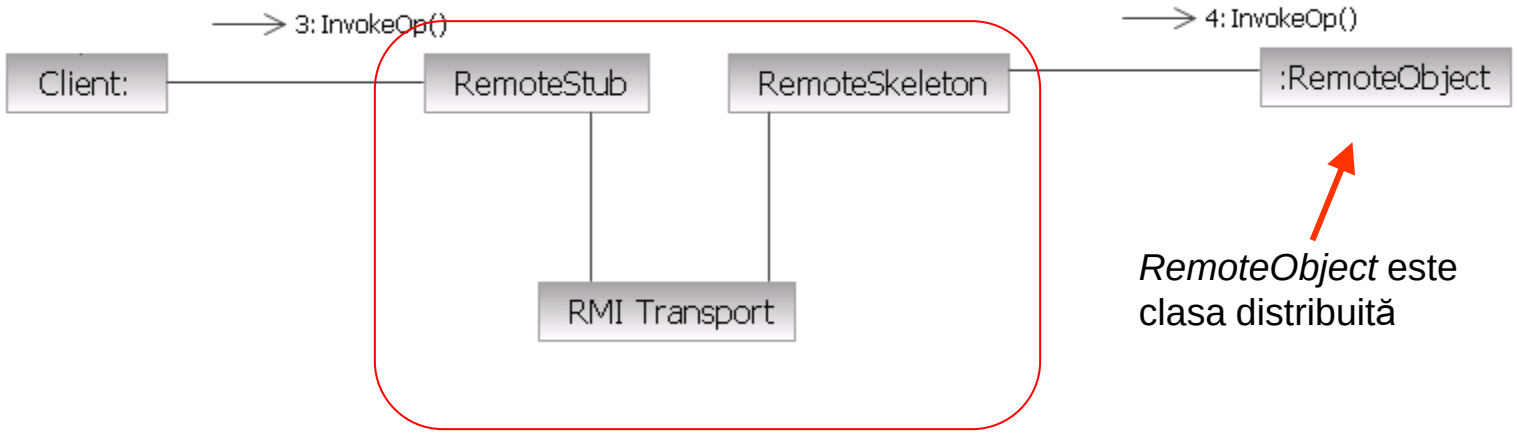
Mecanism de ***analiză*** (**conceptual**) – Distribuire

Mecanism de ***proiectare*** (**concret**) – Obiecte distribuite (Remote Procedure Call)

Mecanism de ***implementare*** (**actual**) – Java RMI (Remote Method Invocation)

CONSIDERAȚII DE PROIECTARE

Exemplu : RMI - mecanism de implementare pentru distribuire



RMI (Remote Method Invocation) – mecanism specific Java ce permite obiectelor client să invoce operaţii pe obiecte server aflate la distanţă (remote) ca şi cum obiectul server ar fi local.

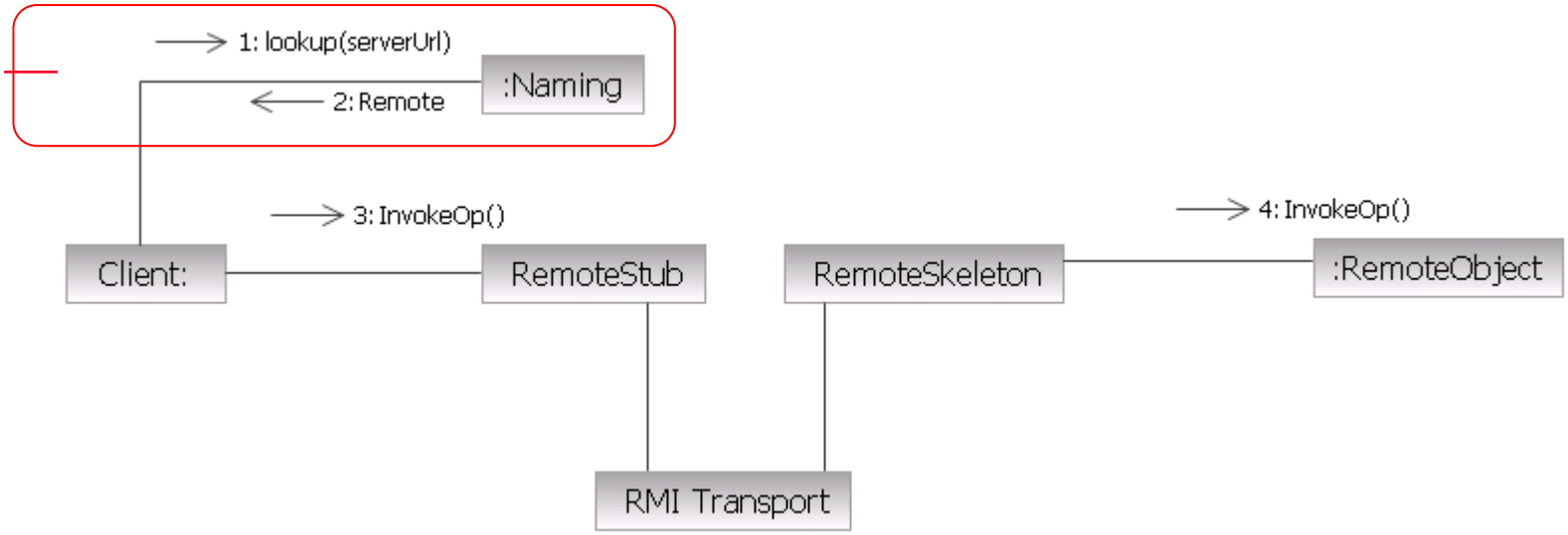
În varianta de bază, clientul trebuie să cunoască localizarea obiectului server.

Mecanismul de invocare a obiectului aflat la distanţă este implementat folosind câte un element “proxy” şi câte un serviciu de comunicare (RMI Transport) pe client şi pe server.

RemoteStub şi *RemoteSkeleton* sunt generate automat prin compilarea cu *rmic* a clasei distribuite compilate (cu *javac*).

CONSIDERAȚII DE PROIECTARE

Exemplu : RMI - mecanism de implementare pentru distribuire



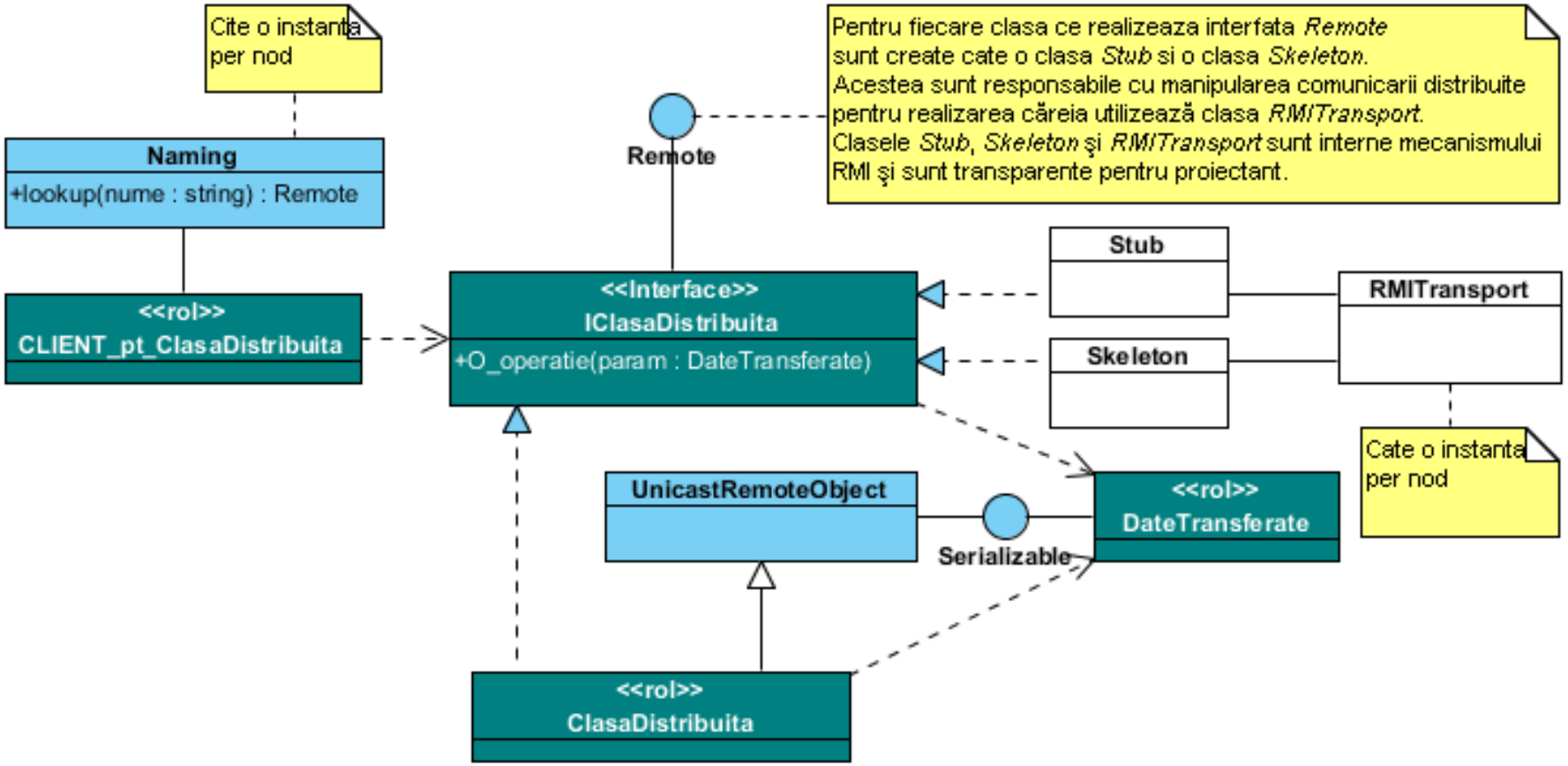
În varianta extinsă clientul stabileşte legătura cu obiectul aflat la distanţă folosind utilitatea *Naming* furnizată cu RMI.

Pe fiecare nod există o singură instanţă a clasei *Naming*. Instanţele clasei *Naming* comunică între ele pentru a localiza obiectele server.

Pentru căutarea unui obiect aflat la distanţă trebuie adăugat la client cod ce apelează metoda *lookup()* de pe clasa *Naming*. Acesta va întoarce o referinţă la *RemoteStub*.

Odată stabilită (prin *lookup()*) conexiunea cu un obiect aflat la distanţă, ea poate fi reutilizată ori de câte ori clientul necesită un acces la acesta.

CONSIDERAȚII DE PROIECTARE
Exemplu : RMI – vedere statică

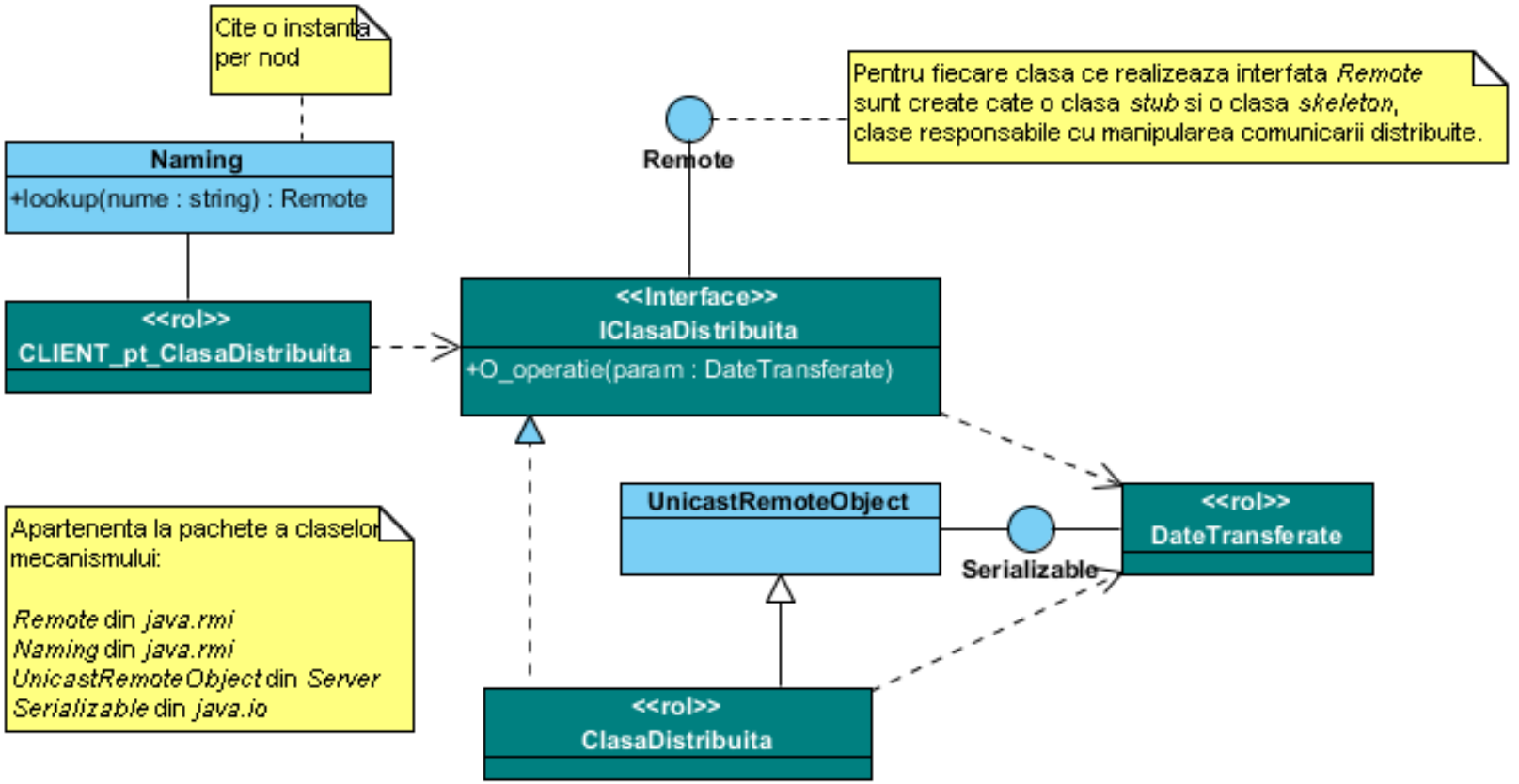


CONSIDERAȚII DE PROIECTARE
Exemplu : RMI – vedere statică

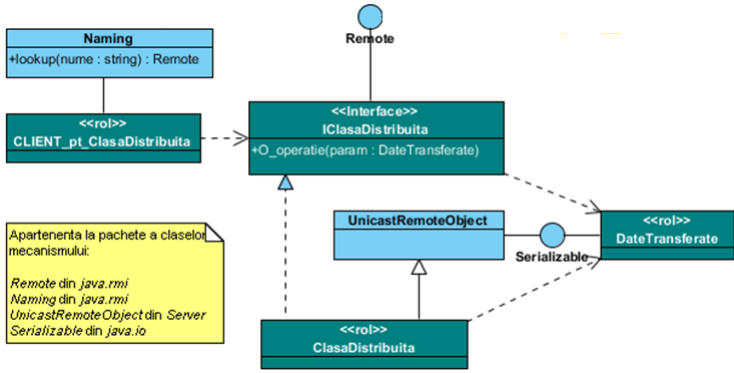
Mecanismul RMI pentru distribuire utilizat la proiectare

ClasaDistribuita şi *DateTransferate* sunt nume generice şi vor fi înlocuite cu numele claselor particulare în fiecare caz.

Proiectantul va înlocui clasele stereotipate cu <<rol>> cu clasele specifice aplicaţiei.



CONSIDERAȚII DE PROIECTARE
Exemplu : RMI – vedere statică



Descrierea claselor

Naming : Implementează mecanismul pentru obţinerea referinţelor la obiectele aflate la distanţă pe bază de URL. URL-ul unui obiect aflat la distanţă este specificat cu formatul tipic:

```
rmi://host:port/name
```

unde

`host` = numele host-lui unde se află registry (implicit host-ul curent)
`port` = numărul portului unde ascultă registry (implicit 1099)
`name` = numele obiectului aflat la distanţă

ClasaDistribuita : Nume generic pentru o clasă distribuită.

Remote : Interfaţă ce trebuie implementată (direct sau indirect) de toate obiectele ce trebuie accesate în regim distribuit, pentru a permite generarea obiectelor stub şi skeleton responsabile cu realizarea comunicării la distanţă ca suport pentru distribuire.

- Numai metodele specificate într-o interfaţă *Remote* sunt disponibile la distanţă.
- O clasă poate implementa orice număr de interfeţe *Remote* şi poate extinde alte clase ce implementează interfaţa *Remote*.

CLIENT_pt_ClasaDistribuita : Client generic pentru clasa distribuită.

DateTransferate : Date generice transferate la/de la clasa distribuită.

UnicastRemoteObject : Clasa *de infrastructură* ce implementează schema de comunicare cu obiectul distribuit.

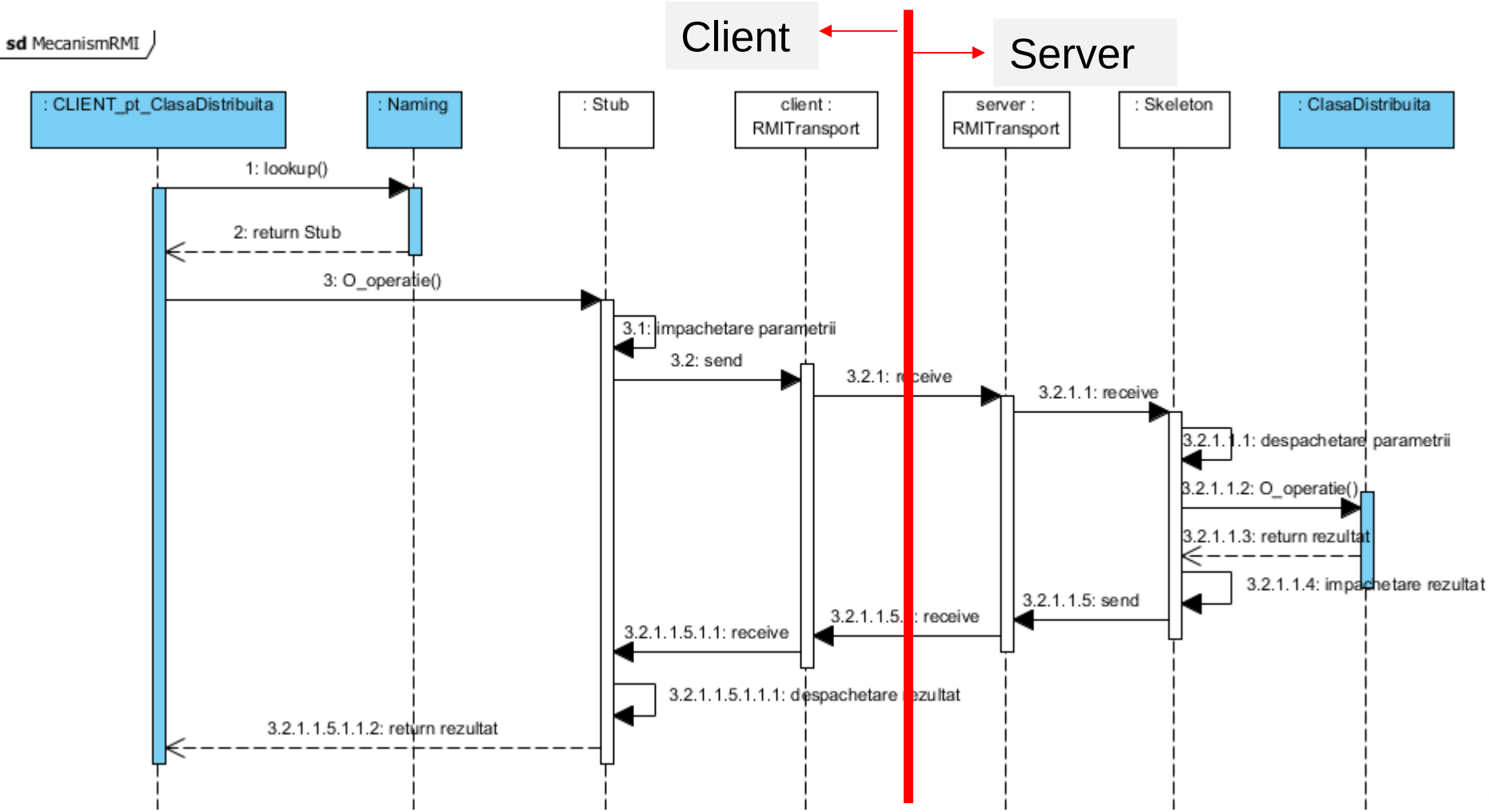
IClasaDistribuita : O interfaţă pentru clasa distribuită.

Serializable : Interfaţă ce trebuie realizată de orice clasă ale cărei instanţe trebuie transmise ca parametri la distanţă.

CONSIDERAȚII DE PROIECTARE
Exemplu : RMI – vedere dinamică

Stub, RMITRansport si *Skeleton* sunt transparente pentru proiectant.

DAR Clientul accesează *Stub* care oferă o interfață la *ClasaDistribuită.*

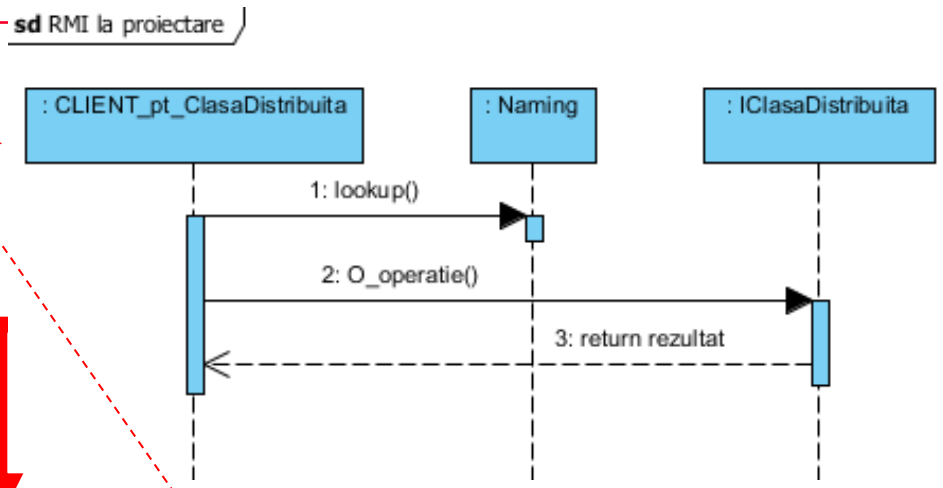


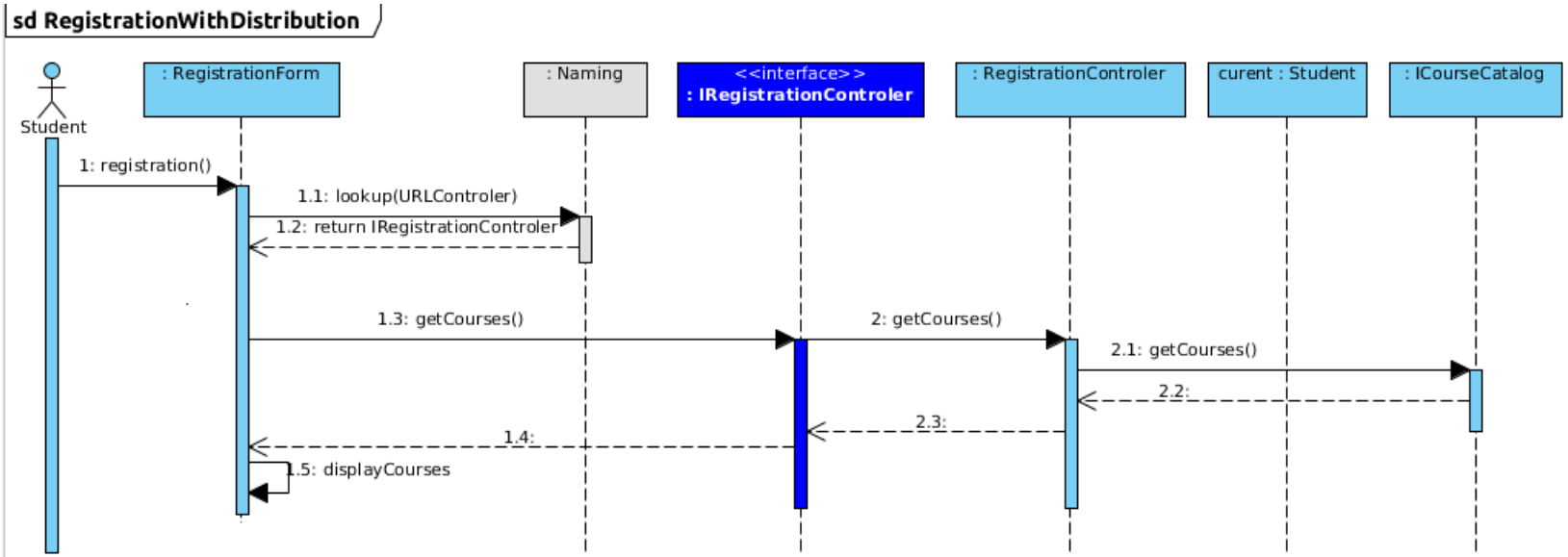
CONSIDERAȚII DE PROIECTARE
Exemplu : RMI – vedere dinamică

Mecanismul RMI utilizat la proiectare

Exemplu: Extras din diagrama de secvenţe a cazului de utilizare “Register for Courses”.

Modificări impuse de distribuire:
1. Introducerea clasei *Naming*
2. Accesarea serviciului *RegistrationControler* prin interfaţa *IRegistrationControler*.





...

Evaluare formativă

1. Dați exemple de două criterii de alocare procese la noduri.

2. Care este semnificația stereotipului <<manifest>> din diagrama de instalare (delopyment) ?

https://forms.gle/GpbpgFGqLhB8UcSE7

PLAN CURS

Descrierea arhitecturii la execuție și a distribuirii

Proiectare clase (cont.)

Proiectare BD

Principiile practice ale proiectării software-lui

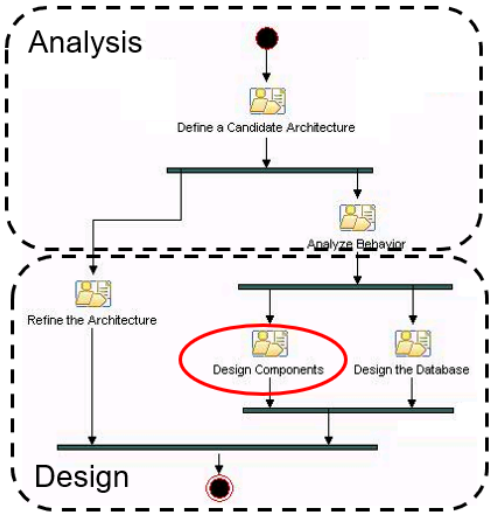
PROIECTARE CLASE (cont.)

Analiză

- Analiză arhitecturală (definire arhitectură candidat)
- Analiza UC (analiză comportament)

Proiectare

- Identificare elemente de proiectare (rafinarea arhitecturii)
- Identificare mecanisme de proiectare (rafinarea arhitecturii)
- Proiectare clase (proiectare componente)
- Proiectare subsisteme (proiectare componente)
- Descrierea arhitecturii la execuţie şi a distribuirii (Rafinarea arhitecturii)
- Proiectarea BD



Punerea în corespondenţă a claselor de analiză cu elemente de proiectare

Identificarea subsistemelor şi a interfeţelor dintre subsisteme

Actualizarea organizării modelului

OPORTUNITĂȚI de REUTILIZARE

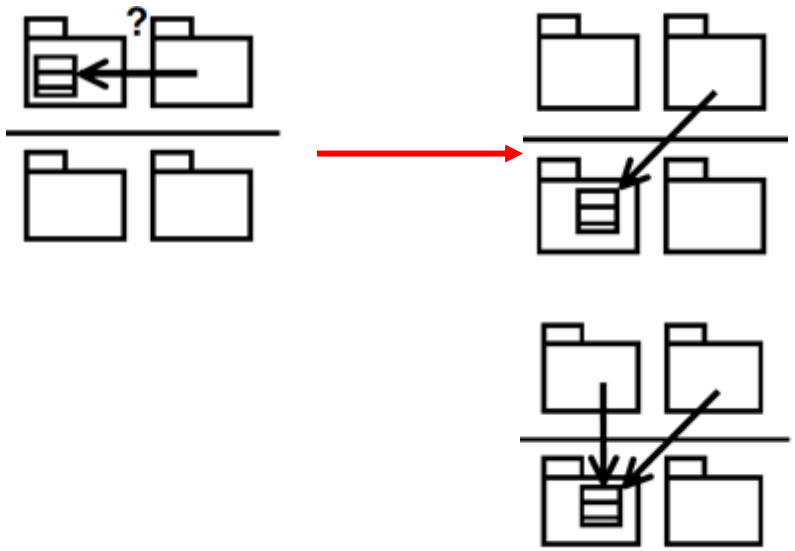
OBIECTIV:

- Identificarea posibilității de reutilizare a subsistemelor și/sau componentelor, pe baza interfețelor acestora.

ETAPE:

1. Căutarea de interfețe similare
2. Modificarea interfețelor noi pentru a se potrivi cu unele existente
3. Înlocuirea interfețelor candidat cu interfețe existente
4. Realizare corespondențe între subsisteme candidat și componente existente

REORGANIZARE PACHETE



Obiective:

- reducerea cuplării între pachete
- creșterea coeziunii în pachete

⇓

creșterea independenței funcționale.

- Problemă : Unele elemente sunt folosite de elemente din pachete diferite.
- Soluții :
  - Divizare elemente în mai multe obiecte: în cadrul elementului se pot identifica responsabilități disjuncte.
  - Mutare element într-un pachet de pe un nivel inferior astfel încât toate elementele de pe nivelul superior vor depinde de acesta în mod egal.

OPORTUNITĂȚI de REUTILIZARE
Elemente interne sistemului

După construirea unei prime aplicații și a unor părți generale, la construirea unei a doua aplicații se poate constata că unele părți din prima aplicație pot fi reutilizate, dar acestea nu au fost proiectate pentru a fi reutilizate.

⇓

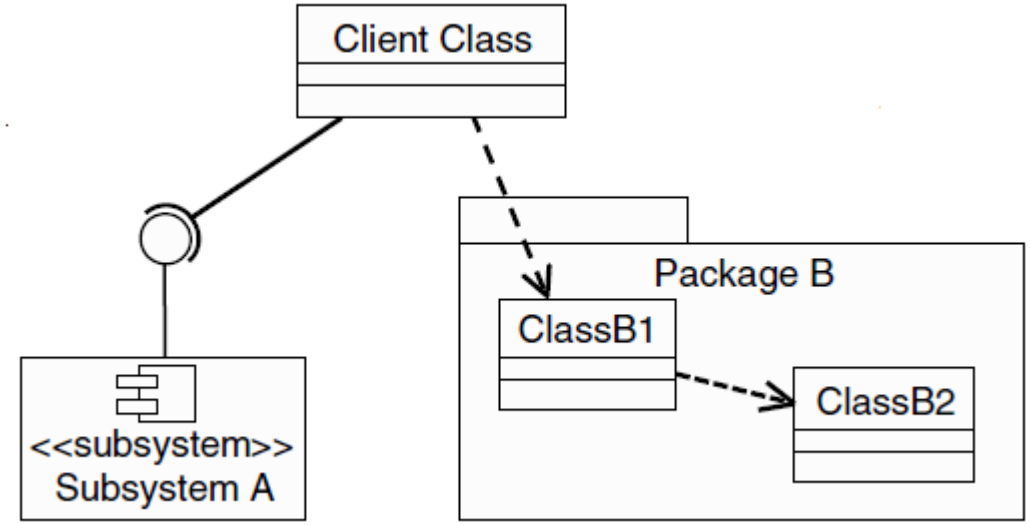
- Transformarea elementelor de proiectare (clase, pachete, subsisteme) candidate la reutilizare pentru a deveni reutilizabile: modificarea numelor lor, generalizarea lor, parametrizarea lor, îmbunătățirea documentației, etc.
- Elementul candidat la reutilizare este trimis pe un nivel inferior (ce conține funcționalități comune) al arhitecturii, astfel încât alte elemente să îl poată accesa. Inițial îl va utiliza elementul client original.

Avantaje :

- Elementele devenite reutilizabile pot fi folosite în noua aplicație.
- La actualizarea primei aplicații, vechile versiuni ale acestor elemente pot fi înlocuite cu variantele lor reutilizabile.

PACHETE vs. SUBSISTEME

SUBSISTEME	PACHETE
Modelează comportament	Modelează structură
Încapsulează complet conținutul	Nu încapsulează complet conținutul
Ușor de înlocuit	Ar putea fi dificil de înlocuit

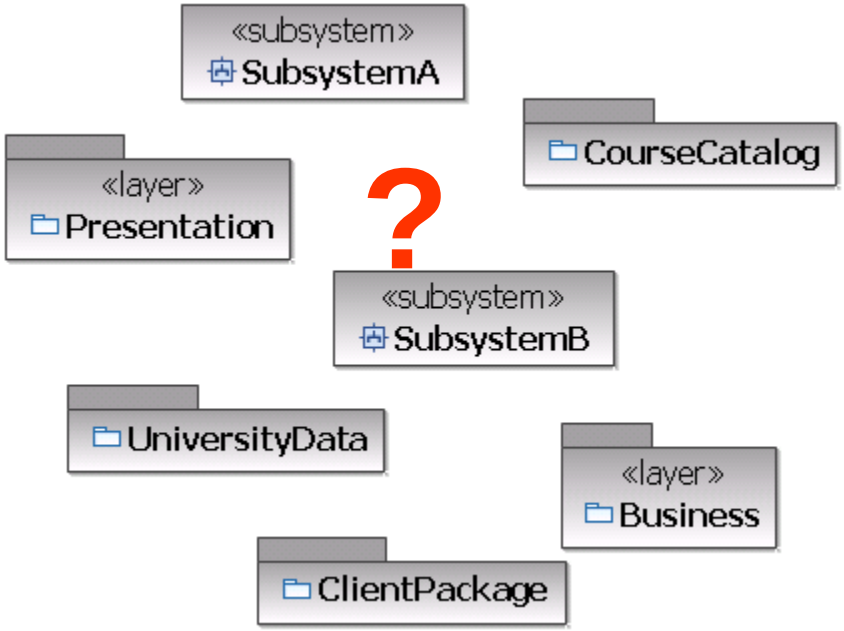


BLOCURILE CONSTITUTIVE ALE ARHITECTURII

- Atenţie! Construim o arhitectură bazată pe componente
  - Blocurile care compun arhitectura sunt pachetele, subsistemele, precum şi alte componente ale sistemului.

- Blocurile costitutive sunt organizate pe nivele, cu scopul de a atinge mai multe obiective cum ar fi disponibilitatea, securitatea, performanţa, utilizabilitatea, reutilizabilitatea aplicaţiei şi bineînţeles funcţionalitatea oferită utilizatorilor finali.

- Pentru atingerea acestor obiective trebuie să controlăm modul în care blocurile constitutive sunt împachetate şi asignate nivelelor.

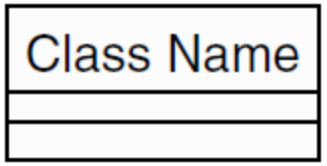


PACHETE DE PROIECTARE

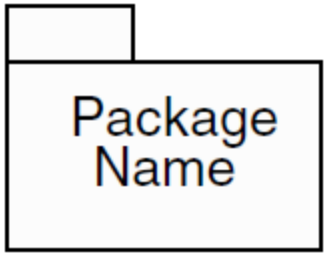
- Pachetele de proiectare sunt utilizate pentru a *grupa elemente de proiectare* aflate în *relaţie*.
- Pachetele de proiectare şi subsistemele sunt *blocuri constitutive* ale arhitecturii.
  - Trebuie organizate a.î. să îndeplinim obiectivele arhitecturii
  - Nu este suficientă simpla grupare a claselor aflate în relaţie
  - Se aplică principiile de bază ale orientării obiect:
    - încapsularea
    - separarea interfeţei de implementare
    - cuplarea slabă cu exteriorul
- Pachetele de proiectare sunt utilizate, de asemenea, ca elemente de *configurare* şi pentru a organiza *alocarea* lucrului la *echipele de dezvoltar*e.
- Atenţie! Dacă un element din pachetul A are o relaţie cu cel puţin un element din pachetul B, atunci între pachetul A şi pachetul B există o relaţie de dependenţă.

CLASE și PACHETE

Clasă = descrierea unui set de obiecte care partajează aceleași reponsabilități, relații, operații, atribute, semantică (semnificație).

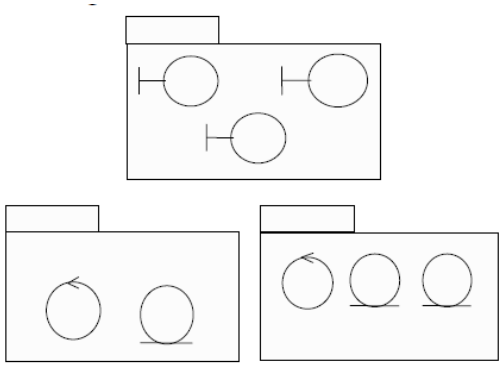


Pachet = mecanism general de organizare a elementelor în grupuri; este un element de modelare care conține elemente de modelare.

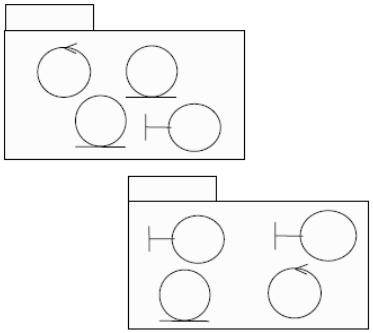


RECOMANDĂRI de ORGANIZARE CLASE în PACHETE

Dacă se preconizează că interfața sistemului va suferi modificări considerabile, clasele <<*boundary*>> vor fi plasate în pachete separate.



Dacă se preconizează că interfața sistemului NU va suferi modificări considerabile, clasele <<*boundary*>> vor fi plasate în pachete cu clase cu care au relații funcționale.



Clasele <<*boundary*>> care nu au relații funcționale cu clase <<*entity*>> sau <<*control*>> trebuie plasate în pachete separate, împreună cu clasele <<*boundary*>> care aparțin aceleiași interfețe.

Dacă o clasă <<*boundary*>> este în relație cu un serviciu opțional, aceasta va fi grupată într-un subsistem separat împreună cu clasele care colaborează pentru a furniza acel serviciu.

RECOMANDĂRI de ORGANIZARE CLASE în PACHETE

Criterii de determinare dacă două clase sunt *relaționate funcțional* :

- Modificări la comportamentul și structura unei clase necesită modificări în cealaltă clasă.
- Îndepărtarea unei clase are impact asupra celeilalte clase.
- Obiecte din clase diferite interacționează printr-un număr mare de mesaje sau au o intercomunicare complexă.
- O clasă <<*boundary*>> este legată funcțional de o clasă <<*entity*>> dacă funcția clasei <<*boundary*>> este de a prezenta datele din clasa <<*entity*>>.
- Două clase interacționează cu, sau sunt afectate de schimbări la un același actor.
- Există relații între cele 2 clase.
- O clasă crează instanțe ale celeilalte clase.

RECOMANDĂRI de ORGANIZARE CLASE în PACHETE

- Dacă un element este în relaţie cu un serviciu opţional, acest serviciu va fi grupat împreună cu colaboratorii săi într-un subsistem separat.
- Consideraţi mutarea a două elemente de proiectare în pachete diferite dacă:
  - Unul este opţional şi celălalt obligatoriu
  - Sunt în relaţie cu actori diferiţi
- Analizaţi dependenţele pe care elementele co-locatare le au cu elementul studiat.
- Analizaţi cât este de stabil elementul de proiectare:
  - Încercaţi să mutaţi elementele stabile pe nivele inferioare şi cele instabile pe nivele superioare ale arhitecturii.

DEPENDENȚE ÎNTRE PACHETE

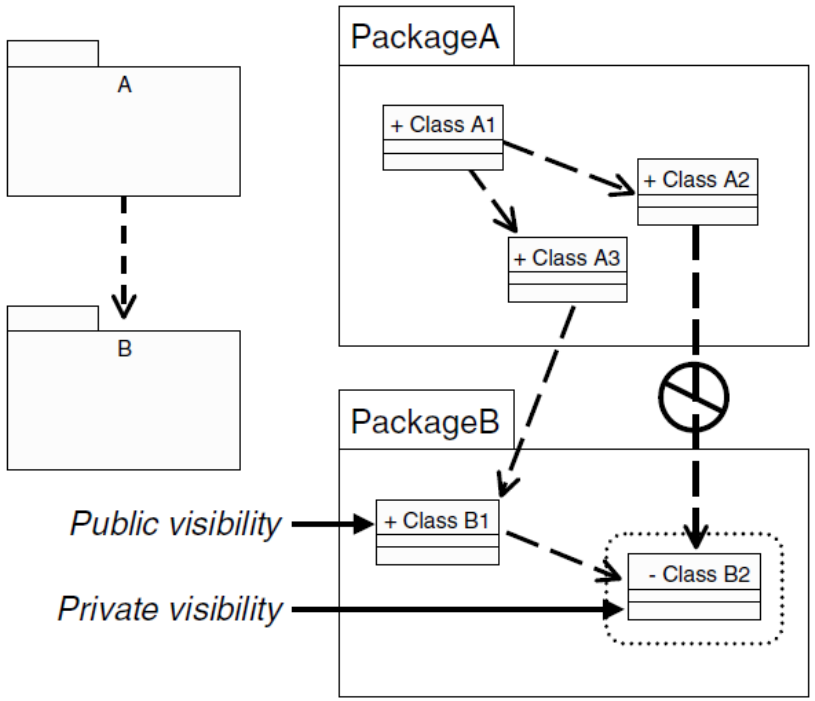
Se poate defini vizibilitate la nivel de element conținut în pachet (aici la nivel de clasă).

Tipuri de vizibilitate:

Public: Accesibile și din afara pachetului propriu. Simbol: +.
Private: Accesibile doar din pachetul propriu și din orice pachet care moștenește din pachetul propriu. Simbol: -.

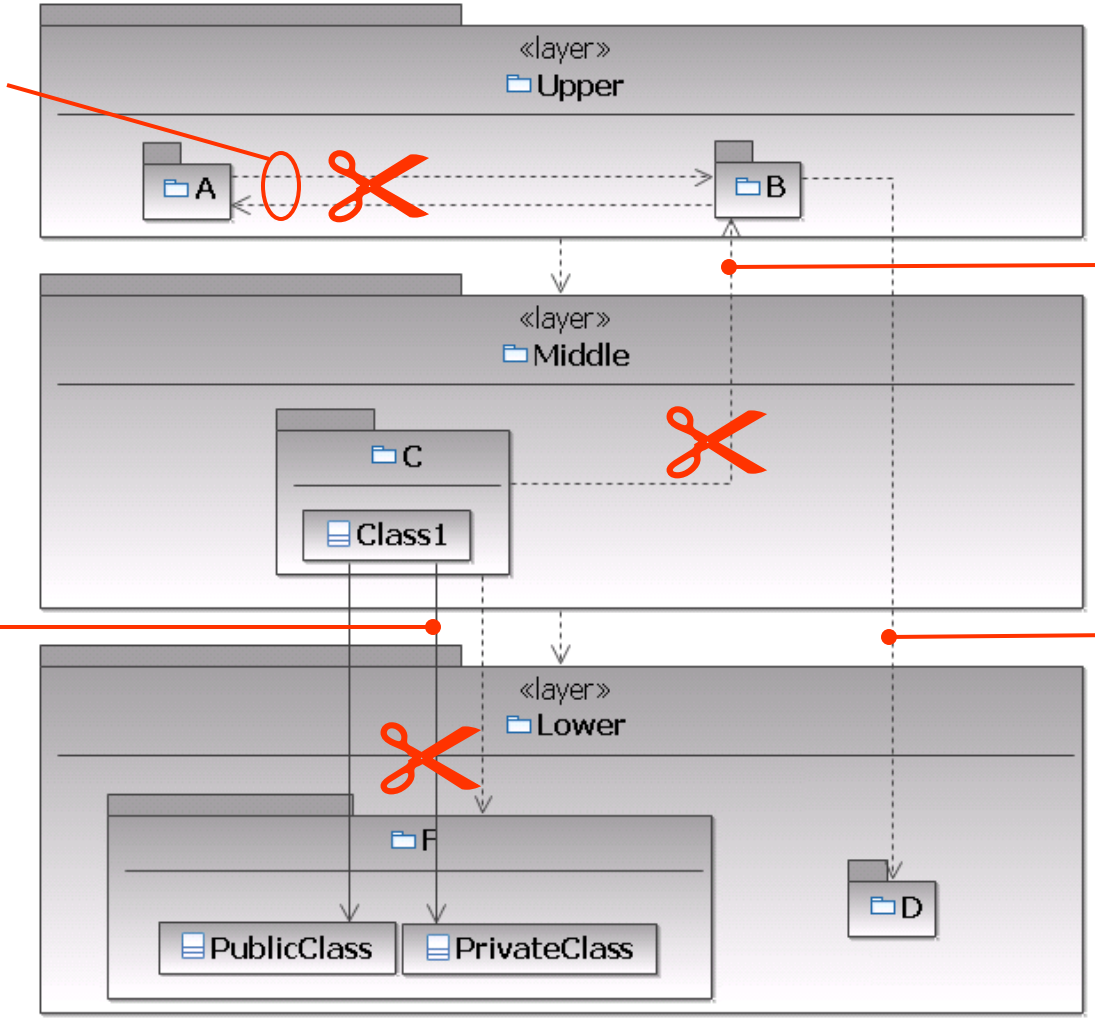
Elementele publice ale unui pachet constituie interfața pachetului.

Toate dependențele de un pachet trebuie să fie dependențe de elementele publice ale acestuia.



EVALUAREA CUPLĂRII PACHETULUI

Nu sunt permise dependenţele circulare.



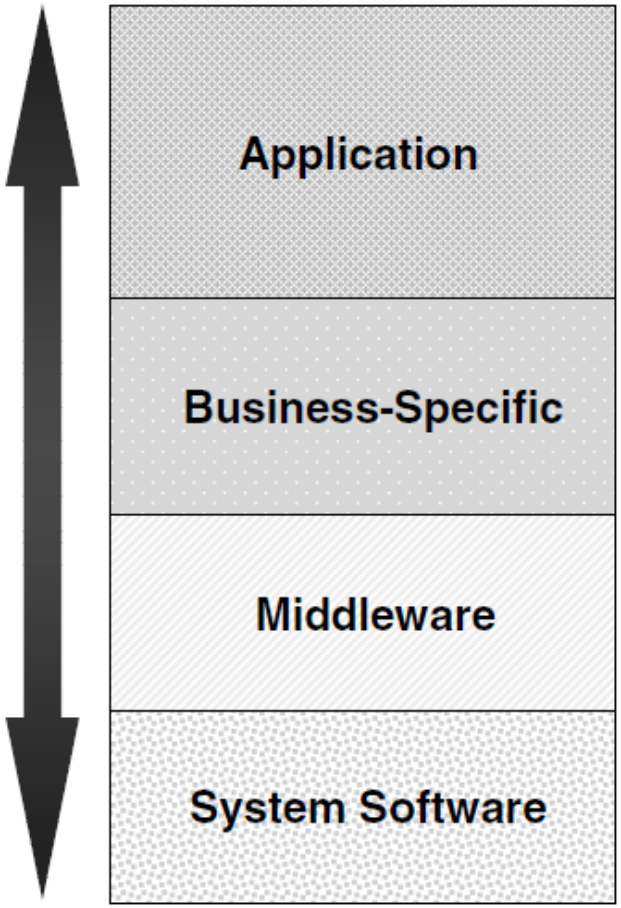
Pachetele de pe nivelele inferioare nu trebuie să depindă de pachete de pe nivelele superioare.

Doar clasele publice pot fi accesate din exteriorul pachetului ce le conţine.

Evitaţi trecerea peste nivele.

ORGANIZARE MULTI-NIVEL TIPICĂ

Funcționalitate specifică



Funcționalitate generală

Componentele specifice aplicației

Sisteme reutilizabile specifice domeniului aplicației (tipului de business).

Subsisteme cu clase utilitare și servicii independente de platformă (pentru calcul distribuit în platforme eterogene, etc.)

Software de infrastructură (SO, interfețe la dispozitive Hw specifice, device drivers, etc.)

CARACTERISTICI ALE ORGANIZĂRII MULTI-NIVEL

Vizibilitate

- Dependențe doar în nivelul curent și cu nivele inferioare

Volatilitate

- Nivelele superioare sunt afectate de modificări ale cerințelor
- Nivelele inferioare sunt afectate de modificări ale contextului

Generalitate

- Elemente de model mai abstracte pe nivelele inferioare

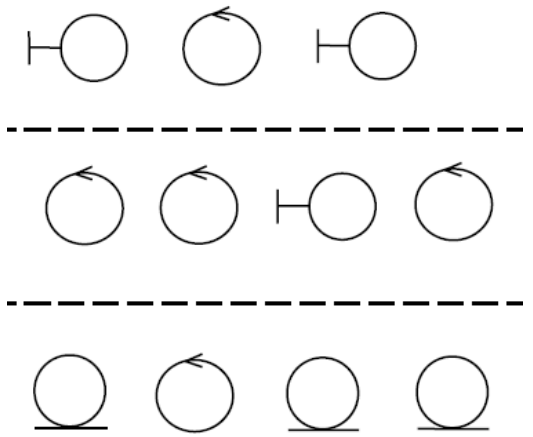
Numărul nivelelor

- Sisteme mici : 3-4 nivele
- Sisteme complexe : 5-7 nivele

ALOCARE ELEMENTE DE PROIECTARE LA NIVELE

Deși toate cele 3 stereotipuri pot să apară în orice nivel, există tendințe generale care ghidează proiectantul.

- Majoritatea claselor <<*boundary*>> tind să apară pe nivelul superior
- Majoritatera claselor <<*control*>> tind să apară pe nivelul serviciilor unde este necesar controlul entităților de date.
- Majoritatea claselor <<*entity*>> apar către nivelele inferioare.

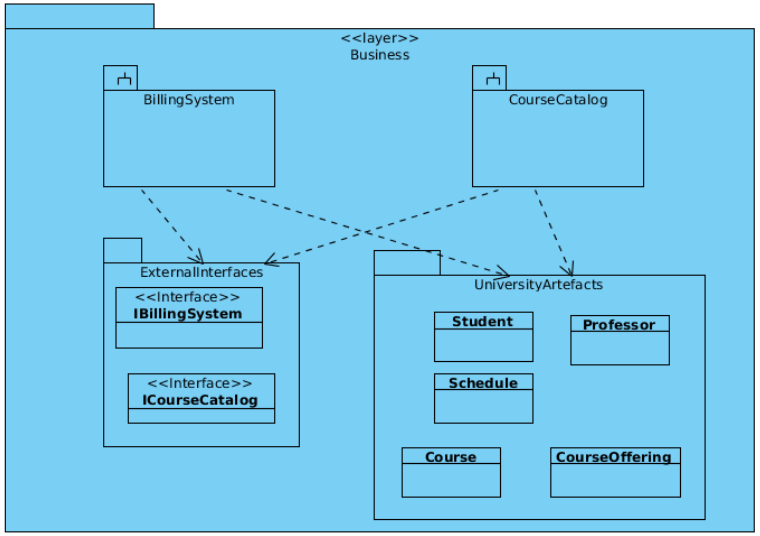


Proiectanții experimentați crează pachete de *clase care colaborează* pentru a oferi *un serviciu*, adică vor grupa elemente din sistem care colaborează strâns pentru a oferi o capabilitate de nivel înalt a sistemului $\Rightarrow$ *pachete coezive, reutilizabile.*

Evaluare formativă

1. Dați două exemple de criterii pentru grupare clase în pachete.

2. Care este punctul slab al acestui model de organizare ? Ce modificări sugeraţi?



https://forms.gle/scV4fvYtMjpuo1sP7

PLAN CURS

Descrierea arhitecturii la execuție și a distribuirii

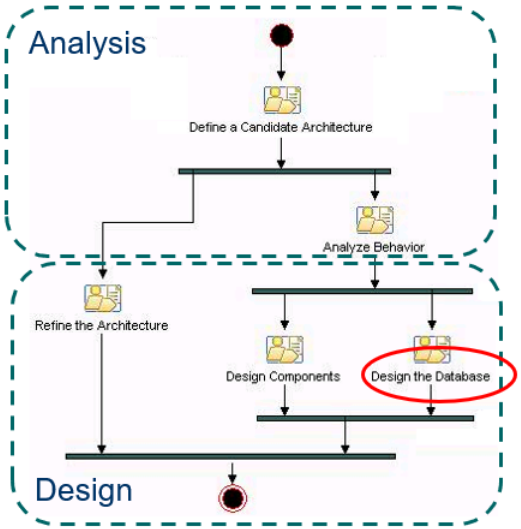
Proiectare clase (cont.)

Proiectare BD

Principiile practice ale proiectării software-lui

PROIECTAREA BAZEI DE DATE

- Analiză
  - Analiză arhitecturală (definire arhitectură candidat)
  - Analiza UC (analiză comportament)
- Proiectare
  - Identificare elemente de proiectare (rafinarea arhitecturii)
  - Identificare mecanisme de proiectare (rafinarea arhitecturii)
  - Proiectare clase (proiectare componente)
  - Proiectare subsisteme (proiectare componente)
  - Descrierea arhitecturii la execuţie şi a distribuirii (Rafinarea arhitecturii)
  - Proiectarea BD



Baze de date relaţionale şi orientarea obiect

Maparea claselor de obiecte pe tabele din BD relaționale

Strategii de implementare a persistenţei

MODELUL RELAȚIONAL și ORIENTAREA OBIECT

RDBMS şi Orientarea Obiect nu sunt perfect compatibile.

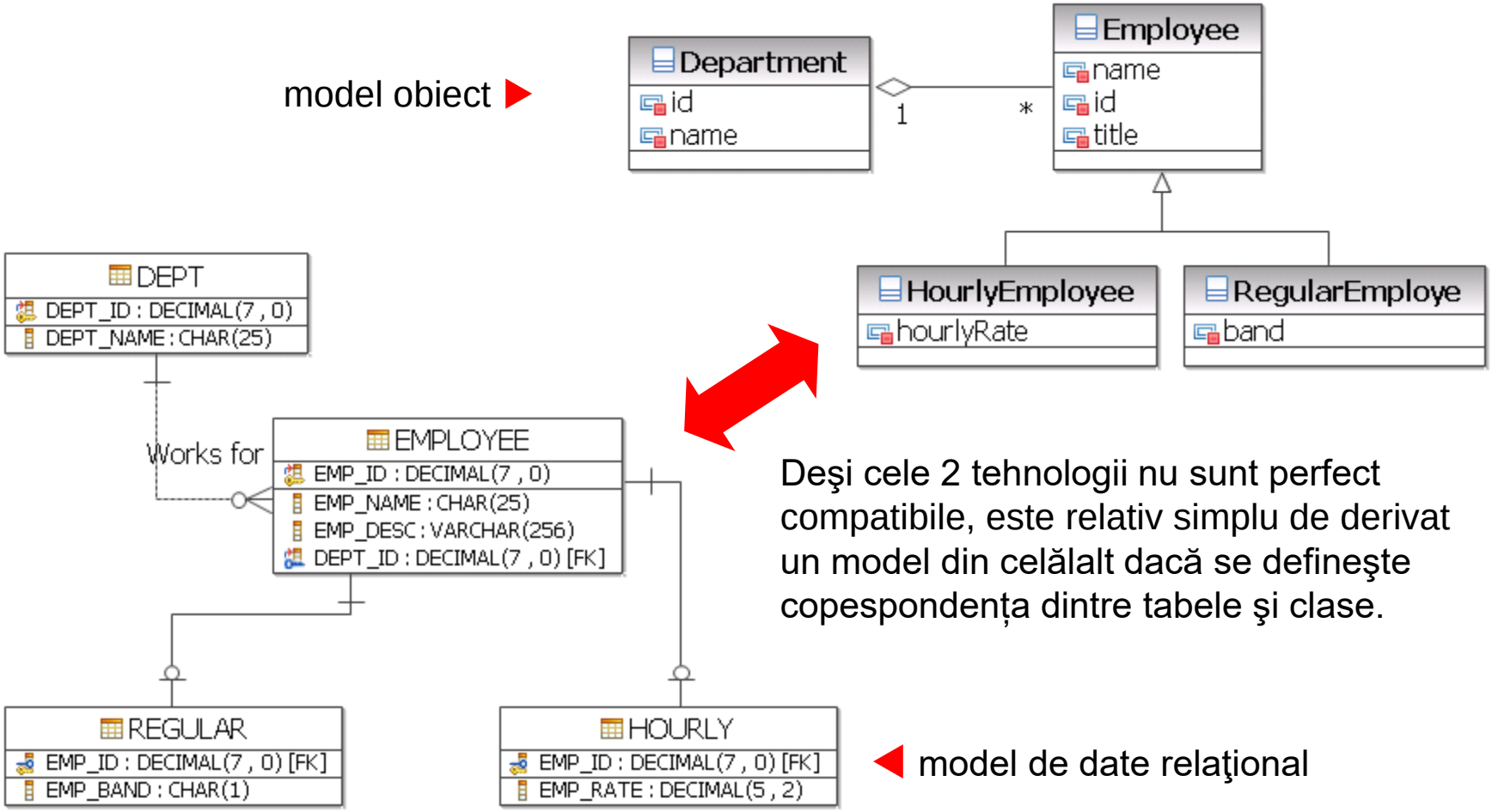
RDBMS

- Concentrat pe date
- Potrivit pentru aplicaţii ce realizează rapoarte cu date aflate în relaţii stabilite dinamic sau ad-hoc
- **Expune datele (valorile din coloane)**

Sistem OO

- Concentrat pe comportament
- Potrivit pentru sisteme cu comportament complex şi comportament bazat pe stări în care datele sunt de interes secundar, sau pentru sisteme în care datele sunt accesate navigaţional într-o ierarhie naturală (ex. Facturi de materiale)
- **Ascunde datele (încapsularea atributelor în spatele interfeţei publice)**

MODELUL DATELOR și MODELUL OBIECT



Deşi cele 2 tehnologii nu sunt perfect compatibile, este relativ simplu de derivat un model din celălalt dacă se defineşte copespondența dintre tabele şi clase.

MAPAREA CLASELOR PERSISTENTE
pe TABELE din BD relaționale

Clasele persistente din modelul proiectării reprezintă *informaţiile ce trebuie memorate* de către sistem. Conceptual, aceste clase se aseamănă cu un design relaţional (de exemplu, *clasele* din modelul proiect trebuie să fie reflectate ca *entităţi ale unei scheme relaţionale*).

De la elaborare la construcţie, obiectivele modelului proiect şi modelului relaţional al datelor au o evoluţie divergentă.

- Obiectivul dezvoltării bazei de date relaţionale este *normalizarea datelor*.
- Obiectivul modelului proiect este *încapsularea datelor și comportamentului* cu complexitate în continuă creştere.

⇓

necesitatea punerii în corespondenţă (mapării) a elementelor, din cele două modele, aflate în relaţie.

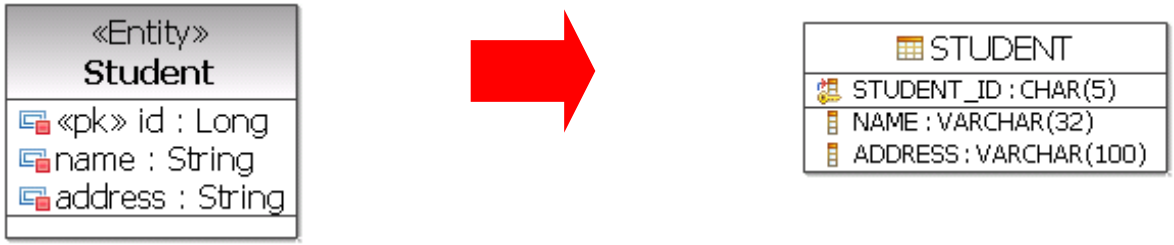
Într-o bază de date relaţională scrisă în forma normală 3, fiecare *rând* de tabelă ("tuplu") este văzut ca un *obiect*. O *coloană* de tabelă este echivalentă cu un *atribut persistent* al unei clase (o clasă persistentă poate avea şi atribute tranzitorii).

Dacă nu există asocieri cu alte clase, maparea este simplă. Tipul atributului trebuie să corespundă cu tipul coloanei pe care se mapează.

MAPAREA CLASELOR PERSISTENTE
pe TABELE din BD relaționale

Numai clasele UML persistente trebuie mapate pe tabele din baza de date (tipic clasele <<Entity>>)

- Un obiect UML se mapează pe o înregistrare (row).
- Un atribut UML se mapează pe un atribut (coloană).
- Cheia primară a tabelei se mapează pe atribute explicite ale clasei UML sau ea trebuie creată (dacă nu există echivalent în clasa UML).



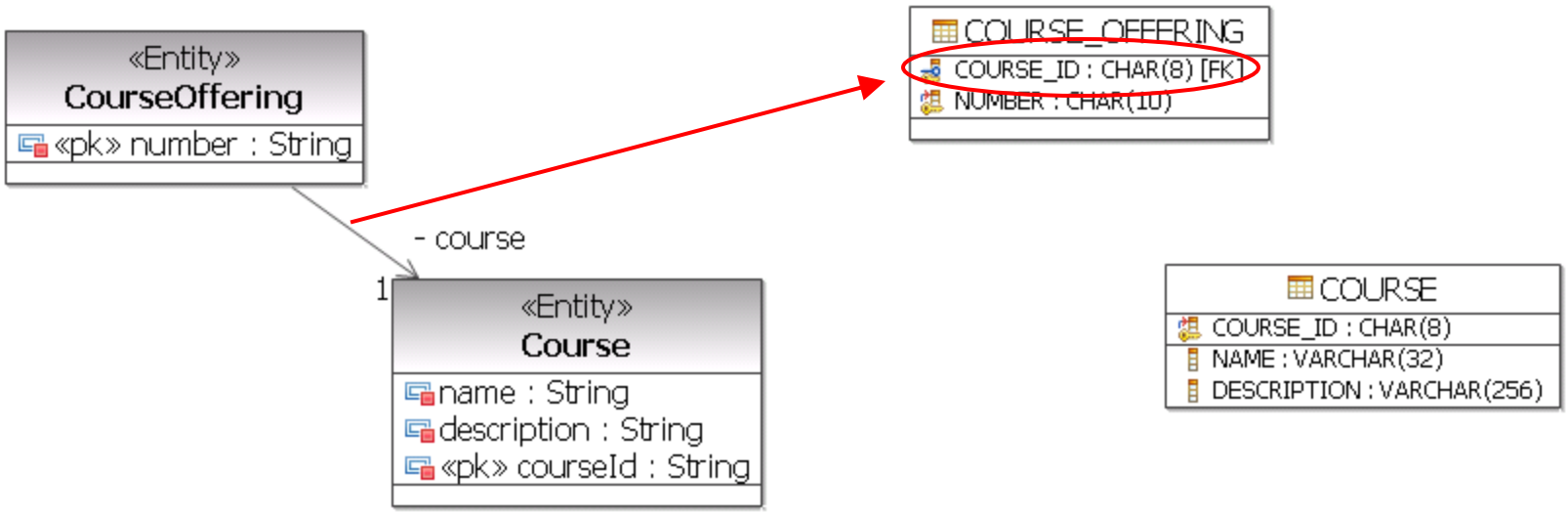
- În cadrul OMG se lucrează la specificaţiile unui **profil UML standard pentru modelarea datelor**.
- Acesta este util pentru reglajul fin al mapării dintre clase şi tabele (ex. utilizarea unui stereotip <<pk>> pentru a indica ce atribut trebuie mapat pe cheia primară).

Exemplu : http://www.agiledata.org/essays/umlDataModelingProfile.html

MAPAREA ASOCIERILOR dintre CLASE PERSISTENTE, în BD relaționale

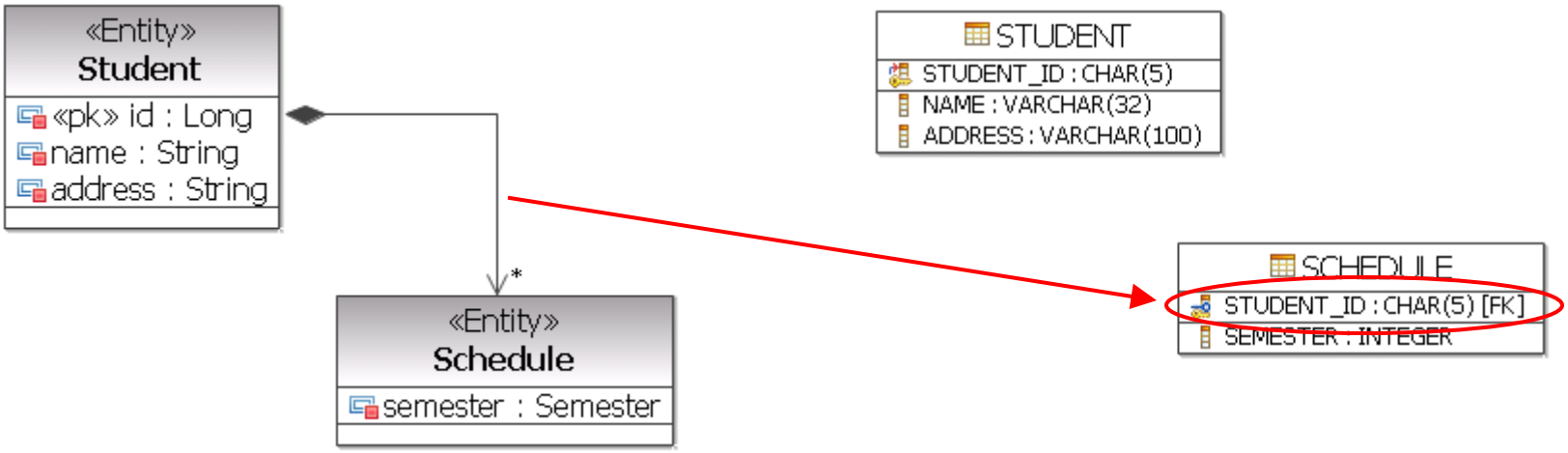
Asocierile dintre două obiecte persistent sunt realizate sub formă de *chei străine* ale obiectelor asociate.

O cheie străină (foreign key) este o coloană dintr-o tabelă care conţine valoarea cheii primare a obiectului asociat.



MAPAREA AGREGĂRILOR pe MODELUL DATELOR, în BD relaționale

Agregarea este modelată folosind, de asemenea, *chei străine*.



MODELAREA ALTOR RELAȚII OO, în BD relaționale

- Modelarea relaţiilor *many-to-many*
  - Crearea unei *tabele asociative* care conţine *cheile străine* ale celor două tabele.

- Modelarea *moştenirii* în modelul datelor

  Modelul relaţional al datelor *nu suportă modelarea directă a moştenirii*.
  - Opţiuni:
    - Maparea *întregii ierarhii* de clase pe *o singură tabelă*.
    - Maparea fiecărei *clase concrete* pe o *tabelă proprie*.
    - Maparea fiecărei *clase* pe o *tabelă proprie* $\Rightarrow$ subclasa va conţine o *cheie străină ce referă superclasa*.

STRATEGII DE IMPLEMENTARE A PERSISTENȚEI

Obiectele business accesează direct sursele de date

- în aplicaţii Java aceasta se realizează în mod tipic prin utilizarea JDBC
- simplu, dar obiectele business sunt cuplate direct la baza de date

Data access objects (DAOs)

- DAOs încapsulează logica de acces la baza de date
- izolează obiectele business de sursele de date, oferindu-le o interfaţă de acces la date în termeni de obiecte şi tipuri de date specifice domeniului.

Cadre (frameworks) pentru persistenţă

- codul de acces la baza de date este generat automat de către framework-ul de persistenţă.
  - oferă în general o performanţă mai bună pe ansamblu.

Exemple:

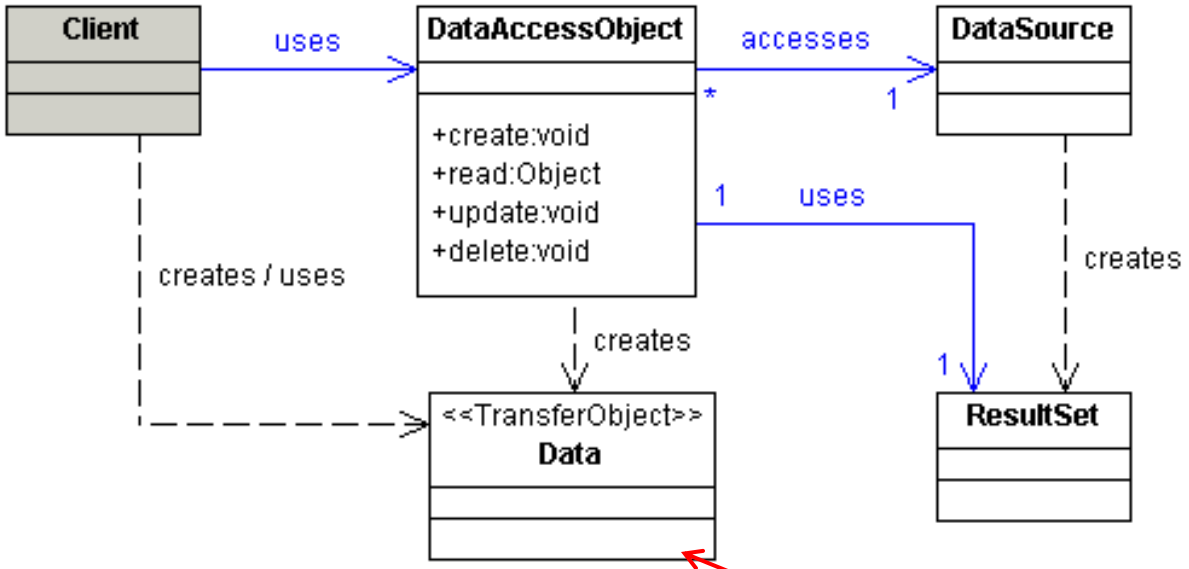
- ORM – Object-Relational Mapping : Enterprise JavaBeans (EJB), Hibernate, OJB (ObJectRelationalBridge), iBATIS, CodeIgniter, etc.
- Pt. BD NoSQL : Eclipse JNoSQL, NeoEMF (a multi NoSQL Persistence Framework for Very Large Models), etc.

Orice combinaţie între cele anterioare

ŞABLONUL DAO (Data Access Object)

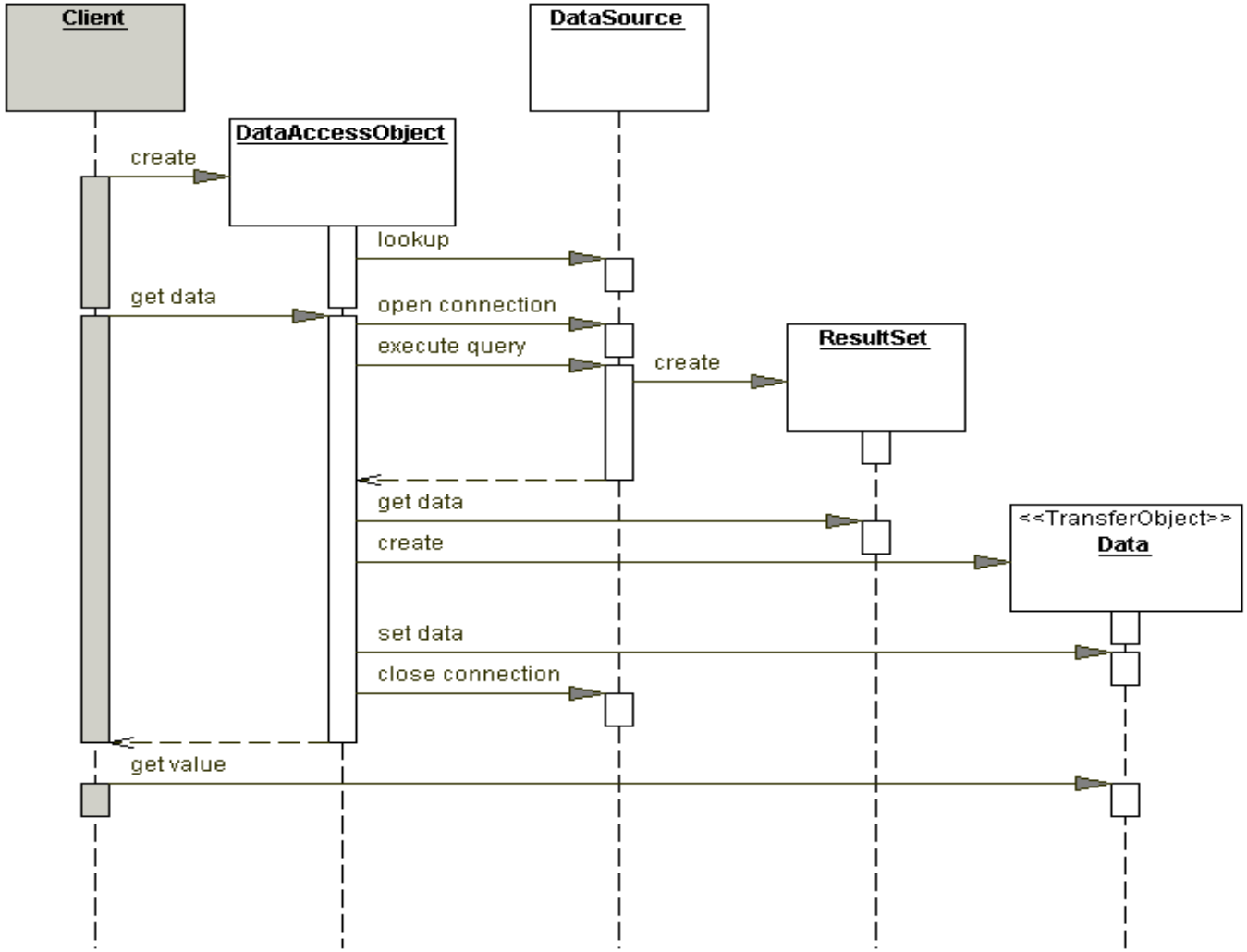
Sursă: Core J2EE Patterns, Deepak Alur, John Crupi & Dan Malks, Prentice Hall, 2003

- DAO încapsulează toate accesele la memoria persistentă
- DAO gestionează conexiunea cu sursa de date pentru a memora şi extrage date.



https://www.oracle.com/technetwork/java/dataaccessobject-138824.html

ȘABLONUL DAO (Data Access Object)



Evaluare formativă

1. Realizați corespondența corectă între elemente ale claselor persistente din diagrama de clase și elemente din baza de date relațională.

2. Enumerați strategiile de implementare a persistenței.

https://forms.gle/4oBLwRSKbohw6Cvz6

PLAN CURS

Descrierea arhitecturii la execuție și a distribuirii

Proiectare clase (cont.)

Proiectare BD

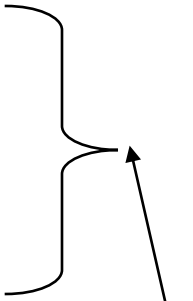
Principiile practice ale proiectării software-lui

PRINCIPII PRACTICE ALE PROIECTĂRII SOFTWARE-lui

#1 *Corespondenţă* completă, uşor de urmărit, cu *modelul analiză*.

Modelul analiză:

- domeniul informaţional al problemei
- funcţiile vizibile
- comportamentul sistemului
- setul de clase de analiză (obiecte şi servicii business)



Modelul proiect:

- elementele ce realizează modelul analiză
- elemente de infrastructură software

PRINCIPII PRACTICE ALE PROIECTĂRII SOFTWARE-lui

#2 Întotdeauna se începe cu *definirea arhitecturii.*

De ce ?

Arhitectura (scheletul sistemului ce va fi construit) afectează:

- interfeţele,
- structurile de date,
- fluxul de control şi comportamentul,
- modul de conducere a testării,
- mentenabilitatea sistemului, etc.

PRINCIPII PRACTICE ALE PROIECTĂRII SOFTWARE-lui

#3 Proiectarea *datelor* este la fel de importantă ca proiectarea funcţiilor de procesare.

Proiectarea datelor – element esenţial al proiectării arhitecturale.

De ce ?

Structura datelor influenţează:

- complexitatea fluxului de control,
- complexitatea componentelor software,
- eficienţa procesării.

PRINCIPII PRACTICE ALE PROIECTĂRII SOFTWARE-lui

#4 Proiectarea cu grijă a *interfeţelor interne şi externe*.

De ce ?

Fluxul datelor cu elementele din context și între componentele sistemului influenţează:

- eficienţa procesării,
- propagarea erorilor,
- complexitatea design-ului,
- integrarea componentelor
- modul de validare a funcţionalităţilor componentelor

PRINCIPII PRACTICE ALE PROIECTĂRII SOFTWARE-lui

#5 Proiectarea *interfeţei cu utilizatorul* conform necesităţilor acestuia şi cât mai simplu de utilizat.

De ce ?

Impune percepţia asupra calităţii software-lui.

PRINCIPII PRACTICE ALE PROIECTĂRII SOFTWARE-lui

#6 Se va urmări *independenţa funcţională* a componentelor.

Funcţionalitatea componentei - *puternic coezivă*, i.e. concentrată pe o singură funcţie sau subfuncţie.

Cuplarea slabă a componentelor între ele şi cu contextul extern.

Cuplarea (interfeţe, mesaje, date globale) păstrată la nivele cât mai joase.

PRINCIPII PRACTICE ALE PROIECTĂRII SOFTWARE-lui

#8 *Artefactele* proiectării trebuie să fie uşor de înţeles.

De ce ?

Scop – comunicarea de informaţii către:

- programatori
- testeri
- personal de întreţinere

PRINCIPII PRACTICE ALE PROIECTĂRII SOFTWARE-lui

#9 *Dezvoltarea iterativă* a modelului proiect.

La fiecare iteraţie se urmăreşte şi creşterea claritătii şi simplitătii.

Cum ?

Primele iteraţii : rafinare design şi corectare erori.

Iteraţiile ulterioare : simplificarea modelului proiect.

PRINCIPII PRACTICE ALE PROIECTĂRII SOFTWARE-lui

SET GENERIC DE ACTIVITĂŢI (1)

- Proiectarea *structurilor* obiectelor de *date* şi atributelor acestora, pe baza modelului informaţional al domeniului.
- Selectarea unui *stil (şablon) arhitectural* corespunzător, pe baza modelului analiză.
- *Divizarea* modelului analiză în subsisteme şi *alocarea* acestora elementelor arhitecturii:
    - Alocarea *claselor şi funcţiilor*, definite în modelul analiză, la subsisteme.
    - Definire *subsisteme* a.î. fiecare să fie *coezive d.p.d.v. funcţional.*
    - Proiectarea *interfeţelor* subsistemelor.

SET GENERIC DE ACTIVITĂŢI (2)

- Proiectarea tuturor *interfeţelor* necesare *cu sistemele şi dispozitivele externe.*
- Crearea setului de *clase de proiectare*:
  - Traducerea descrierii fiecărei *clase de analiză* în clasă sau subsistem (de clase) de proiectare.
  - *Verificarea* fiecărei clase de proiectare.
  - Definirea *metodelor şi mesajelor* asociate fiecărei clase de proiectare.
  - Evaluarea şi selectarea *şabloanelor/mecanismelor de proiectare* pentru diferite clase de proiectare sau subsisteme.

PRINCIPII PRACTICE ALE PROIECTĂRII SOFTWARE-lui

SET GENERIC DE ACTIVITĂŢI (3)

- Proiectarea *interfeţei utilizator*:
  - *Revizuirea* rezultatelor analizei *sarcinilor utilizatorului*.
  - Specificarea *secvenţelor de acţiuni* pe baza scenariilor utilizator.
  - Crearea *modelului comportamental* al interfeţei.
  - Definirea *obiectelor de interfaţă* şi a *mecanismelor de control.*
  - *Revizuirea* şi refacerea (dacă e cazul) design-ului interfeţei.

PRINCIPII PRACTICE ALE PROIECTĂRII SOFTWARE-lui

SET GENERIC DE ACTIVITĂŢI (4)

- Proiectarea componentelor:
  - Specificarea tuturor algoritmilor la un nivel relativ scăzut de abstractizare.
  - Rafinarea interfeţelor fiecărei componente.
  - Definirea structurilor de date la nivelul componentelor.
  - Revizuirea modelului componentelor şi corectarea tuturor erorilor.
- Dezvoltarea unui model de instalare (repartizare a artefactelor).

Bibliografie

Roger S. Pressman, **Software Engineering. A Practitioner's Approach**, ed.7, McGraw-Hill, 2010.

capitolele 6 - 14

Fairbanks, G. **Just Enough Software Architecture, A Risk Driven Approach**, Marshall&Brainerd, 2010

CONTROLUL CONCURENȚEI în DOMENIUL BAZELOR DE DATE

- Scop
  - Asigurarea faptului că operaţiile cu bazele de date sunt executate în manieră protejată

- Două forme principale de control al concurenţei:

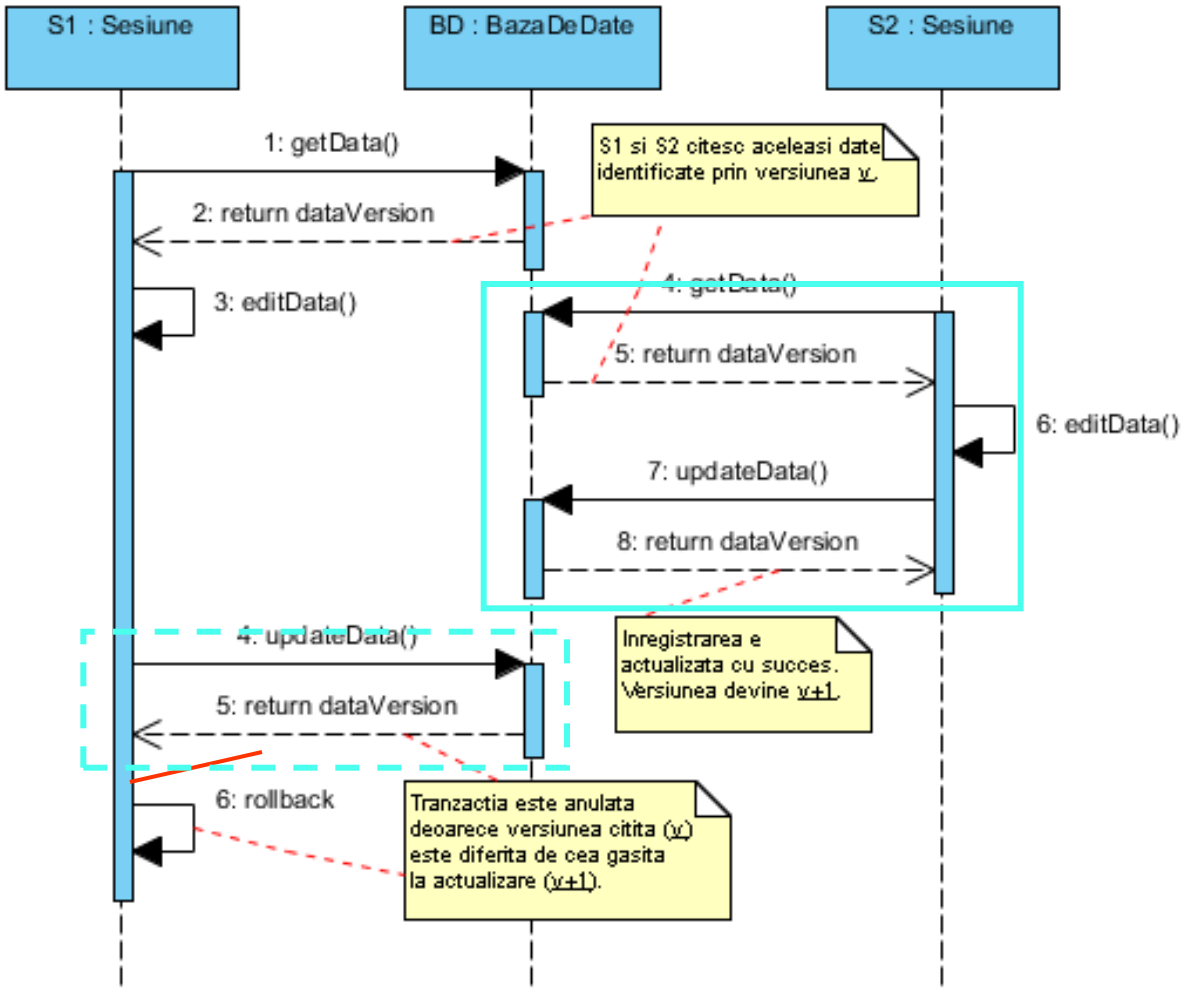
  - Lock optimist
    - Schemă de detectare a conflictului

  - Lock pesimist
    - Schemă de prevenire a conflictului
    - Poate conduce la blocaje permanente (deadlock)

CONTROLUL CONCURENȚEI în DOMENIUL BAZELOR DE DATE

Exemplu: Șablonul de control optimist al concurenței

- Exemplu de modelare a schemei de lock optimist.



Limitele unei tranzacţii *la nivelul logicii aplicaţiei.*

Limitele unei tranzacţii *la nivelul bazei de date.*