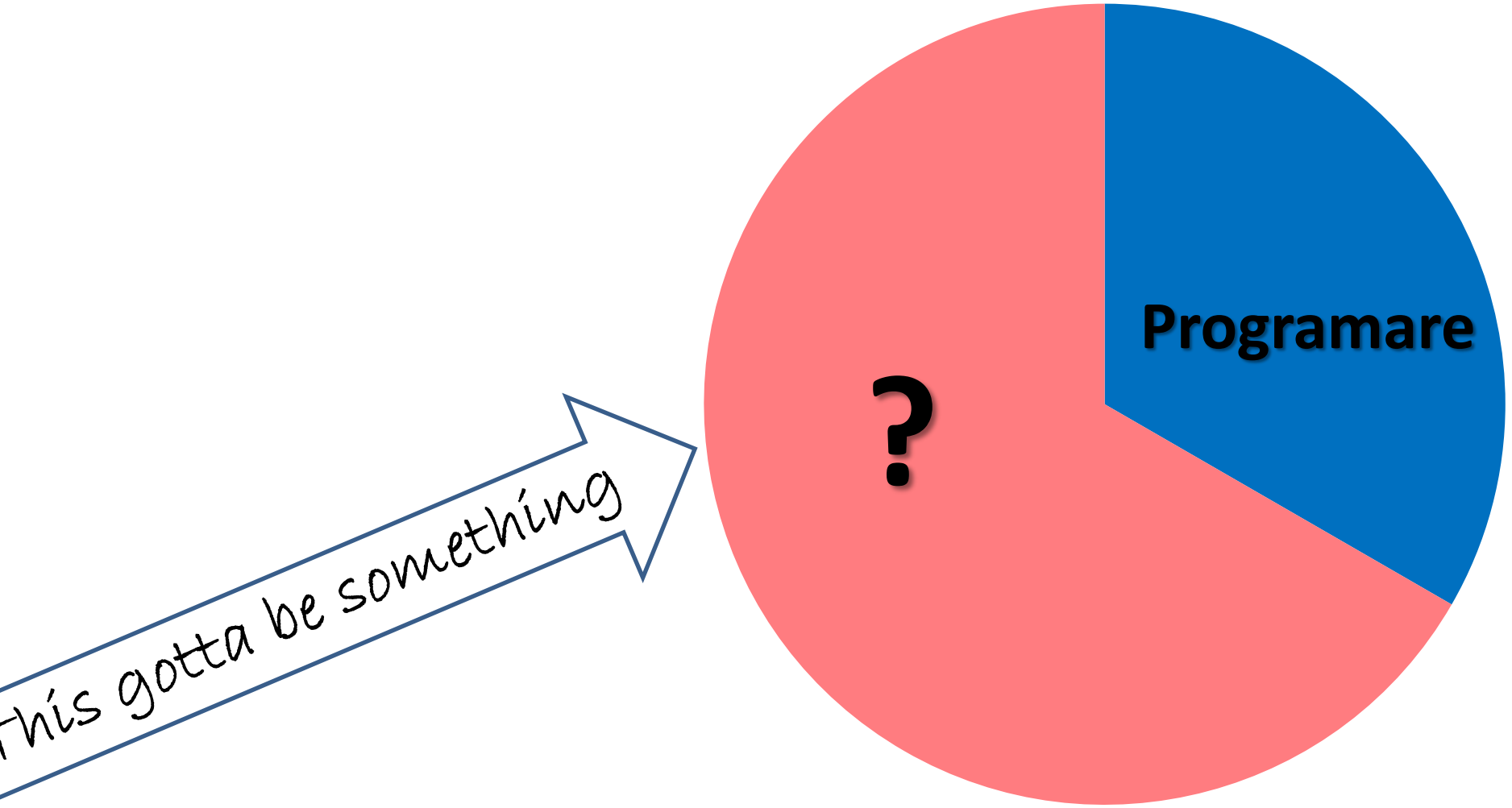
Ingineria Programelor

Prof. dr. ing. Moldoveanu Alin

alin.moldoveanu@cs.pub.ro

Who am I

DISTRIBUTIA RESURSELOR IN DEZVOLTAREA SOFTWARE



CURS RELAXAT. INTERACTIV

De ce învăț ?
Why do I learn ?
Por que eu aprendo?
我为什么学习？

Placere

Pasiune

Perfectionare

~~**pt nota 10**~~

~~**pt 5**~~


honor

ACCENT PE ACTIVITATILE TEHNICE

RELEVANT PENTRU:

DEZVOLTATORI
MANAGERI

CUPRINS

- Introducere:
  - Definiție
  - Scurtă istorie
  - Activități in dezvoltarea software (pe scurt)
  - Punctaj
- Modele ale procesului de dezvoltare software
- Analiza si definirea cerințelor.
- Proiectarea: arhitecturala, detaliata, șabloane
- Modelarea. Unified Modelling Language
- Verificarea si validarea
- [Calitățile produselor software]

- Prezentări studenți
- Prezentări specialiști din industrie

Ingineria Programelor
(Software Engineering)
=
?

DEFINITII

- Software =

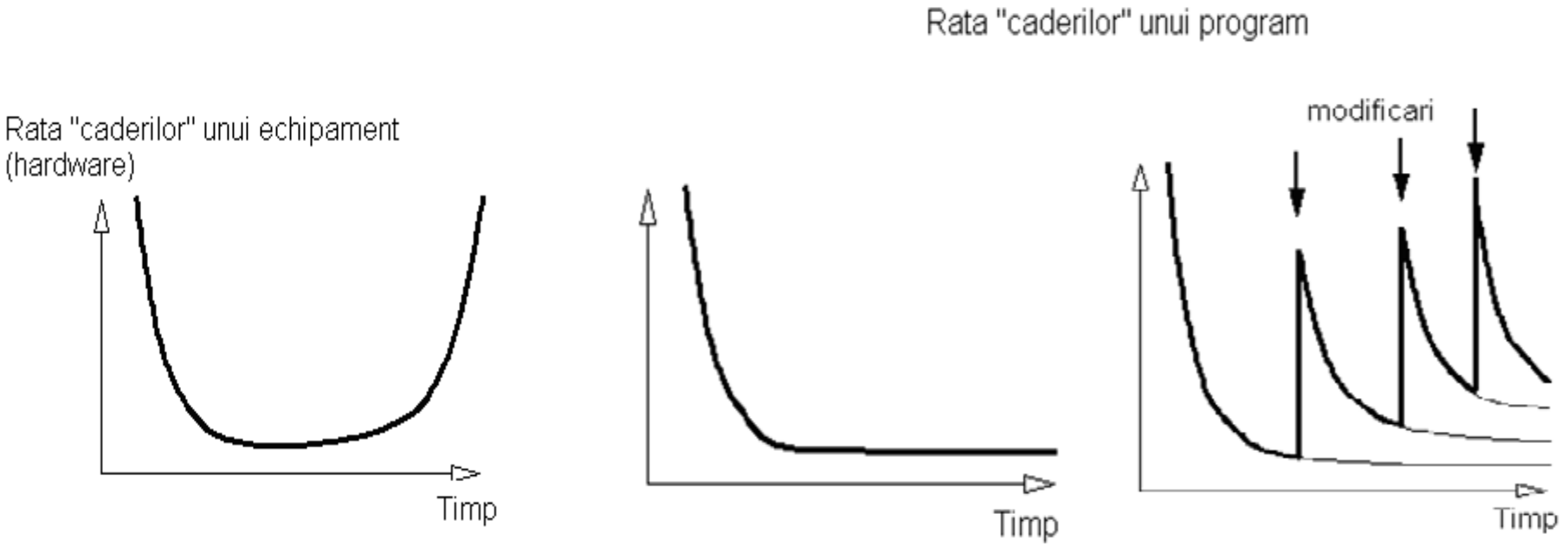
  cod sursă, cod executabil, biblioteci

  +

  documentații (de realizare, de instalare, de utilizare)

SOFTWARE VS. PRODUSE FIZICE

- nu este un produs fizic
- **algoritmi, prelucrări de date/evenimente – nu interacțiuni fizice**
- dezvoltat, nu fabricat; nu exista un proces de fabricație software
- programele nu pot fi “asamblate” integral din componente - trebuie sa răspundă la cerințe specifice, particulare
- nu “îmbătrânește” ci evoluează prin mentenanță.



SCURTĂ ISTORIE

- Anii ’50-’60: începutul dezvoltării de software; primele limbaje de programare si sisteme de operare
- 1965-1985: perioada “crizei” software: multe proiecte eșuate și probleme grave legate de dezvoltarea de software
- 1970-1990: silver bullets:
  - Tools
  - Discipline
  - Formal methods
  - Process
  - Professionalism
- 1990-2000: Internet boom, tehnologii noi, sisteme complexe, distribuite
- Direcții actuale:
  - Aspects
  - Agile
  - Experimental
  - Model-driven
  - Software product lines

CRIZA SOFTWARE-ULUI (1)

- Stimul: Dezvoltarea permanentă a hardware-ului și utilizarea crescută a calculatoarelor

  ---> necesitatea dezvoltării rapide de sisteme software complexe și de calitate

- Problemele (anii 70-80)
  - Proiecte care depășeau timpul sau bugetul alocat dezvoltării
  - Software nefinalizat
  - Erori majore/critice
  - Software de calitate scăzută;
  - Software ineficient
  - Software care nu satisfăcea cerințele
  - Costuri și dificultăți foarte mari pentru mentenanță

CRIZA SOFTWARE-ULUI (2)

Conștientizare

- metodele de dezvoltare erau inadecvate cerintelor
- software-ul si dezvoltarea lui au trasaturi diferite de alte produse si discipline ingineresti
- efortul de dezvoltare creste mai mult decat liniar cu la dimensiunea programului.
- un program nu este o entitate statica, el evolueaza in timp datorita schimbarii cerintelor si a mediului de utilizare.
- codul trebuie sa poata fi inteles si adaptat de persoane diferite de cele care le-au dezvoltat.
- G. Booch: "*building and maintaining software is hard and getting harder; building quality software in a repeatable and predictable is harder still*"

=> SOFTWARE ENGINEERING (IP)

- Dezvoltarea de Sw începe sa fie văzută ca o industrie distinctă
- Software-ul tratat ca un produs ingineresc specific
- Disciplină cadru pentru construirea de software: Software Engineering (Ingineria programelor)
- Definirea de tehnici de producție justificate de teorie și practica

Software Engineering (cf. standardului IEEE 1993)

"Aplicarea unei abordări

sistematice, disciplinate si măsurabile in

dezvoltarea, operarea si întreținerea software-ului"

SOLUTII IN IP

Pentru depășirea crizei software au fost realizate:

- **Clarificarea activităților specifice** (dezvoltării de SW)
- **Modele** (ghiduri) **de dezvoltare**
- **Metode si limbaje de analiza si proiectare** (a progr.)
- Limbaje de programare
- Principii si tehnici de reutilizare a componentelor SW.
- Medii de dezvoltare integrate (IDE)
- Generatoare de aplicații
- Medii de testare
- etc

RATA DE SUCCES / EȘEC

2015 CHAOS Report

	2011	2012	2013	2014	2015
SUCCESS	29%	27%	31%	28%	29%
CHALLANGED	49%	56%	50%	55%	52%
FAILED	22%	17%	19%	17%	19%

Relax---

You deserve a break

ACTIVITATI INTR-UN PROIECT SOFTWARE

- **Activitati tehnice:**
  - **Analiza si specificarea cerintelor**
  - **Proiectarea**
  - **Implementarea unitatilor program (modulelor)**
  - **Integrarea**
  - **Testarea de sistem**
  - **Testarea de acceptare**
  - **Intretinerea si operarea**
- Activitati de management al proiectului
- Activitati de asigurare a calitatii

ANALIZA SI SPECIFICAREA CERINTELOR

- Cerințele utilizator
  - Descriu punctul de vedere al utilizatorilor: CE doresc viitorii utilizatori de la viitorul produs.
  - Sunt cerințe de: funcționare, performantă, securitate, interfață utilizator, alte interfețe, etc.
  - Definite în: **URD** (User Requirements Document) - Documentul Cerințelor Utilizator, numit uneori Documentul de definiție a sistemului
  - URD → parte din contractul cu clientul
  - URD → baza pentru testele de acceptare.

- Cerințele software
  - Sunt mai exacte, tehnice; se stabilesc prin analiza, reformularea mai tehnică a URD.
  - Sunt specificate în **SRD** (Software Requirements Document); uneori, pentru sisteme foarte complexe, cu componente hw si sw, se creează și System Requirements Document
  - Se folosesc pentru clarificarea și planificarea dezvoltării
  - SRD → baza pentru testele de sistem

Cerințele (URD, SRD) sunt "abstracte" – independente de implementare
(exceptând anumite constrângeri).

PROIECTAREA

- Proiectarea arhitecturala
  - Se gândește arhitectura sistemului care va implementa cerințele:
    - Tehnologii
    - Subsisteme (module principale)
  - Se alege soluția optimă dintre alternativele posibile
  - Toate cerințele trebuie să fie acoperite de arhitectură
  - Rezultat: ADD (Architectural Design Document): subsistemele, rolul și interfețele fiecărui modul, modul de cooperare între module.
  - ADD se folosește pentru specificarea testelor de integrare
- Proiectarea de detaliu
  - Proiectări individuale și descompuneri succesive ale modulelor în module din ce în ce mai mici, până la nivelul unităților de implementare (în funcție de proiect)
  - Rezultat: DDD (Detailed Design Document): rolul și interfețele fiecărui modul/submodul, modul de cooperare între module.
  - DDD se folosește pentru definirea testelor unitare

IMPLEMENTAREA

= codificarea + testarea unitara (individuala) a modulelor
(definite de DDD)

- cea mai bine stapanită și “utilată” dintre activități:
  - limbaje de programare, compilatoare, debuggere
  - IDE (medii de dezvoltare integrate)
  - unelte software pentru testarea unitara

~ 15-20% din efortul total de dezvoltare a unui program.

INTEGRAREA

- Modulele care au fost testate independent sunt integrate treptat in subsisteme, pana la nivel de sistem.
- Se testeaza comunicarea si interactiunea intre componentele integrate.

TESTAREA

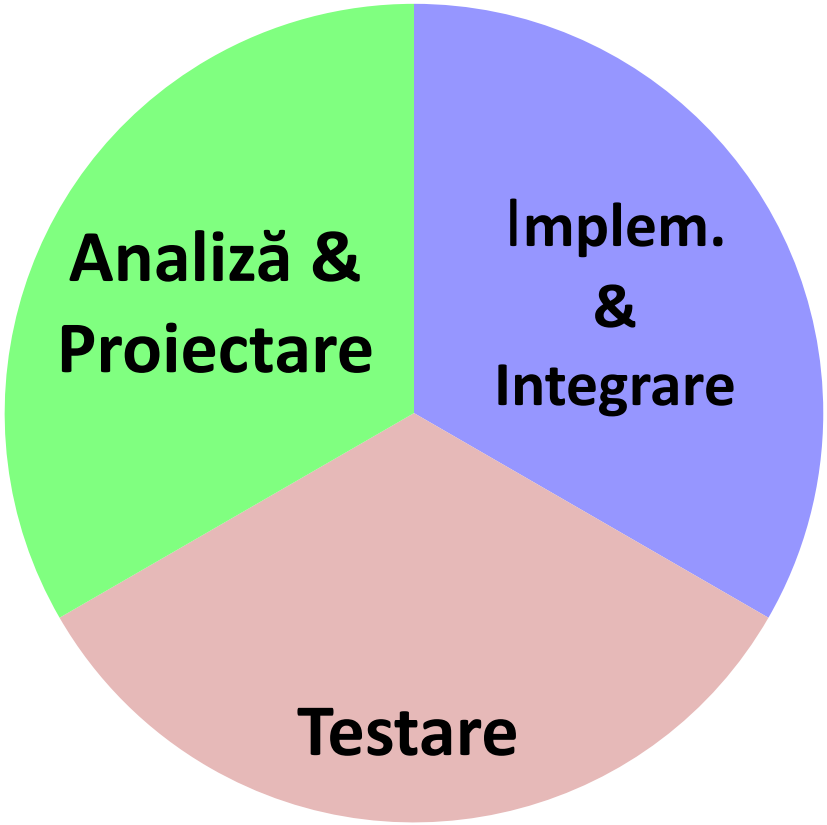
- Teste unitare
- Teste de integrare
- Testarea de sistem
  - Sistemul satisface SRD ?
  - Efectuata in interiorul companiei dezvoltatoare
- Testarea de acceptare
  - Sistemul satisface URD ?
  - Se efectueaza de o echipa de testare care include si clientul
  - Testare alfa/beta

UTILIZAREA SI INTRETINEREA

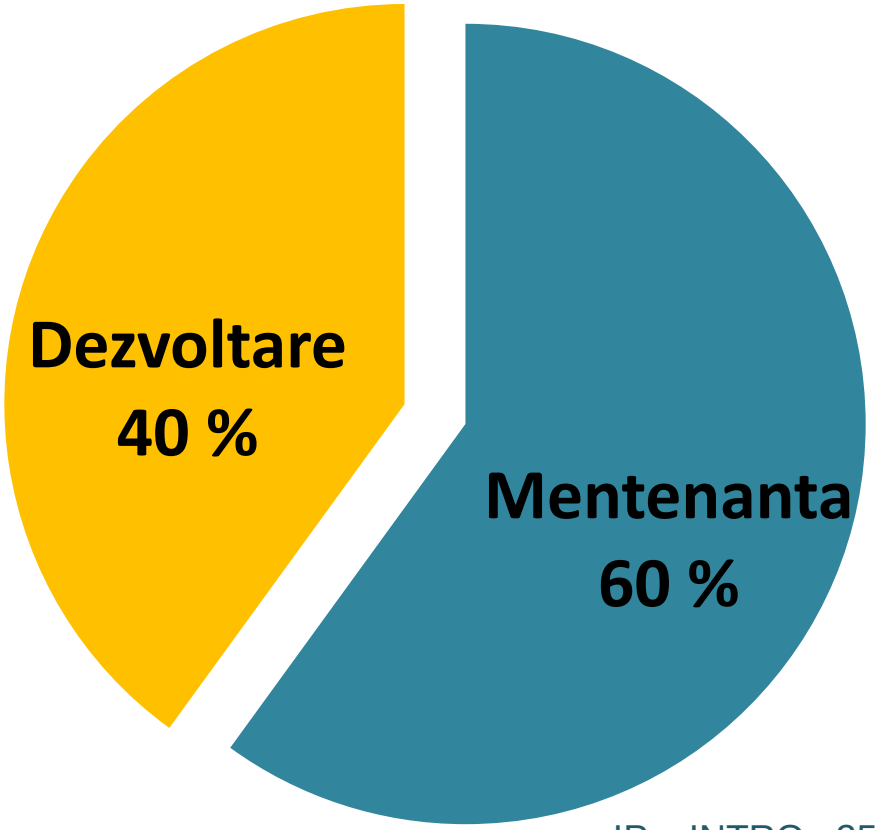
- Utilizarea efectiva a software-ului in mediul real
- Modificari:
  - Corectie bug-uri descoperite
  - Imbunatatire functii existente
  - Adaptare la noi tehnologii
  - Adaugare noi functii
- Intretinerea poate fi efectuata de alti dezvoltatori decat cei initiali

DISTRIBUTIA EFORTULUI

Activitati



Dezvoltare / Mentenanta



ASIGURAREA CALITATII

- Scop: asigurarea cerintelor tehnice si a standardelor de calitate pentru:
  - in procesul de dezvoltare
  - produsul final
- Activitati:
  - Alegerea metodelor si a standardelor de specificare, proiectare si implementare
  - Revizii, pe tot parcursul procesului de dezvoltare
  - Definirea strategiilor de testare
  - Definirea metodelor de documentare
  - Definirea metricilor de evaluare a produselor si a instrumentelor de masurare

MANAGEMENT

- Scrierea propunerii pentru obținerea proiectului
- Estimări asupra necesarului de resurse
- Stabilirea și planificarea etapelor
- Selecția și evaluarea personalului
- Monitorizarea dezvoltării
- Identificarea si monitorizarea riscurilor
- (re-)alocări de resurse
- Revizii
- Rapoarte

RISCURILE UNUI PROIECT SOFTWARE

- factori de experienta:
  - a managerului
  - a echipei
  - a organizatiei
- factori de planificare
  - alegerea modului de dezvoltare
  - estimarea resurselor umane
  - a perioadelor de timp pentru diferite activitati
  - definirea responsabilitatilor
- factori tehnologici:
  - noutatea tehnologica
  - metodele de dezvoltare
  - instrumentele de dezvoltare
- factori externi:
  - calitatea specificatiei cerintelor
  - stabilitata cerintelor
  - stabilitatea si disponibilitatea altor factori de care depinde proiectul

LECTURI RECOMANDATE

- http://www.cs.st-andrews.ac.uk/~ifs/Books/SE7/Presentations/PDF/ch1.pdf
- http://www.baz.com/kjordan/swse625/intro.html
- http://www.stsc.hill.af.mil/resources/tech_docs/GISE.DOC
- Wikipedia:
  - Software Engineering
  - User requirements document
  - Software Requirements Specification

PUNCTAJ

- Proiect: dezvoltare software, in echipa (3-4)
  - Analiza (2 sapt)
  - Proiectare (2 sapt)
  - 4 iteratii de implementare (4 x 2 sapt)
  - Pitch (1 sapt)
- Laborator (UML, Design Patterns)
- Prezentari la curs (optionale, in limita numarului de locuri FCFS)
  - subiect la alegere (din lista, sau propus)
  - In echipe de 3
- Prezenta
- Extemporale (bonus-uri)
- Examen
  - Sinteza
  - Probleme
  - Grila

Proiect+Lab	**5.5+0.5** (+1 bonus)	**3**
Prezentare	**1.5**	
Prezenta	**1.3**	**0.6**
Examen	**4**	**2**
	12.8	**6.7**

Modele de Dezvoltare Software

Ciclul de viata al unui program

Software life cycle

Ciclu de viață = Întreaga secvență de etape din existenta produsului software (dezvoltare + întreținere)

(caracter repetitiv)
Întreținerea ===> reluare activități de dezvoltare

Mod de descriere:

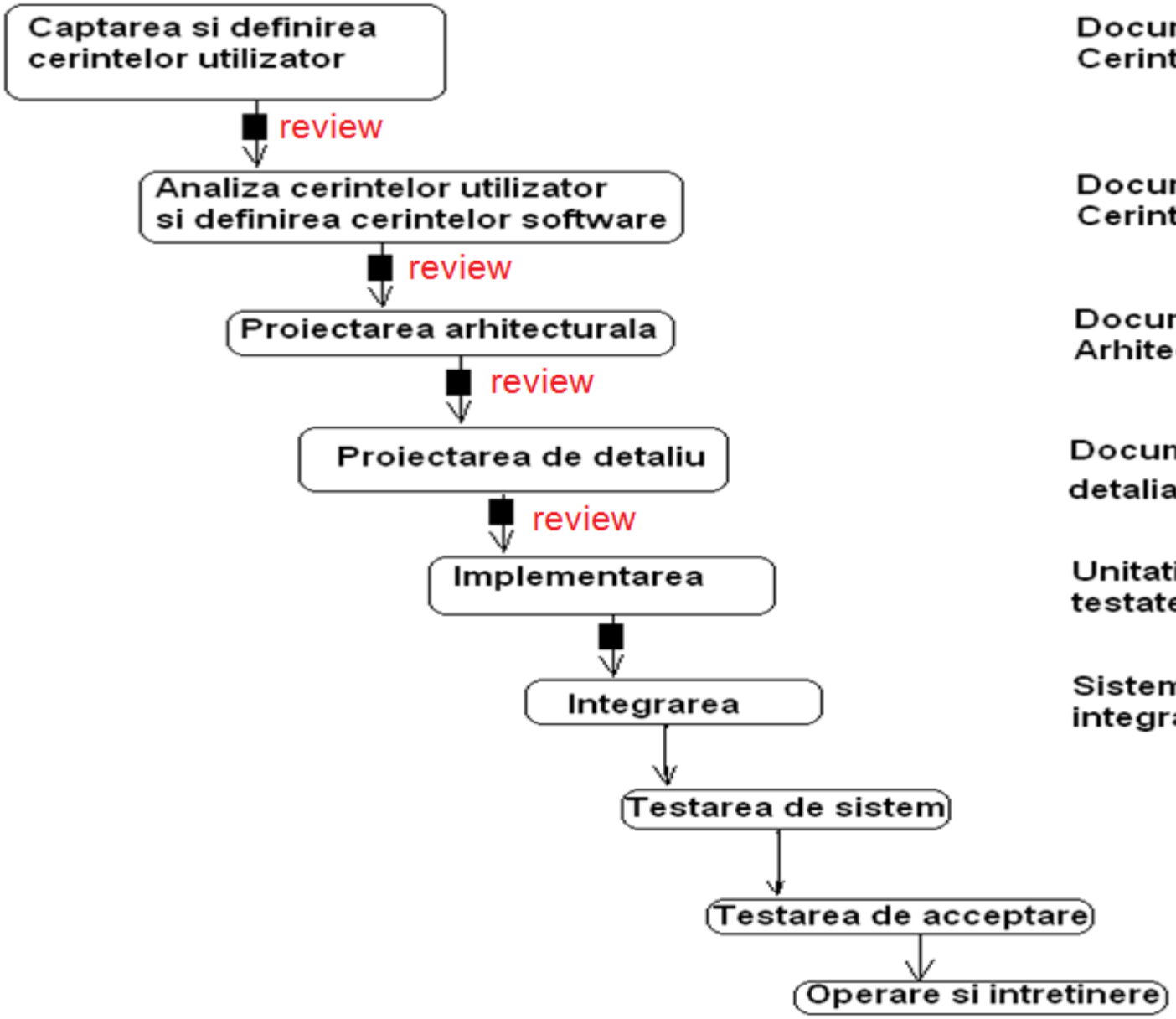
- Etape:
  - Activități specifice
  - Produsele rezultate
- Relațiile temporale și funcționale între etape
- Reguli și principii

Modele ale ciclului de viata software

Software development life cycle models /
Software development process models

- **Modelul cascada** (Waterfall model)
- **Modelul in V** (Vmodel)
- **Modelul ESA** (European Space Agency)
- **Modelul iterativ si incremental**
- **Dezvoltarea “agila”** (Agile development)
- **Dezvoltarea pe baza de prototip** (Prototyping)
- **Modelul in spirala** (Spiral model)

Modelul Cascada

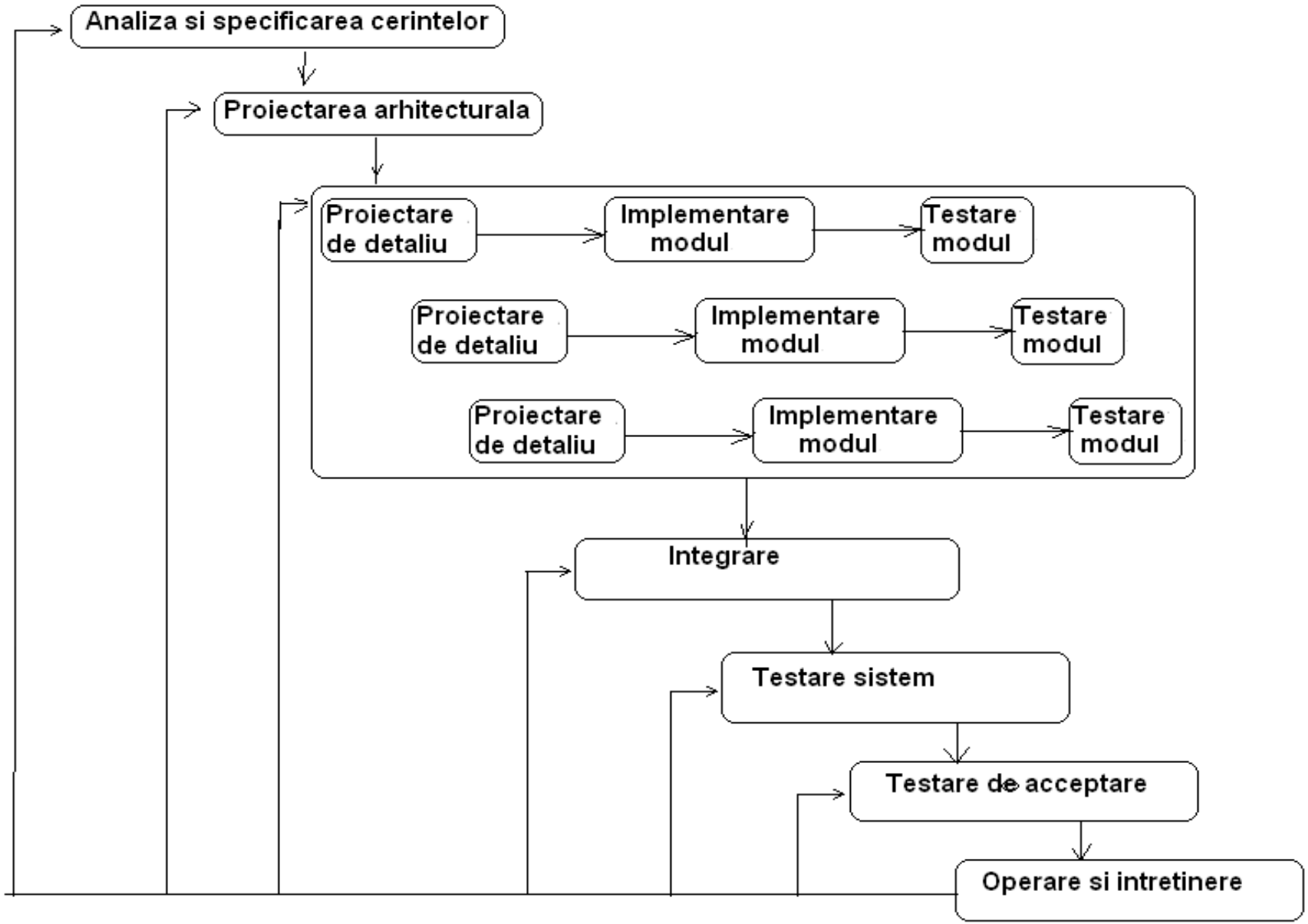


Modelul Cascada

- **Fiecare Faza:**
  - Corespunde unei activitati
  - Trebuie sa se termine la o anumita data
  - Producere anumite documente sau programe

- **Tranzitii**
  - Rezultatele fazelor sunt supuse unor revizii aprofundate
  - Nu se trece la faza urmatoare decat atunci cand sunt considerate satisfacatoare.
  - Sunt descurajate tranzitiile la faze anterioare

Modelul Cascada - rafinare



Modelul Cascada – avantaje/dezav.

- **Avantaje:**
  - Modelul este usor de inteles si pus in practica
  - Sistemul este bine documentat
  - Permite un bun management al proiectului:
    - planificarea resurselor umane pe etape
    - estimari de cost / durata mai exacte
- **Dezavantaje:**
  - Executabilul este disponibil tarziu, dupa integrare (pana atunci s-au produs numai documente)
  - Exista numai feedback local, la tranzitiile intre faze.
  - Multe erori sunt descoperite tarziu: cost crescut de rezolvare !
  - Toate riscurile sunt incluse intr-un singur ciclu de dezvoltare
  - Adaptabilitate mica:
    - La riscuri
    - La modificari de cerinte
    - La optimizarea costurilor dezvoltarii prin intrepatrunderea/suprapunerea etapelor
- **Adecvat pentru:**
  - proiectele in care cerintele sunt bine intelese de la inceput si nu se modifica pe parcursul procesului de dezvoltare.
  - Nu exista riscuri tehnologice / organizationale /de personal / etc.
- **Utilizare practica:**
  - Descris si deja criticat in 1970
  - Experienta ultimelor decenii a demonstrat ca modelul este valoros.
  - Este utilizat si in prezent de multe organizatii mari.

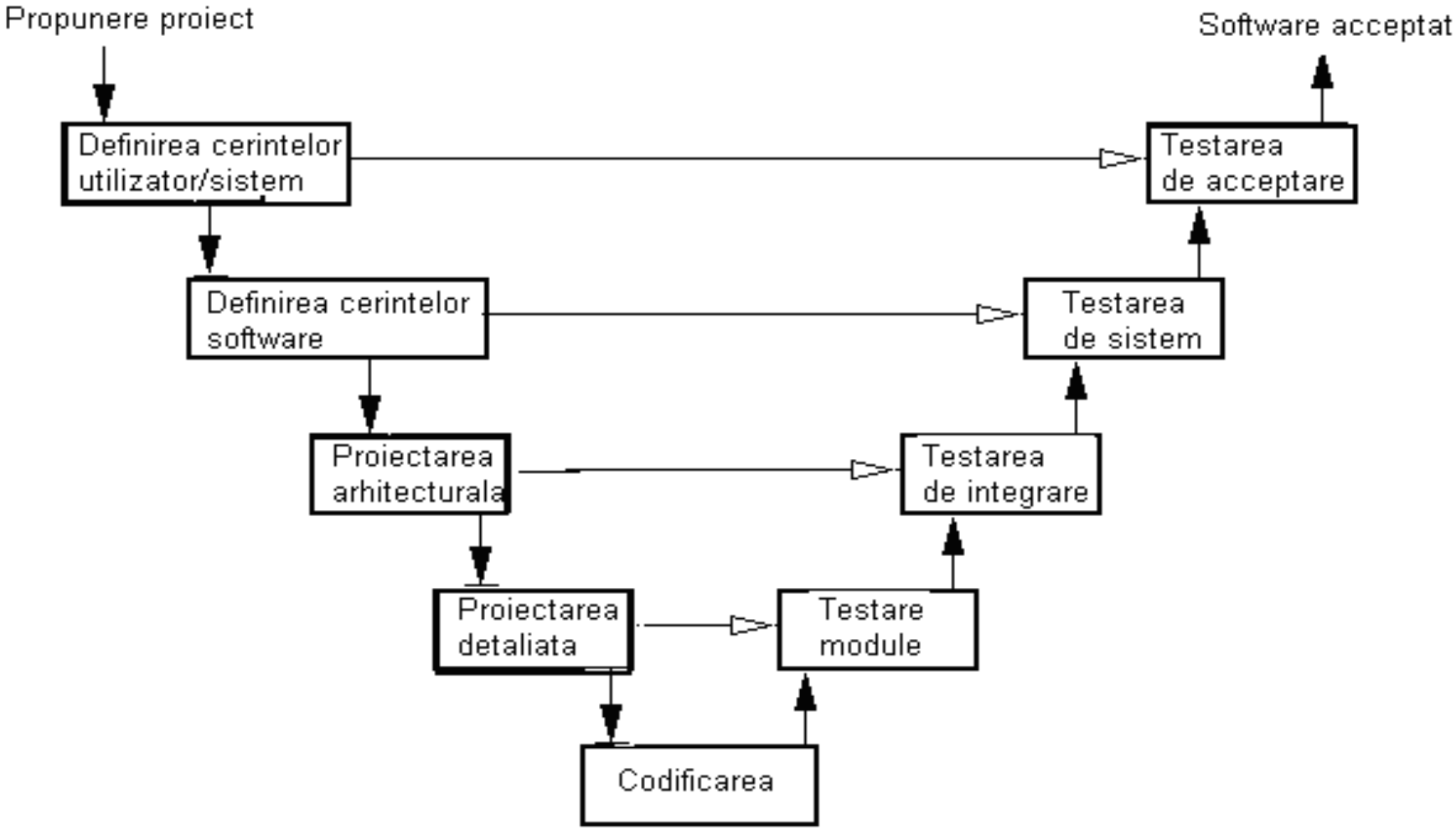
“not only is it impossible in practice to define the requirements accurately before construction starts, it is wrong in principle because the process of developing the application changes the users' understanding of what is desirable and possible and thus changes the requirements. Furthermore, there is widespread agreement that the Waterfall is particularly damaging for interactive applications”

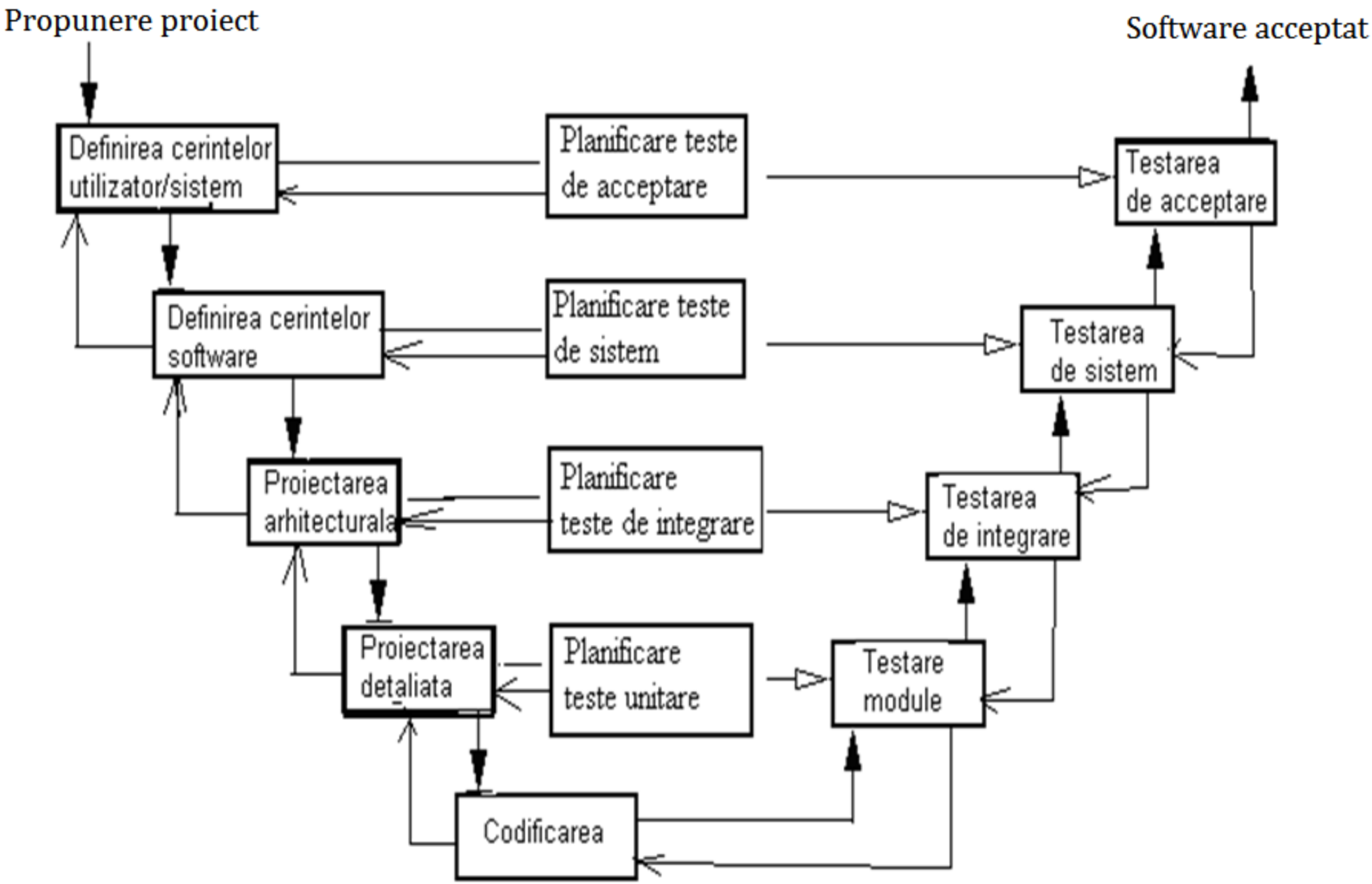
“Waterfall (and V) satisfy the needs of project managers, lawyers and accountants rather than software developers or users.”

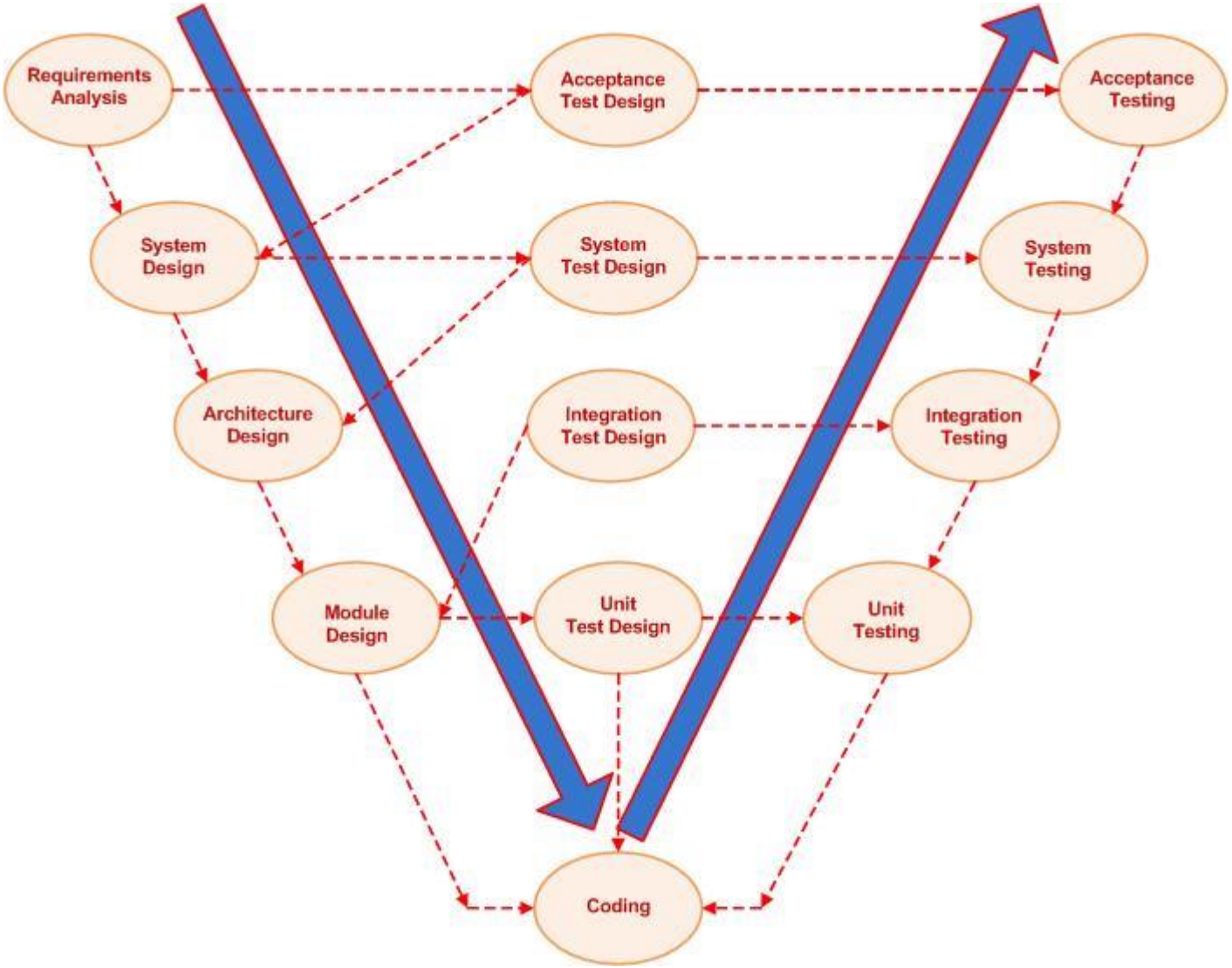
"If the Waterfall model were wrong, we would stop arguing over it. Though the Waterfall model may not describe the whole truth, it describes an interesting structure that occurs in many well-defined projects and it will continue to describe this truth for a long time to come. I expect the Waterfall model will live on for the next one hundred years and more".

Modelul in V

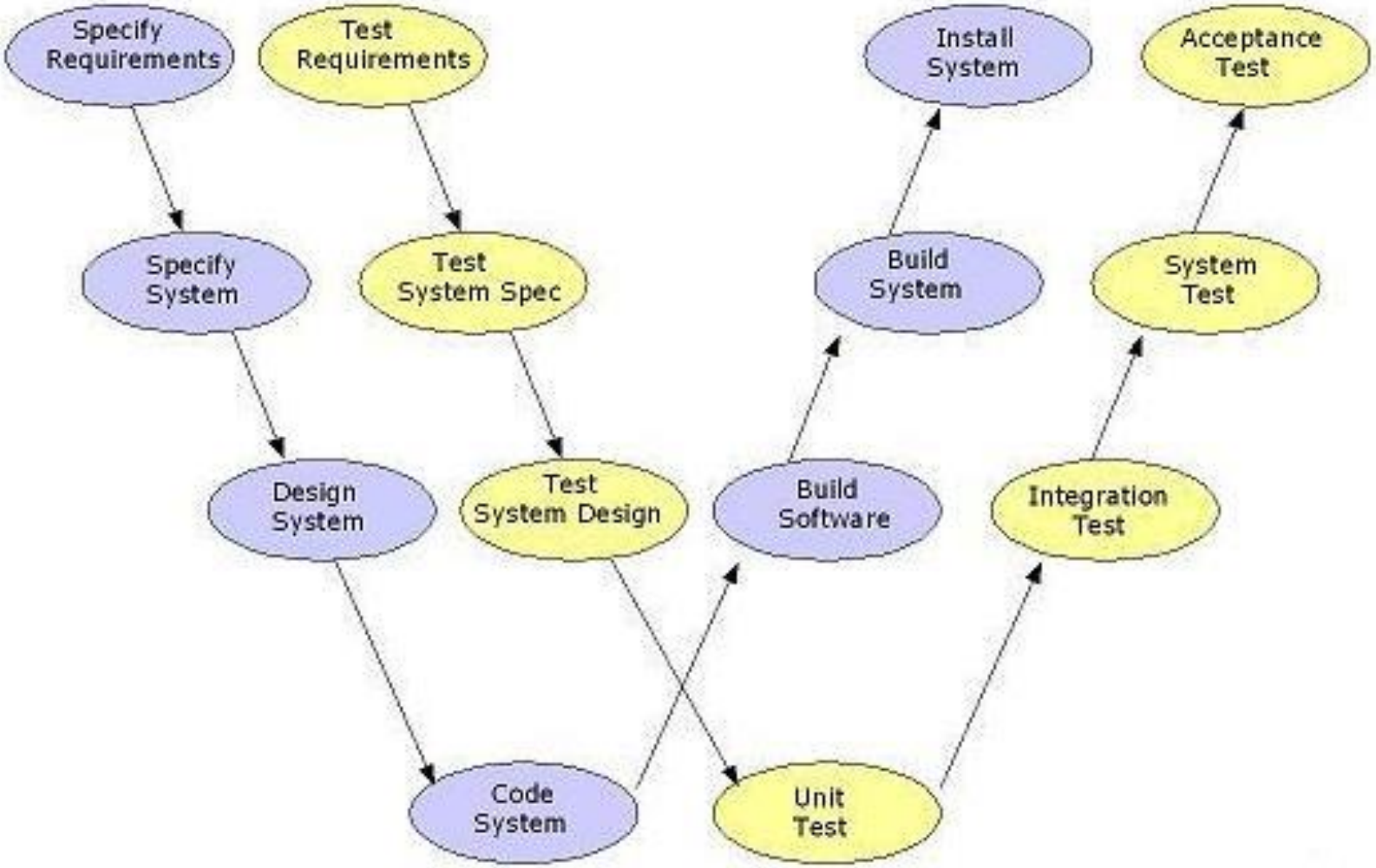
- **Este o varianta a modelului cascada, care include elaborarea planurilor de test in sau in paralel cu fazele de specificare/proiectare corespunzatoare**
- **Variante, reprezentari si interpretari, in diferitele scoli de SwE**
- **Avantaje:**
  - abordare disciplinata
  - promovează activitati meticuloase, calitative
  - creaza documentația necesară pentru produse software stabile
  - larg adoptat in eHealth (cerinte foarte mari de siguranta si fiabilitate)
- **Critici:**
  - Este prea simplu pentru a reflecta natura dezvoltarii de software;
  - poate induce un fals sentiment de securitate;
  - reflecta mai curand cerintele managerilor decat ale dezvoltatorilor si utilizatorilor
  - Nu este flexibil; promoveaza o vedere liniara asupra dezvoltarii de software; nu raspunde bine la schimbari si riscuri
  - Incurajeaza metode ineficiente de testare
  - lipsa unui model V unanim acceptat si inteles


Propunere proiect
Definirea cerintelor utilizator/sistem
Definirea cerintelor software
Proiectarea arhitecturala
Proiectarea detaliata
Codificarea
Testare module
Testarea de integrare
Testarea de sistem
Testarea de acceptare
Software acceptat

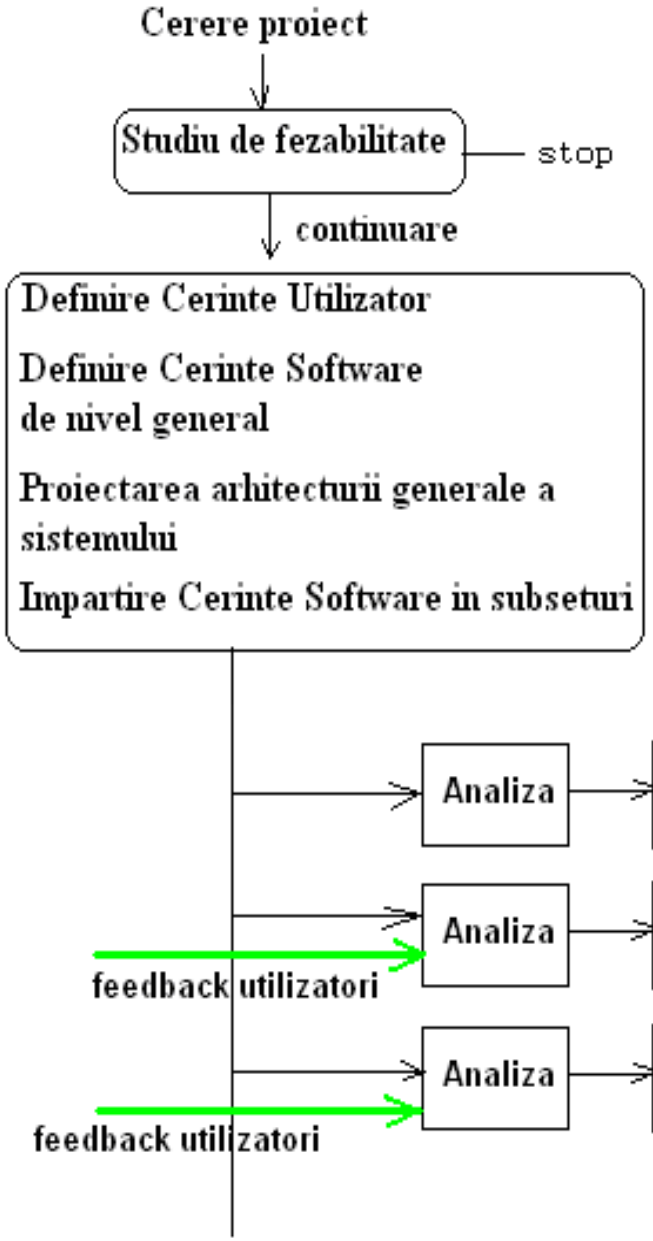

Propunere proiect
Software acceptat
Definirea cerintelor utilizator/sistem
Planificare teste de acceptare
Testarea de acceptare
Definirea cerintelor software
Planificare teste de sistem
Testarea de sistem
Proiectarea arhitecturala
Planificare teste de integrare
Testarea de integrare
Proiectarea detaliata
Planificare teste unitare
Testare module
Codificarea


Requirements Analysis
Acceptance Test Design
Acceptance Testing
System Design
System Test Design
System Testing
Architecture Design
Integration Test Design
Integration Testing
Module Design
Unit Test Design
Unit Testing
Coding

“W”


Specify Requirements
Test Requirements
Install System
Acceptance Test
Specify System
Test System Spec
Build System
System Test
Design System
Test System Design
Build Software
Integration Test
Code System
Unit Test

Modelul Incremental



- Produsul este dezvoltat in mai multe iteratii
- Fiecare iteratie este un mini-ciclu cascada
- Fiecare iteratie se incheie cu un executabil

Modelul Incremental – av/dezav.

- **Avantaje:**
  - In fiecare etapa este livrat un produs executabil, care satisface o parte din cerintele utilizator.
  - Prototipurile sunt livrate clientului/utilizatorilor.
  - Feedback-ul utilizatorilor este distribuit pe intreg parcursul dezvoltarii.
  - In cazul aparitiei unor schimbari in cerinte acestea pot fi incoporate in urmatorul prototip.
- **Dezavantaje**
  - Arhitectura initiala a programului poate fi degradata
  - Erorile de proiectare sunt mai greu de eliminat.
  - Clientul poate vedea ce se poate face si poate cere mai mult!
  - Abordarea incrementala se poate transforma usor in « codifica si repara « (« build and fix »).

  Sunt importante planificarea si controlul atent pt a evita dezav.
- **Adecvat pentru:**
  - proiectele in care este necesara reducerea riscurilor de specificatii / tehnologice / organizationale /de personal / etc.
- **Utilizare practica:**
  - Modelul valoros, utilizat in multe proiecte

Metodele “agile” - principii

Engleza: Agile software development

2001: Agile Manifesto

- Prioritatea maxima o are satisfacerea clientului prin furnizarea rapida si continua de software functional
- Principala masura a progresului este un software functional
- Cererile de schimbare sunt solutionate chiar si in fazele avansate ale proiectului
- Software-ul functional trebuie sa fie livrat la intervale de timp scurte (ex. saptamanal)
- Dezvoltatorii trebuie sa conlucreze cu expertii din domeniul aplicatiei pe toata durata proiectului
- Cea mai eficace metoda de transmitere a informatiilor este comunicarea directa in echipa
- Echipa trebuie sa se poata organiza singura, sa-si analizeze performantele si sa se adapteze

Metodele “agile” - practica

www.agilealliance.org

- **Analiza si proiectare sumare**
  - Identificarea obiectivelor generale
  - Arhitectura generala
  - Împărțirea sarcinilor in incremente mici (stories / backlog items), ce vor fi tratate ulterior in iterații
- Software-ul este elaborat in **iteratii foarte scurte** - timeboxes / sprints
  - Durata fixa (1-4 saptamani). Nu se prelungeste. De regula toate iteratiile sunt egale
  - Iteratie = lucrul in echipa la toate activitatile necesare livrarii mini-incrementului: planificare, analiza cerintelor, proiectare, codificare, testare, documentare.
  - Fiecare iteratie trebuie sa se incheie cu un progres vizibil –un prototip mai bun
  - la sfarsitul fiecarei iteratii se re-evalueaza prioritatile.
- **Clientul**:
  - stabileste si prioritizeaza cerintele
  - poate schimba cerintele sau prioritatea lor in orice moment
- **Comunicarea**: directa (intre participantii la dezvoltare), in locul documentelor

Rational Unified Process (RUP)

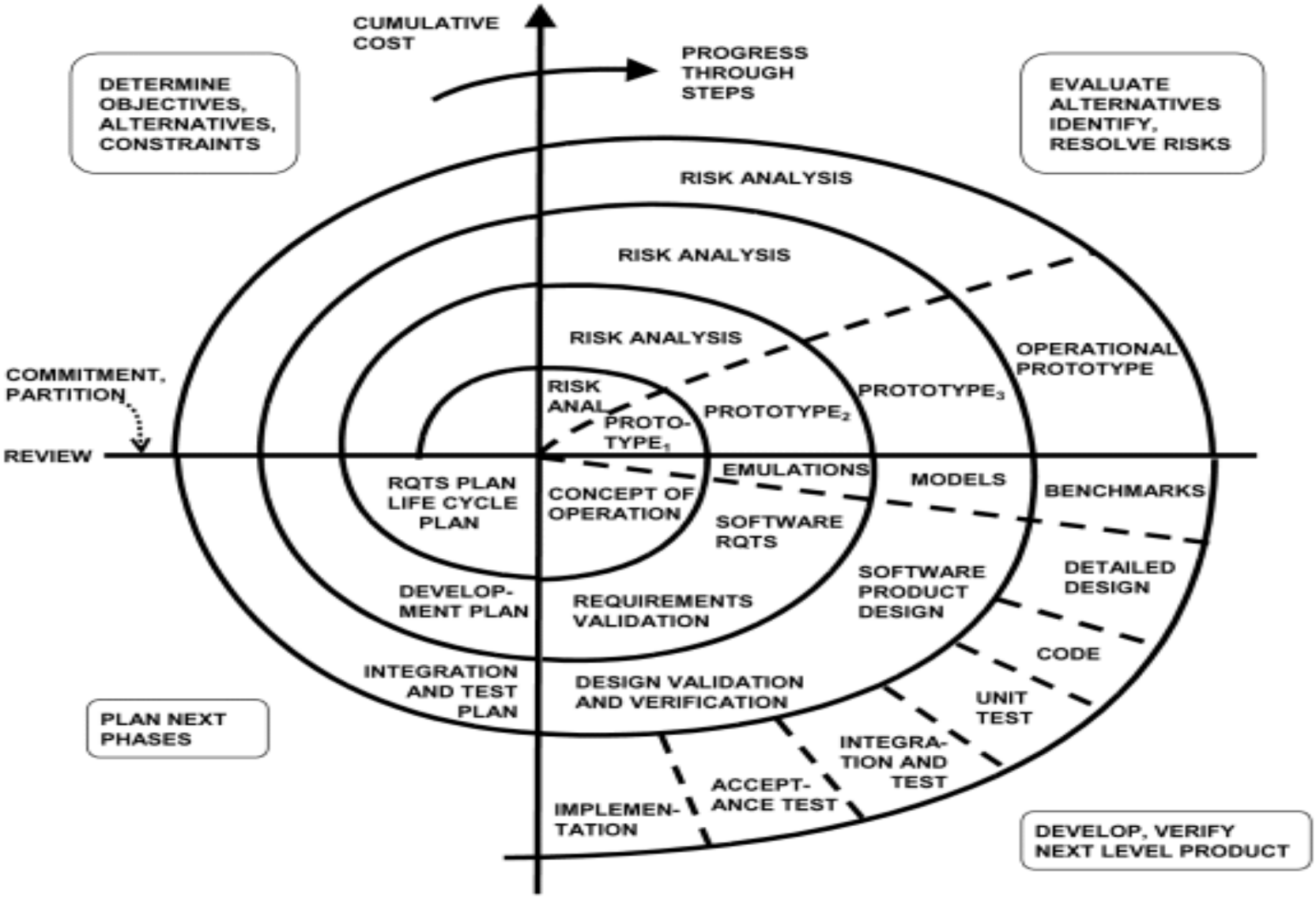
www.rational.com

- **Iterativ**
- ***Dirijat de cazurile de utilizare***
- ***Centrat pe arhitectura***

Modelul spirala

- **Model incremental focalizat pe analiza riscurilor in fiecare etapa**
- **Fiecare ciclu consta din 4 faze:**
  - Determinarea obiectivelor:
    - definirea produsului
    - determinarea scopurilor si a constrangerilor
    - generarea alternativelor
  - Evaluarea alternativelor
    - analiza riscurilor
    - prototiparea
  - Dezvoltarea produsului: proiectarea de detaliu, testarea unitara, integrarea, ….
  - Planificarea urmatorului ciclu: evaluarea de catre client, planificarea dezvoltarii si a livrarii catre client.

Modelul spirala



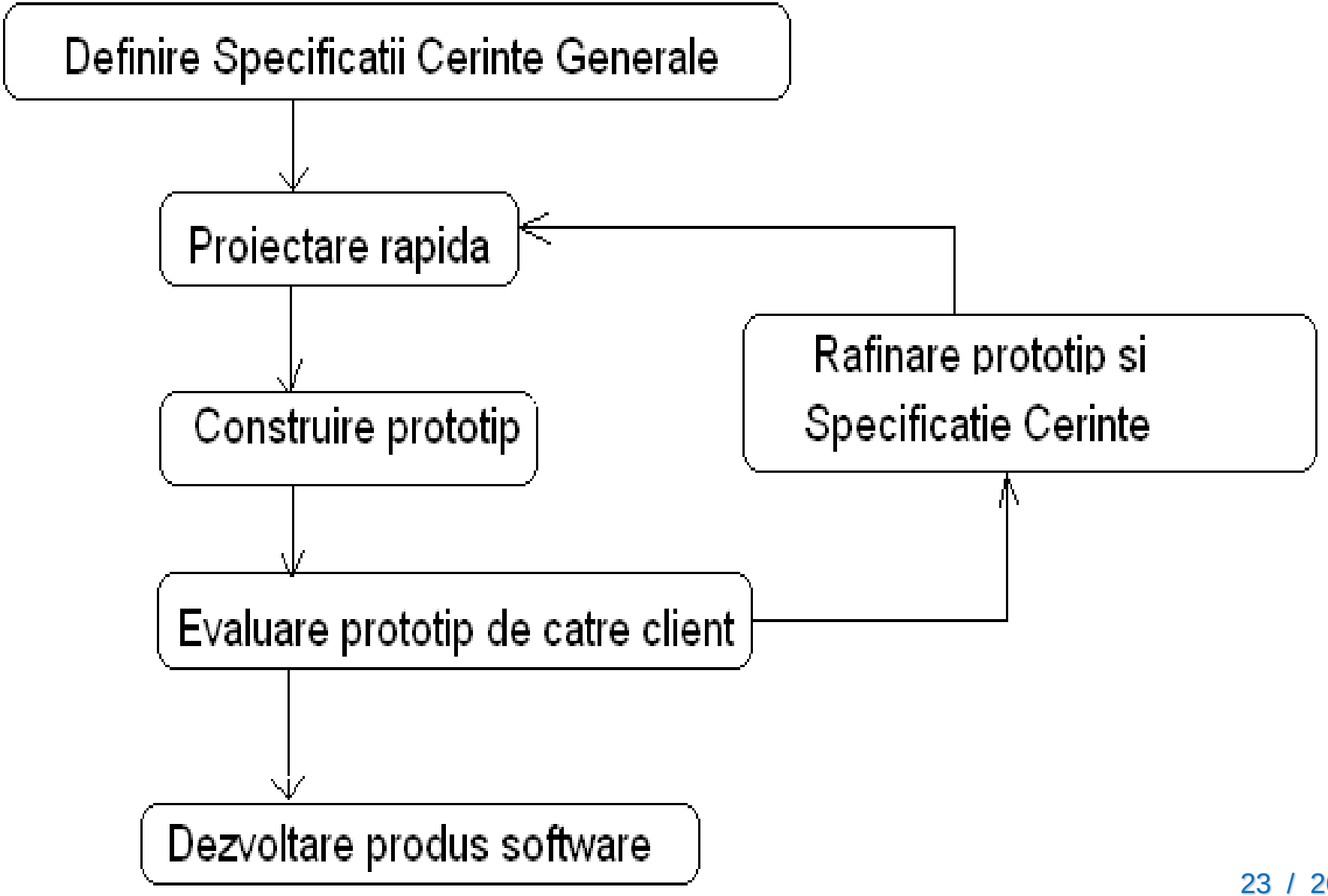
Prototiparea

Prototip = versiune incompleta (partiala) a unui program

Utilizari:

- **analiza: comunicarea cu utilizatorul si extragerea cerintelor**
- **proiectare: eliminarea riscurilor**

Definire Cerinte folosind Prototip



Comparatie intre metode

Agile ***Iterative/Incrementale*** ***Cascada***

<----------|----------|---------->

Adaptive Predictive

- **Metodele predictive:** (accent pe) planificarea in detaliu a activitatilor, in timp
- **Metodele adaptive:** (accent pe) livrare continua, flexibilitate, adaptare la schimbari

Concluzii

- **Fiecare metodă are avantaje si dezavantaje**
- **Nu exista o metoda perfecta pentru toate tipurile de proiecte si organizații**
- **Alegerea trebuie făcută in funcție de proiect si obiectivele organizației**
- **Este necesara înțelegerea tuturor modelelor pt:**
  - Alegerea modelului potrivit
  - Punerea lui in practica
  - Activitatea eficienta in cadrul unei dezvoltări

"Any form of life cycle is a project management structure [artificially] imposed on system development.

To contend that any life cycle scheme, even with variations, can be applied to all system development is either to fly in the face of reality or to assume a life cycle so rudimentary as to be vacuous."

Lecturi recomandate

V model and other interesting things: http://www.clarotesting.com/page11.htm

General SwE: http://www.cs.cmu.edu/~aldrich/courses/413/

RUP: rup_bestpractices.html

ANALIZA

(Procesul de definire a cerintelor)

- **In modelul “Cascada”: se desfasoara in fazele de inceput ale procesului.**
- **Intr-o dezvoltare iterativa si incrementala:**
  - Este initiat la inceputul proiectului software, la un nivel general
  - Se continua si se rafineaza (pentru subseturi de functionalitati) pe intreg parcursul ciclului de viata.

- **Este adaptat la nivelul fiecarei organizatii si in functie de cerintele fiecarui proiect.**

- **Include activitati de:**
  - extragere a cerintelor
  - analiza cerintelor
  - specificarea cerintelor
  - validarea cerintelor

Actorii procesului (stakeholders)

- **Din partea dezvoltatorului:**
  - Analistul/ analistii software
  - Managerul de proiect
  - Alte categorii de ingineri software (designeri, testeri, etc)
- **Din partea clientului**
  - Utilizatori
  - Factori de decizie
  - [Specialisti IT]
- Daca nu este un client anume: **Analisti de marketing**

Analistul trebuie sa:

- coordoneze activitatea de extragere a cerintelor
- sa inteleaga toate cerintele
- sa medieze si “negocieze” intre cerintele diferitilor actori si diversele aspecte ale cerintelor (cerintele tehnice, bugetare, legislative, etc.)

Extragerea cerintelor

(capturarea, descoperirea, achizitia)

- **Este etapa de intelegere a problemei**
- **Se stabileste o lista de cerinte de nivel inalt care reflecta punctul de vedere al diferitelor grupuri de actori asupra sistemului.**
- **Sunt incluse cerinte de: functionare a sistemului, de performanta, de securitate, de interfata utilizator, s.a.**
- **Cerintele sunt exprimate intr-un limbaj specific domeniului aplicatiei, familiar actorilor care participa la extragerea cerintelor.**
- **Rezultatul etapei:**
  - In documentul general de Specificare a Cerintelor
  - intr-un document separat, numit Documentul cerintelor utilizator (URD) sau Documentul de definitie a sistemului.

Analiza cerintelor

- **Clasificarea cerintelor dupa diferite criterii:**
  - Functionale/ne-functionale
  - Cerinte de produs/proces
  - Prioritatea
  - Stabilitatea
- **Rezolvarea conflictelor intre:**
  - cerintele diferitilor actori
  - cerinte functionale si ne-functionale
- **Modelarea conceptuala a sistemului**

Specificarea cerintelor

- **Definitia cerintelor utilizator**
- **Documentul de definitie a cerintelor software**
- **Documentul de definitie a cerintelor de sistem**

Documentul de Definitie a Cerintelor Utilizator (User Requirements Document)

- Defineste cerintele de nivel inalt ale sistemului, din perspectiva utilizatorilor.
- Descrie:
  - cerintele de sistem (sau descrierea mediului de operare)
  - obiectivele generale ale sistemului
  - principalele entitati ale domeniului
  - cerinte operationale
  - constrangeri
  - Scenarii
  - fluxul informational, etc.
- De regula, sta la baza contractului dintre client si furnizor/ echipa de dezvoltare
- Este folosit in testele de acceptare

Documentul cerintelor software (Software Requirements Document)

- Descriere completa a functiilor pe care trebuie sa le realizeze produsul software
- Reflecta punctul de vedere al dezvoltatorului asupra viitorului sistem
- Contine un model logic al sistemului, construit prin aplicarea unei metodologii (e.g. OO)
- Furnizeaza o baza mai amanuntita pentru estimarea costurilor si a planificarii
- Este folosit in testele de verificare si validare de sistem

Documentul de definitie a cerintelor de sistem (System Requirements Document)

- Este necesar atunci cand sistemul include mai multe componente hardware
- Cerinte de sistem:
  - Functiile sistemului in ansamblul sau
  - Configuratia hardware
  - Alocarea functiilor pe componente hardware/software
  - Performantele sistemului
  - Cerintele de siguranta in functionare
  - Interfata utilizator
  - Instructiuni de punere in functiune
  - Comunicarea cu sisteme externe

Validarea cerintelor

- **Se verifica daca cerintele:**
  - au fost bine intelese de analist
  - sunt clare
  - sunt consistente
  - sunt complete (??)
  - satisfac standardele impuse
- **Moduri de validare:**
  - Revizii
  - Prototipuri
  - Validarea modelelor conceptuale
  - Stabilirea detaliata a modurilor de verificare finala (testelor de acceptare / sistem)

Lecturi recomandate

- http://cisas.unipd.it/didactics/STS_school/Software_development/Guide_to_the_user_requirements_definition_phase-0502.pdf
- http://cisas.unipd.it/didactics/STS_school/Software_development/Guide_to_the_SW_requirements_definition_phase-0503.pdf
- **Google: ESA Software Engineering Standards**
- http://www.agilemodeling.com/essays/agileAnalysis.htm

Extragerea cerintelor
și
Specificarea cerintelor

EXTRAGEREA CERINTELOR

Extragerea cerintelor

Scopul: identificarea si definirea cerintelor utilizator

- Se desfasoara in fazele de început ale procesului de dezvoltare a unui produs software.
- Poate continua pe intreg parcursul procesului de dezvoltare (in functie de modelul de dezvoltare folosit)
- Metoda folosită pentru extragerea cerintelor este adaptată la nivelul fiecarei organizatii şi în functie de caracteristicile fiecarui proiect.
- Rezultatul, “Cerintele utilizator” (Use Requirements) este sintetizat in “**Documentul de specificare a cerintelor**”.

- O **cerinţă** este o functionalitate pe care viitorul sistem trebuie sa o ofere sau o constrângere pe care sistemul trebuie să o satisfacă pentru a fi acceptat de client.

Participanţi la extragerea cerintelor (*stakeholders*)

• Analistul (Requirements engineer) – coordoneaza extragerea si definirea cerintelor

• Utilizatorii finali, beneficiarii viitorului produs

• Clientul / posibili cumparatori

• Analisti de marketing (pentru produse destinate pietei) – studiul pietei

• Reprezentanti ai autoritatilor din domeniul de operare al produsului software (probleme legislative, reguli in cadrul organizatiei)

• Ingineri software

- analizeaza fezabilitatea cerintelor exprimate de ceilalti participanti, cerintele tehnice, costurile.
- definesc scenarii şi cazuri de utilizare ale viitorului sistem informatic

Analistul negociaza între cerintele diferitilor participanti, cerintele tehnice, bugetare, legislative, etc.

Sursele cerintelor

- **studiul de piata**
- **studiul de fezabilitate** (realizabilitate, analiza scop/cost)
- **studiul domeniului:**
  - de ex. bancar, clinic, controlul in timp real al proceselor, etc; anumite cerinte sunt specifice domeniului;
- **procesul – pe care noul produs trebuie sa-l imbunatateasca**
- **actorii – fiecare are interese specifice**

Tehnici de extragere a cerintelor

Difera, în functie de natura produsului software:

- Studiul produselor software similare existente – se doreste realizarea unui produs care sa ofere ceva in plus: functionalitate, uzabilitate, tehnologie, etc.
- Studiul sistemului informational al organizatiei in care va fi implementat sistemul informatic, entitatile, taskurile, etc, modul in care poate fi imbunatatit sistemul informational prin introducerea unui sistem informatic
- Interviuri cu viitorii utilizatori, clientul, reprezentanti ai autoritatilor din domeniul de operare al produsului software
- Descriere scenarii şi cazuri de utilizare ale viitorului sistem
- Utilizarea de prototipuri executabile ale viitorului sistem

Activităţi

Extragerea cerinţelor include urmatoarele activitati:

- **Identificarea actorilor:** entitati externe sistemului care interactioneaza cu sistemul
- **Identificarea şi descrierea scenariilor de utilizare:** dezvoltatorii definesc scenarii pentru functionalitatile tipice care vor fi furnizate de viitorul sistem.
  - Scenariile sunt exemple concrete de utilizare a viitorului sistem
  - Ele uşureaza comunicarea dezvoltatorului cu utilizatorii viitorul sistem şi înţelegerea domeniului aplicatiei
- **Definirea cazurilor de utilizare ale sistemului:** plecand de la scenarii, dezvoltatorii definesc un set de cazuri de utilizare care descriu toate posibilitatile de utilizare a viitorului sistem.
- **Rafinarea cazurilor de utilizare:** cazurile de utilizare se detaliaza descriind comportarea sistemului în prezenţa erorilor şi a condiţiilor exceptionale
- **Identificarea relatiilor dintre cazurile de utilizare**
- **Identificarea şi definirea cerintelor nefunctionale:** constrangeri privind performanta sistemului, consumul de resurse, cerinte de securitate, etc.

Identificarea actorilor (1)

- Se determina tipurile de persoane care pot influenta dezvoltarea noului sistem informatic:
  - Care vor utiliza direct sistemul
  - Care vor utiliza rezultatele produse de sistem (de ex. rapoarte din baza de date)
  - Care nu vor utiliza sistemul dar pot influenta conceptia sa (impun reguli in organizatie)
- Se descriu **caracteristicile posibililor utilizatori**: sarcini, abilitati de lucru cu calculatorul, cunostintele lor tehnice.
- Se determina **tipurile de utilizatori directi,** operatiile pe care le va efectua cu sistemul fiecare tip de utilizatori, nr. aproximativ de utilizatori din fiecare categorie.
- Se determina entitatile externe sistemului, care vor comunica cu sistemul

➢ Un **actor este un rol pe care o entitate externa îl joaca in raport cu sistemul**:

- Un utilizator direct al sistemului (un utilizator poate juca mai multe roluri)
- Un echipament extern sau alt sistem cu care sistemul analizat comunica

Identificarea actorilor (2)

Intrebări care pot ajuta la identificarea actorilor:

- Care grupuri de utilizatori vor folosi direct în munca lor noul sistem?
- Care grupuri de utilizatori vor utiliza principalele functii ale sistemului?
- Care grupuri de utilizatori vor efectua operatii secundare, cum ar fi cele de mentenanta sau de administrare?
- Care sunt sistemele externe hardware sau software care vor comunica cu sistemul?

Identificarea actorilor (3)

STUDIU DE CAZ: SGCB - Sistem de gestiune electronica a cartilor din mai multe biblioteci

Se doreste realizarea unui sistem informatic pentru gestiunea centralizata a cartilor existente in mai multe biblioteci.

Sistemul trebuie sa ofere urmatoarele functionalitati:

-Sa permita inregistrarea persoanelor ca abonati

-Sa permita abonatilor sa caute si sa imprumute sau sa restituie carti

-Sa permita inregistrarea de noi carti si scoaterea din evidenta a unora existente

-Sa pastreze evidenta abonatilor si a cartilor imprumutate de fiecare abonat.

Deoarece sistemul realizeaza o gestiune centralizata, pentru accesul său se propune o interfaţă Web.

Identificarea actorilor (4)

Din descrierea anterioara rezulta ca vor exista doua tipuri de utilizatori directi ai sistemului:

- Persoanele care vor accesa sistemul pentru a căuta, împrumuta sau restitui cărti
- Persoanele care vor înregistra noi abonaţi, vor actualiza lista de carti împrumutate de un abonat, vor înregistra noi carti sau vor elimina pe unele existente

- **Identificarea tipurilor de utilizatori**: **pe baza operatiilor efectuate cu sistemul de utilizatorii direcţi**, care definesc două roluri:
  - Rolul de abonat
  - Rolul de bibliotecar
- O persoana poate juca atat rolul de abonat cat si rolul de bibliotecar.
- Fiecare dintre cele 2 roluri poate fi jucat de mai multe persoane.

- Actorii identificati sunt **Abonatul** şi **Bibliotecarul**, reprezentati în UML astfel:

Descrierea utilizatorilor

- **Tipurile de utilizatori:** pentru fiecare categorie
  - numarul aproximativ
  - rolul in raport cu sistemul
  - caracteristicile (abilitati necesare, etc)
  - profilul temporal de utilizare a sistemului (frecventa, durata, etc.)
  - lista operatiilor pe care le vor putea efectua
- **Alte (categorii de) persoane relevante**:
  - Utilizatori indirecti: utilizeaza rezultatele sistemului, fara a interactiona direct cu el: descriere similara
  - Care nu utilizeaza sistemul dar pot influenta dezvoltarea sa – de ex. cei care impun proceduri, reguli de ordine interna in organizatie: descrierea responsabilitatilor si a modurilor de rezolvare a diverselor situatii in raport cu aceste persoane

Scenarii de utilizare (1)

Identificarea şi descrierea scenariilor de utilizare

- Un **scenariu** este o descriere narativa (informala) a unei singure functionalitati a unui sistem informatic, prin prisma unei interacţiuni concrete dintre un actor şi sistem.
- Scenariile uşurează extragerea cerinţelor, fiind un instrument uşor de înţeles de utilizatori si clienţi.

Scenarii de utilizare (2)

Exemple de scenarii de utilizare a **SGCB:**

(1)

1. Un utilizator care doreşte să împrumute o carte completează rubricile rezervate numelui de utilizator şi parolei de acces, în interfaţa Web a sistemului, apoi apasa butonul “Submit”.
2. Sistemul preia datele şi verifica identitatea utilizatorului.
3. Sistemul afişează mesajul: „Nume de utilizator inexistent. Nu sunteti înregistrat ca abonat. Efectuati procedura de înregistrare”.

(2)

1. Un utilizator care doreşte sa împrumute o carte completează rubricile rezervate numelui de utilizator şi parolei de acces, în interfaţa Web a sistemului, apoi apasa butonul “Submit”.
2. Sistemul preia datele şi verifica identitatea utilizatorului.
3. Sistemul afişează mesajul: „Ati depasit numarul maxim de cărti împrumutate. Restituiti o parte dintre ele”.

Scenarii de utilizare (3)

(3)

1. Un utilizator care doreşte sa împrumute o carte completează rubricile rezervate numelui de utilizator şi parolei de acces, în interfaţa Web a sistemului, apoi apasa butonul “Submit”.
2. Sistemul preia datele şi verifica identitatea utilizatorului.
3. Sistemul afişeaza formularul de împrumut.
4. Abonatul completeaza formularul de împrumut, cu titlul cartii, numele şi prenumele autorului şi codul ISBN al cartii apoi apasa butonul “Submit”.
5. Sistemul preia datele şi cauta cartea.
6. Sistemul afişează mesajul: „Cartea nu există în bibliotecile noastre”.

Scenarii de utilizare (4)

Intrebari care pot ajuta la identificarea scenariilor:

- Care sunt operatiile pe care un anumit actor doreste ca sistemul sa le efectueze?
- Care sunt datele pe care le acceseaza actorul? Cine creaza aceste date? Pot fi ele modificate sau eliminate? De catre cine?
- Care sunt schimbarile externe pe care actorul trebuie sa le comunice sistemului? Cand?
- Care sunt evenimentele pe care sistemul trebuie sa le comunice actorului? Cat de repede?

Cazuri de utilizare (1)

Definirea cazurilor de utilizare ale sistemului

- Numarul de scenarii de utilizare a unui sistem este foarte mare!
- Un set de scenarii de utilizare corelate se abstractizeaza într-un **caz de utilizare**.
- Un scenariu este o instanta a unui caz de utilizare.
- Un caz de utilizare descrie o functionalitate a sistemului în raport cu un actor; este o abstractizare a tuturor scenariilor care descriu acea functionalitate.
- *Totalitatea cazurilor de utilizare ale unui sistem reprezintă toate modurile în care poate fi utilizat sistemul respectiv.*

- Un caz de utilizare este initiat de un actor.
- Dupa initierea sa, cazul de utilizare poate interactiona si cu alti actori.
- **Un caz de utilizare reprezinta un flux complet de evenimente prin sistem, declanşat ca urmare a initierii sale**.

Cazuri de utilizare (2)

Desi UML admite variatii în descrierea cazurilor de utilizare, **în general un caz de utilizare este descris prin secventa tipica de paşi (comportamentul de bază) şi alternativele la secvenţa tipica.**

Exemplu: Cazul de utilizare ”Imprumut” al sistemului de gestiune a cartilor (SGCB)

Fluxul de baza (secvenţa tipica de paşi)

1. Un utilizator care doreşte sa împrumute o carte completează rubricile rezervate numelui de utilizator şi parolei de acces, în interfaţa Web a sistemului, apoi apasa butonul “Submit”.
2. Sistemul preia datele şi verifică identitatea utilizatorului.
3. Sistemul afişează formularul de împrumut.
4. Abonatul completeaza formularul de împrumut, cu titlul cartii, numele şi prenumele autorului şi codul ISBN al cartii apoi apasa butonul “Submit”.
5. Sistemul preia datele şi cauta cartea.
6. Sistemul înregistreaza împrumutul.
7. Sistemul afişează mesajul “Luati cartea de la ghiseu”.

Cazuri de utilizare (3)

Alternative:

La pasul 3:

3a) Utilizatorul nu este inregistrat ca abonat si atunci sesiunea este incheiata de sistem, cu mesajul: „ Nume de utilizator inexistent. Nu sunteti inregistrat ca abonat. Efectuati procedura de înregistrare”.

3b) Utilizatorul este inregistrat dar a depasit numarul maxim admis de carti împrumutate. Sesiunea este încheiată de sistem cu mesajul:

„Ati depasit numarul maxim de carti imprumutate. Restituiti o parte dintre ele”.

La pasul 6:

6a) Cartea nu este găsita. Sistemul afiseaza mesajul: „Cartea nu există în bibliotecile noastre”.

Cazuri de utilizare (4)

Repetarile de comportament intr-un caz de utilizare pot fi descrise prin formulari de tipul:

n. **repeta**
  n.1. ---------
  n.2.-----------
până (conditie)

sau

n. **cat timp (conditie) repetă**
  n.1. …….
  n.2. ……

Sunt admise, de asemenea, formulari de tipul:

if conditie se continua cu pasul x

sau

if conditie se continua cu pasul x

else se continua cu pasul y

Cazuri de utilizare (5)

Descrierea tipică a unui caz de utilizare cuprinde urmatoarele elemente:

- Descrierea comportamentului de bază al sistemului (fluxul principal de evenimente și operaţiile declanșate) și alternativele – o descriere pas cu pas a acțiunilor actorului și sistemului.
- Pre-condiția (opțional) – o constrângere asupra sistemului la inițierea cazului de utilizare; se specifică în limbaj natural.
- Post-condiția (opțional) – o constrângere asupra sistemului la terminarea execuției cazului de utilizare; se specifică în limbaj natural.
- Cerințe speciale – cerințe ce nu pot fi descrise cu ușurință în fluxul de evenimente.
- O schiță a interfeței utilizator (opțional).

In faza de analiza a cerintelor, cazurile de utilizare se rafinează si formalizează folosind diagrame de secvență UML pentru descrierea scenariilor şi diagrame de activitate UML pentru descrierea operațiilor.

Cazuri de utilizare (6)

- Preconditia si postconditia pot fi specificate fie la inceputul fie la sfarsitul descrierii cazului de utilizare.

De exemplu, pentru cazul de utilizare anterior:

Preconditie:

Postconditie: In baza de date a sistemului exista o inregistrare a imprumutului catre abonat

In acest caz, preconditia are valoarea *true* intotdeauna.

In UML, un caz de utilizare se reprezinta printr-o elipsa in interiorul careia este scris numele cazului de utilizare.

Relatia dintre un actor si un caz de utilizare se reprezinta printr-o linie de comunicare:

Cazuri de utilizare (7)

Rafinarea cazurilor de utilizare

In acest pas se adauga detalii la descrierile cazurilor de utilizare, care specifică comportarea sistemului în prezenţa erorilor şi a condiţiilor exceptionale.

De exemplu, pentru cazul de utilizare **”Imprumut”** se pot adauga cazurile de exceptie:

Exceptii

La pasul 2: Sesiunea este încheiata de sistem cu mesajul “Eroare de acces la baza de date de utilizatori”

La pasul 2: Sesiunea este incheiata de sistem cu mesajul “Eroare la scrierea in baza de date de carti”

Relaţii între cazurile de utilizare (1)

Identificarea relaţiilor dintre cazurile de utilizare

Multe cazuri de utilizare descrise în pasul anterior au subfluxuri de evenimente comune, care pot fi descrise separat, în loc de a fi repetate.

Intre cazurile de utilizare se pot stabili relatii de:

- **Includere** – un caz de utilizare A include, in unul sau mai multi pasi, comportamentul descris într-un alt caz de utilizare, B.
- **Extindere** – un caz de utilizare E extinde un alt caz de utilizare, A, daca A poate include comportamentul descris in E, numai in anumite conditii.
- **Generalizare**

In UML, reprezentarea unui set de cazuri de utilizare şi a relaţiilor dintre ele se numeşte **diagramă de cazuri de utilizare.**

Relatii între cazurile de utilizare (2)

Relatia de includere - exemplificare

Orice utilizator al SGCB ar trebui ca, inainte de orice operatie cu sistemul, sa se autentifice:

- operatia de autentificare este inclusa in orice caz de utilizare al sistemului;
- poate fi descrisa intr-un caz de utilizare separat care va fi inclus in toate celelalte cazuri de utilizare

Cazul de utilizare "Autentificare"

Fluxul de baza

Preconditie:

1. Un utilizator completează rubricile rezervate numelui de utilizator şi parolei de acces, în interfaţa Web a sistemului, apoi apasa butonul "Submit".
2. Sistemul preia datele şi verifică identitatea utilizatorului.
3. Numele de utilizator si parola sunt corecte si utilizatorul este autentificat in sistem.

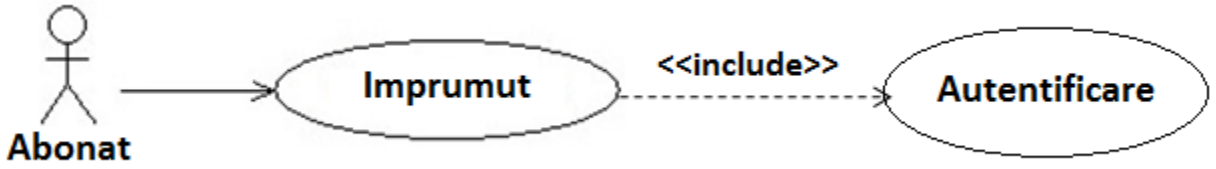
Postconditie: utilizatorul este autentificat in sistem

Relatii între cazurile de utilizare (3)

Alternative:

La pasul 3: Sistemul afiseaza mesajul: „Nume de utilizator inexistent sau combinatia nume utilizator-parola este incorecta” si incheie sesiunea.

Cazul de utilizare “autentificare” va fi inclus in cazul de utilizare “Imprumut” si în toate celelalte cazuri de utilizare:

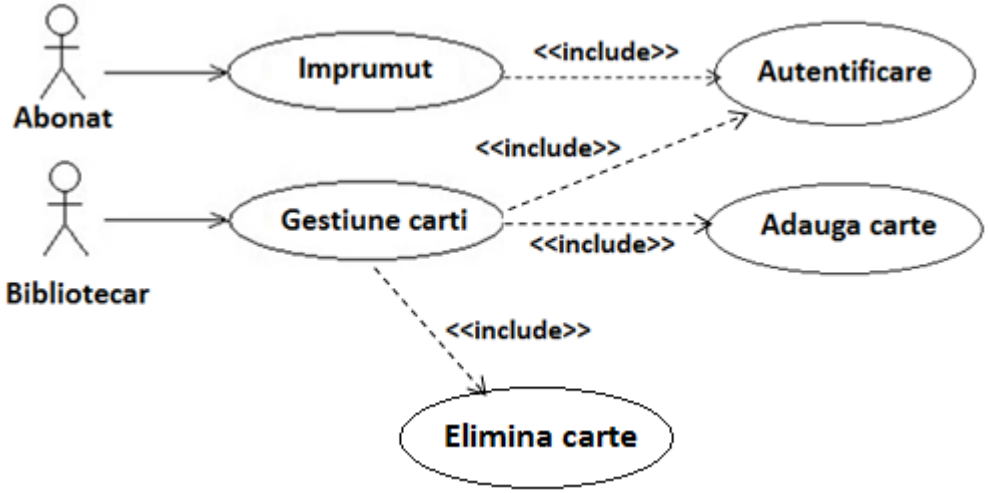


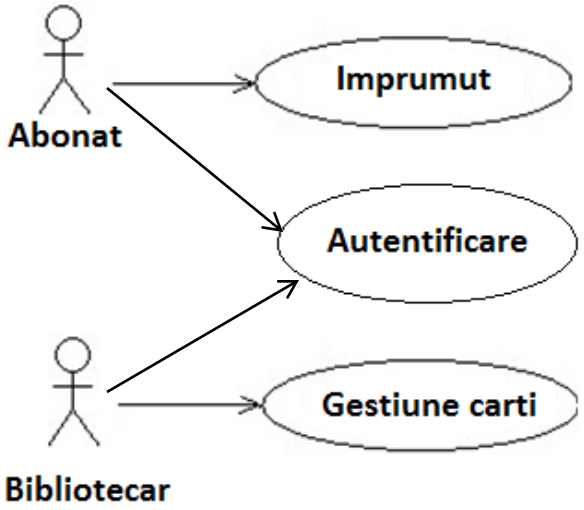
Cazul de utilizare “Imprumut”

Preconditie:

1. Utilizatorul executa procedura de “Autentificare”
2. Daca utilizatorul nu este autentificat, atunci cazul de utilizare se incheie.
3. Sistemul afişează formularul de împrumut.
4. ……

Relatii între cazurile de utilizare (4)





O alta posibilitate de descriere a cazului de utilizare “Imprumut”, si a celorlalte, care corespunde diagramei din partea dreaptă, este:

Cazul de utilizare “Imprumut”
Preconditie: utilizatorul este autentificat în sistem

1. Sistemul afişează formularul de împrumut.

2…….

Aceasta presupune că înainte de orice caz de utilizare a sistemului, utilizatorul trebuie sa execute procedura de autentificare (cazul de utilizare ***Autentificare***) terminata prin autentificarea sa în sistem.

Relatii între cazurile de utilizare (5)

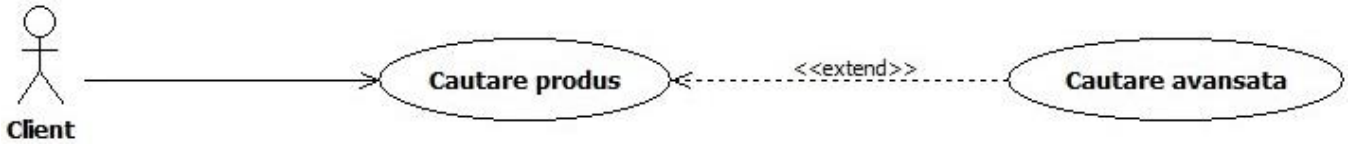
Relatia de extindere - exemplificare

Utilizatorul unui site on-line dorește să găsească un anumit produs. Dacă funcționalitatea de căutare nu îi este suficientă pentru găsirea produsului dorit, îi este oferită opțiunea de căutare avansată.

Operaţia de cautare avansata poate fi descrisa printr-un caz de utilizare separat, **“Căutare avansată”,** care extinde cazul de utilizare **“Cautare produs”** atunci cand cautarea normala nu are rezultat.

Un posibil flux de evenimente pentru cazul descris este:

1. Utilizatorul introduce un criteriu de căutare.
2. Căutarea nu întoarce niciun rezultat şi utilizatorului îi este oferită opţiunea de a efectua o căutare avansată.
3. Utilizatorul selectează opţiunea de căutare avansată.
4. Se executa operatia de cautare avansata.
5. Se afişează rezultatele căutării avansate.



Relatii între cazurile de utilizare (6)

Euristici privind relatiile de includere si extindere:

- Relatia de extindere se utilizeaza pentru a evidentia comportamente ce corespund unor cazuri optionale, exceptionale, sau care se produc rar.
- Relatia de includere se utilizeaza atunci cand un comportament este comun mai multor cazuri de utilizare.
- Ambele relatii trebuie utilizate cu discretie deoarece prea multe relatii intre cazurile de utilizare pot ingreuna intelegerea acestora.

Generalizarea

- Atât între actori, cât și între cazuri de utilizare, se pot stabili relații de *generalizare*, care au aceeaşi semnificaţie cu relația de generalizare dintre clase: mai multe tipuri de actori/cazuri de utilizare fac parte dintr-un același super-tip, având caracteristici comune.

Relatii între cazurile de utilizare (7)

Relatia de generalizare

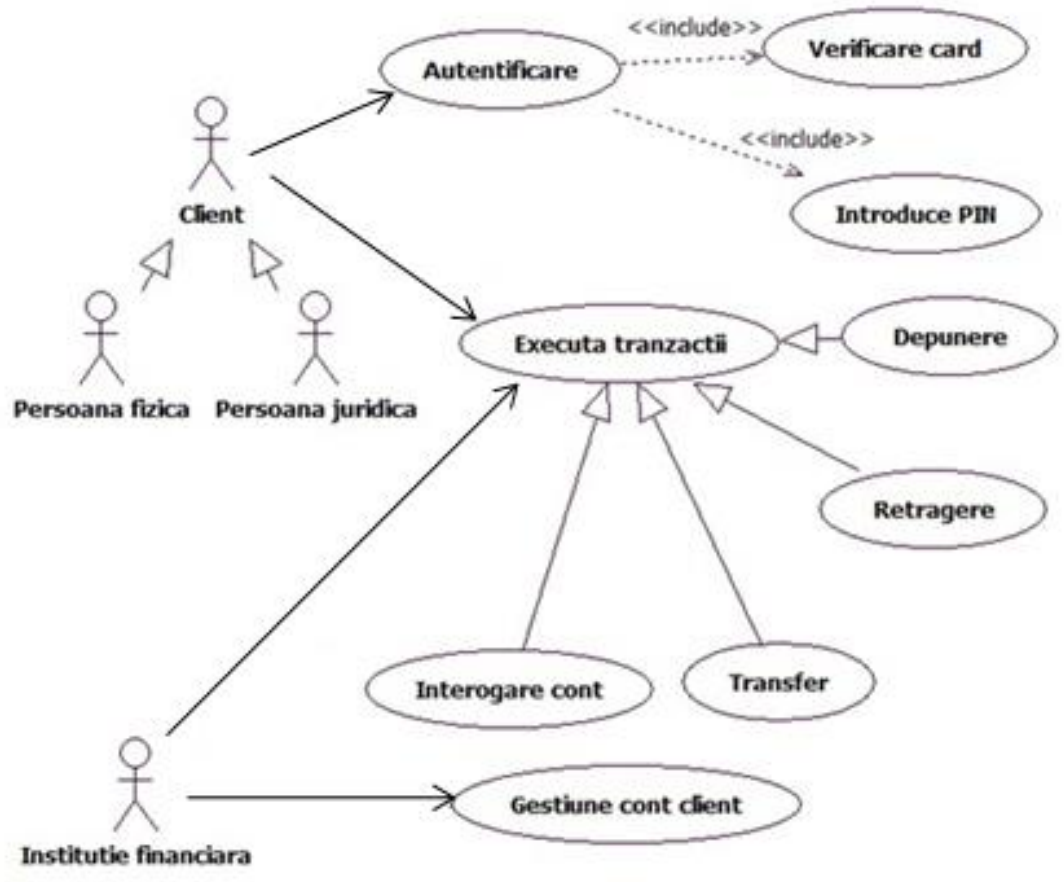
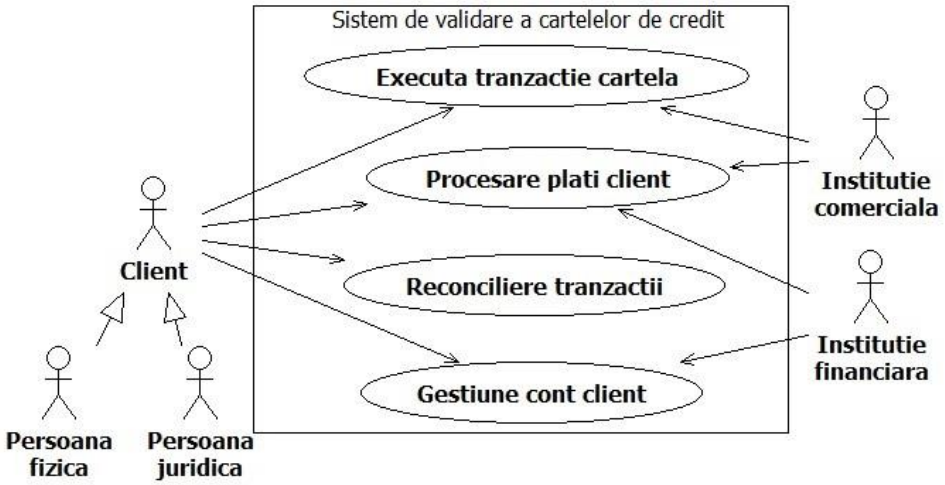


Diagrama de context

- Este o diagrama care cuprinde toate cazurile de utilizare ale unui sistem şi actorii.
- Are rolul de a delimita sistemul de mediul sau de operare.



Cerinte nefunctionale (1)

Identificarea cerintelor nefunctionale

Cerintele nefunctionale sunt **constrangeri de realizare a sistemului si a proiectului**:

- De interfata:
  - comunicare hardware-software, compatibilitate cu alte sisteme software, interfata utilizator.
- De calitate a produsului:
  - Fiabilitate, portabilitate, adaptabilitate, disponibilitate, securitate, standarde.
- De planificare a proiectului: termene, produse livrate, resurse necesare

Cerinte nefunctionale (2)

Cerinte de performanta

- Valori numerice atasate unor parametri masurabili cum ar fi: viteza, capacitatea, precizia, frecventa.

  Exemplu: sistemul masoara temperatura cu o precizie de 1 grad Celsius.

- Pot fi reprezentate ca un domeniu de valori: valoare acceptabila: [min-max], valoare ideala.

Cerinte de interfata

- Componente hardware/software cu care interactioneaza produsul
- Interfete interne / externe ale produsului software
- Se definesc prin:
  - Protocoalele de comunicatie de utilizat
  - Fluxul de date prin interfata
  - Cand au loc schimburi de date prin interfata, etc

Cerinte nefunctionale (3)

Cerinte de operare

- Modul de comunicare cu operatorii umani, aspecte fizice si ergonomice ale interfetei utilizator:
  - Descrierea dialogului
  - Aspectul ecranului in timpul dialogului
  - Stilul limbajului de comenzi

Cerinte impuse resurselor fizice

  - Puterea de prelucrare
  - Memoria necesara
  - Spatiul pe disc

Cerinte nefunctionale (4)

Cerinte de verificare a produsului final

- Cerinte impuse mediului de testare.
- Posibilitati de diagnosticare.
- Cerinte pentru testarea de acceptare.
- Efectuarea unor simulari (atunci cand sistemul nu poate fi testat in mediul operational inainte de testarea de acceptare).

Cerinte de calitate

- Utilizarea anumitor standarde de produs sau de proces
- Utilizarea de personal extern pentru asigurarea calitatii

Cerinte de portabilitate

  - "Sa poata rula pe calculatorul X fara modificarea codului sau modificand cel mult 2% din codul sursa”.
  - "Nici o parte a software-ului nu trebuie scrisa in assembler”.

Cerinte nefunctionale (5)

Cerinte de intretinere

- Usurinta de reparare a erorilor, de imbunatatire sau adaptare la schimbarea cerintelor. Exemplu: "Timpul de reparare a unei erori nu va depasi niciodata o saptamana“.

Cerinte de fiabilitate

- Frecventa acceptata a caderilor software, in functie de categoria caderii (cadere: o comportare a sistemului neconforma cu specificatia)
- Exprimate folosind parametrii Mean Time Between Failures (MTBF) si Mean Time To Repair (MTTR).
- Exemplu: "Timpul minim intre doua caderi severe va fi mai mare de 3 luni”.

Cerinte nefunctionale (6)

Cerinte de securitate

- Cum sa fie securizat sistemul impotriva pericolelor:
  - Erori utilizator (distrugerea accidentala a software-ului sau datelor)
  - Hazarduri fizice (foc)
  - Access ne-autorizat
  - Virusi, securitatea comunicarii in retea

Cerinte de siguranta

- Protectia impotriva distrugerilor cauzate de caderile software. De exemplu, ce trebuie sa se intample in cazul producerii unei caderi software:
  - Degradare treptata
  - Continuare dintr-un anumit punct

SPECIFICAREA CERINTELOR

Documentarea cerintelor utilizator(1)

Cerintele extrase sunt definite:

- Intr-un **document general de Specificare a Cerintelor**, care include si rezultatul analizei cerintelor, numit Software Requirements Document (SRD) sau Requirements Analysis Document (RAD).

sau

- Intr-un document separat, numit **User Rquirements Document (URD)**

Sunt definite:

1. **Obiectivele generale ale sistemului**, granitele si constrangerile.
2. **Principalele entitati ale domeniului**, **fluxul informational, mediul de operare al viitorului sistem.**
3. **Cerintele de sistem**: configuratia hardware, siguranta in functionare, punerea in functiune, comunicarea cu sisteme externe, s.a.

Documentarea cerintelor utilizator(2)

4. **Actorii**: rolurile entitatilor externe (inclusiv utilizatori) care interactioneaza direct cu sistemul (om sau componenta externa).

5. **Cerintele functionale ale sistemului, specificate folosind:**

- limbajul natural; exemple:

- *masoara* temperatura si o *afiseaza* in grade Celsius
- *cere codul PIN*, *il verifica* si *afiseaza un mesaj de acceptare sau rejectare*

- scenarii de utilizare a sistemului de catre diferiti actori.

- interfata utilizator, prin capturi ecran, pentru diferite scenarii de utilizare

- cazurile de utilizare ale sistemului

6. Cerintele nefunctionale

Structura documentului de specificare a cerintelor

Introducere
- Scopul documentului
- Domeniul aplicatiei
- Scopul sistemului. Motivatie
- Obiectivele şi criteriile de succes
- Referinţe (produse similar / competitie, sisteme existente, studii de fezabilitate etc.)
- Definitii, abrevieri

Sistemul curent (*daca sistemul propus va inlocui un sistem)*

Sistemul propus
- Descriere generala
- Mediul de operare
- Utilizatori *(descrierea categoriilor de utilizatori direcți/indirecți ai sistemului si alte persoane relevante)*
- Cerinte de sistem *(echipamente de calcul, dispozitive, comunicația în rețea)*
- Cerinte functionale *(in limbaj natural)*
- Cerinte nefunctionale
  - De interfata: Interfata utilizator, Comunicare, Hardware-software, Compatibilitate cu alte sisteme
  - De calitate: Portabilitate, adaptabilitate, disponibilitate, securitate, standard
  - De planificare: Termene, Produse livrate, Resurse, Indatoriri ale beneficiarului etc.
- Modele ale sistemului
  - Model de cazuri de utilizare *(diagrame de c.u., diag. context, descrieri structurate ale c.u.)*
  - Interfata utilizator *(poate fi inclusa si in descrierea cazurilor de utilizare)*
  - Model static *(date, deployment)*
  - Model dinamic

Definirea cerintelor software

- Este faza de analiza a problemei.
- Documentul Cerintelor Software (Software Requirements Document - **SRD**) reflecta punctul de vedere al dezvoltatorului cu privire la problema de rezolvat, nu pe cel al utilizatorilor.

Standardul IEEE 830 (IEEE recommended practice for software requirements specifications) descrie continutul, calitatile si avantajele unei bune specificatii a cerintelor software:

Calitatile unei bune specificatii a cerintelor software

Specificatiile cerintelor software trebuie sa fie:

- Corecte
- Neambigue – fiecare cerinta definita are o singura interpretare
- Complete – ar trebui sa contina tot ceea ce este necesar pentru realizarea software-ului
- Consistente – intre ele si cu documentele pe care le refera
- Clasificate dupa importanta si/sau stabilitate
- Verificabile. Trebuie evitate cerinte ca :”va furniza un raspuns rapid”, “sistemul nu va cadea niciodata”, etc.
- Modificabile. Atunci cand o aceeasi cerinta apare in mai multe parti, actualizarile documentului sunt mai greu de facut
- Usor de corelat cu cerinte formulate in alte documente, de ex. URD

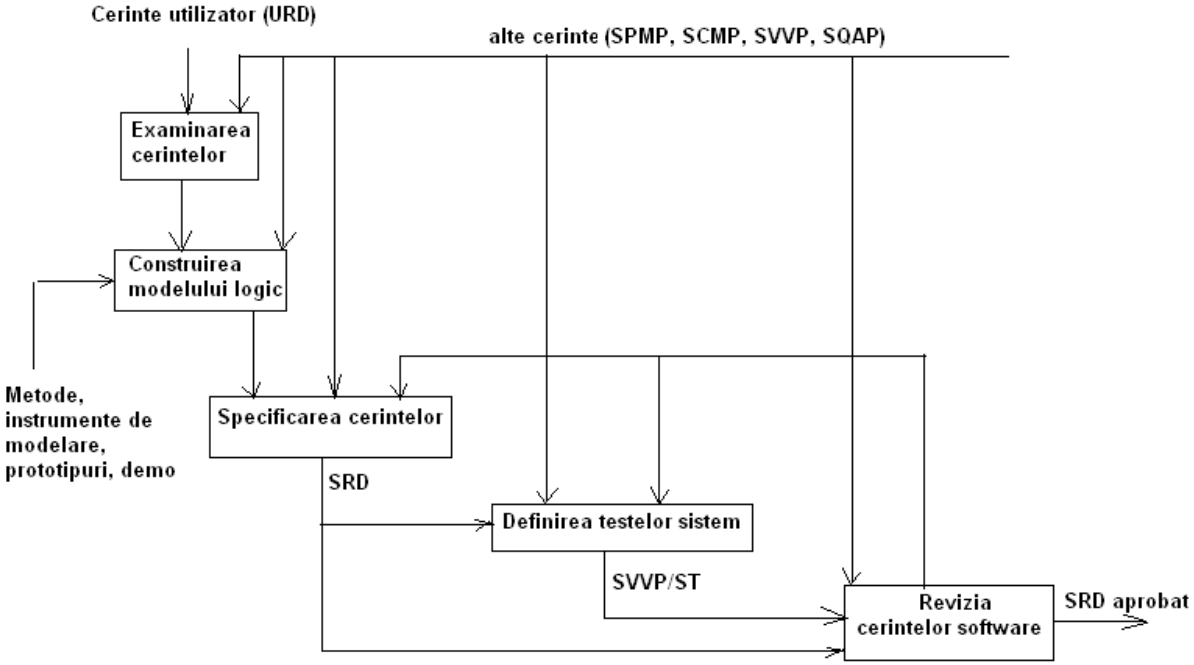
Avantajele unei bune specificatii a cerintelor software

- Sta la baza contractului dintre clienti si furnizori.
- Reduce efortul de dezvoltare.
- Sta la baza estimarii costurilor si a planificarii
- Permite planificarea testelor de verificare si validare
- Usureaza transferul produsului la noi utilizatori sau pe platforme noi.
- Serveste ca baza pentru viitoarele imbunatatiri sau modificari ale produsului.

Cine participa

- Definirea cerintelor software este o responsabilitate a dezvoltatorului.
- Alti participanti:utilizatori, ingineri de sistem, ingineri hardware si personal de operare.

Activitati in etapa de definire a cerintelor software



Activitatile si fluxul documentelor in etapa de definire a cerintelor software

Modelul logic / modelul conceptual / modelul de analiza

- Descriere abstracta a sistemului
- Se folosesc termeni din domeniul aplicatiei → **modelul domeniului** (domain model). Contine entitatile cheie ale domeniului si relatiile dintre ele.
- Este independent de implementare
- Este o descriere de nivel inalt a sistemului
- Este construit utilizand o metoda de analiza recunoscuta si instrumente de modelare
- Are o structura ierarhica, obtinuta prin aplicarea unor criterii de decompozitie consistente
- Alcatuit din simboluri organizate dupa anumite conventii
- Permite intelegerea cerintelor ca un ansamblu, nu numai individual

Elemente de modelare folosite in **analiza structurata**:

- Diagrame de flux de date
- Diagrame de stari-tranzitii
- Diagrame Entitate-Relatie

Elemente de modelare folosite in **analiza orientata obiect**:

- Diagrame de secventa sistem (System Sequence Diagrams)
- Clasele conceptuale (entitati, concepte, abstractii din domeniul aplicatiei) si relatiile dintre aceste clase
- Atribute ale obiectelor din domeniul aplicatiei, care determina schimbari de stare ale sistemului
- Diagrame de interactiune intre obiectele din domeniul aplicatiei
- Diagrame de stari care redau starile sistemului sau starile unui obiect din domeniul aplicatiei

Obs: clasele conceptuale nu sunt clase software; in etapa de proiectare pot fi definite clase software care corespund claselor conceptuale.

Specificarea cerintelor

Sunt 2 tipuri de cerinte software:

- Functionale
- Ne-functionale

Cerinte functionale

- Descriu functiile pe care trebuie sa le realizeze sistemul, intr-un mod independent de implementare.
- Ce transformari trebuie efectuate asupra intrarilor si ce iesiri trebuie sa se obtina pentru fiecare tip de intrari.

Cerinte ne-functionale

- Sunt atasate cerintelor functionale
- Majoritatea rezulta din constrangerile incluse in Specificatia Cerintelor Utilizatorilor
- Cerinte de: performanta, interfata, de operare, de verificare, de portabilitate, de intretinere, de fiabilitate, s.a.

Cerinte de performanta

- Valori numerice atasate unor parametri masurabili cum ar fi: viteza, capacitatea, precizia, frecventa.
  - Sistemul masoara temperatura in grade Celsius
  - Temperatura este masurata cu o precizie de 1 grad Celsius

  sau

  - Sistemul masoara temperatura cu o precizie de 1 grad Celsius
- Pot fi reprezentate ca un domeniu de valori:

- Valoare acceptabila (nivel minim de performanta)
- Valoare nominala
- Valoare ideala

Cerinte de interfata

- Protocoalele de comunicatie de utilizat
- Fluxul de date prin interfata
- Cand au loc schimburi de date prin interfata, etc

Cerinte operationale

Modul de comunicare cu operatorii umani: aspecte fizice si ergonomice ale interfetei utilizator:

- Descrierea dialogului,
- Aspectul ecranului in timpul dialogului
- Stilul limbajului de comenzi

Cerinte impuse resurselor fizice

- Puterea de prelucrare
- Memoria necesara
- Spatiul pe disc

Cerinte de verificare

- Efectuarea unor simulari
- Cerinte impuse mediului de testare
- Posibilitati de diagnosticare

Cerinte pentru testarea de acceptare

Cerinte de portabilitate

- "Poate rula pe calculatorul X fara modificara codului sau ..modificand cel mult 2% din codul sursa”
- "Nici o parte a software-ului nu trebuie scrisa in assembler."

Cerinte de intretinere

"Timpul de reparare a unei erori nu va depasi niciodata o saptamana"

Cerinte de fiabilitate

- Frecventa acceptata a caderilor software, in functie de categoria caderii:
  Severa, avertizare, informare.

- Exprimate folosind parametrii Mean Time Between Failure (MTBF) si Mean Time To Repair (MTTR). Exemple:
  - "Timpul minim intre doua caderi severe va fi mai mare de o luna”

Cerinte de securitate

Cum sa fie securizat sistemul impotriva pericolelor:

- Erori utilizator (distrugerea accidentala a software-ului sau datelor)
- Hazarduri fizice (foc)
- Access ne-autorizat
- Virusi

Cerinte de siguranta

Protectia impotriva distrugerilor cauzate de caderile software

Alte cerinte de calitate

- Utilizarea anumitor standarde de produs sau de proces
- Utilizarea de personal extern pentru asigurarea calitatii.

Documentul Cerintelor Software (SRD)

Contine:

- O descriere generala a scopului produsului
- O descriere a mediului de operare: echipamentele (hardware) si sistemul de operare
- Mediul de dezvoltare care urmeaza sa fie utilizat
- Daca produsul este un sistem independenmt sau o parte a unui sistem mai mare sau inlocuieste un alt sistem. Caracteristicile esentiale acestui sistem
- O descriere a modelului logic al sistemului.
- Lista cerintelor functionale
- Lista cerintelor nefunctionale
- Modelul logic

The Software Requirements Document - sablon definit in standardele ESA

a.	Abstract	
b.	Table of Contents	
c.	Document Status Sheet	Status sheet for configuration control.
d.	Document Change Records since previous issue	A list of document changes.
1. Introduction		
1.1	Purpose	The purpose of this particular SRD and its intended readership.
1.2	Scope	Scope of the software. Identifies the product by name, explains what the software will do.
1.3	List of definitions	The definitions of all used terms, acronyms and abbreviations.
1.4	List of references	All applicable documents.
1.5	Overview	Short description of the rest of the SRD and how it is organized.
2. General description		
2.1	Relation to current projects	The context of this project in relation to other current projects.
2.2	Relation to predecessor and successor projects	The context of this project in relation to past and future projects.
2.3	Function and purpose	A general overview of the function and purpose of the product.
2.4	Environment	Hardware and operating system of target system and development system. Who will use the system (see user roles in URD).
2.5	Relation to other systems	Is the product an independent system, part of a larger system, replacing another system? The essential characteristics of these other systems.
2.6	General constraints	Reasons why constraints exist: background information and justification (analogous to URD).
2.7	Model description	A description of the logical model.
3. Specific requirements		
3.1	Functional requirements	A list of all functional requirements. Note the general remarks about requirements.
3.2 – 3.14	Non-functional requirements	A list of all non-functional requirements. These requirements are linked to functional requirements. Note the general remarks about requirements. Each category of non-functional requirements has its own subsection.
4. Requirements traceability matrix		A table showing how each user requirement of the URD is linked to software requirements in the SRD.

Revizia cerintelor software (Software Requirements Review)

- Este o revizie tehnica.
- Revizie interna, la care participa conducatorul proiectului (project manager), utilizatori si dezvoltatorul. Revizie externa - departamentul de calitate.
- Se verifica daca cerintele sunt clare, consistente, complete si suficient de detaliate pentru proiectarea produsului software.

Cat timp sa se consume cu definirea cerintelor software ?

Reguli generale:

Statistic, s-a constatat ca este bine ca:

- Aproximativ 20-25% din timpul total al proiectului sa fie alocat definirii cerintelor.
- 5% din timpul proiectului sa fie alocat pentru actualizarea cerintelor dupa ce a inceput proiectarea.

Documentatia poate fi minima daca produsul va fi utilizat pentru o perioada de timp scurta sau este dedicat unui numar mic de utilizatori

Se aloca mai mult timp cerintelor complicate.

Nu trebuie supra-documentate functii usor de inteles de multe persoane.

SRD trebuie actualizat ori de cate ori se fac modificari.

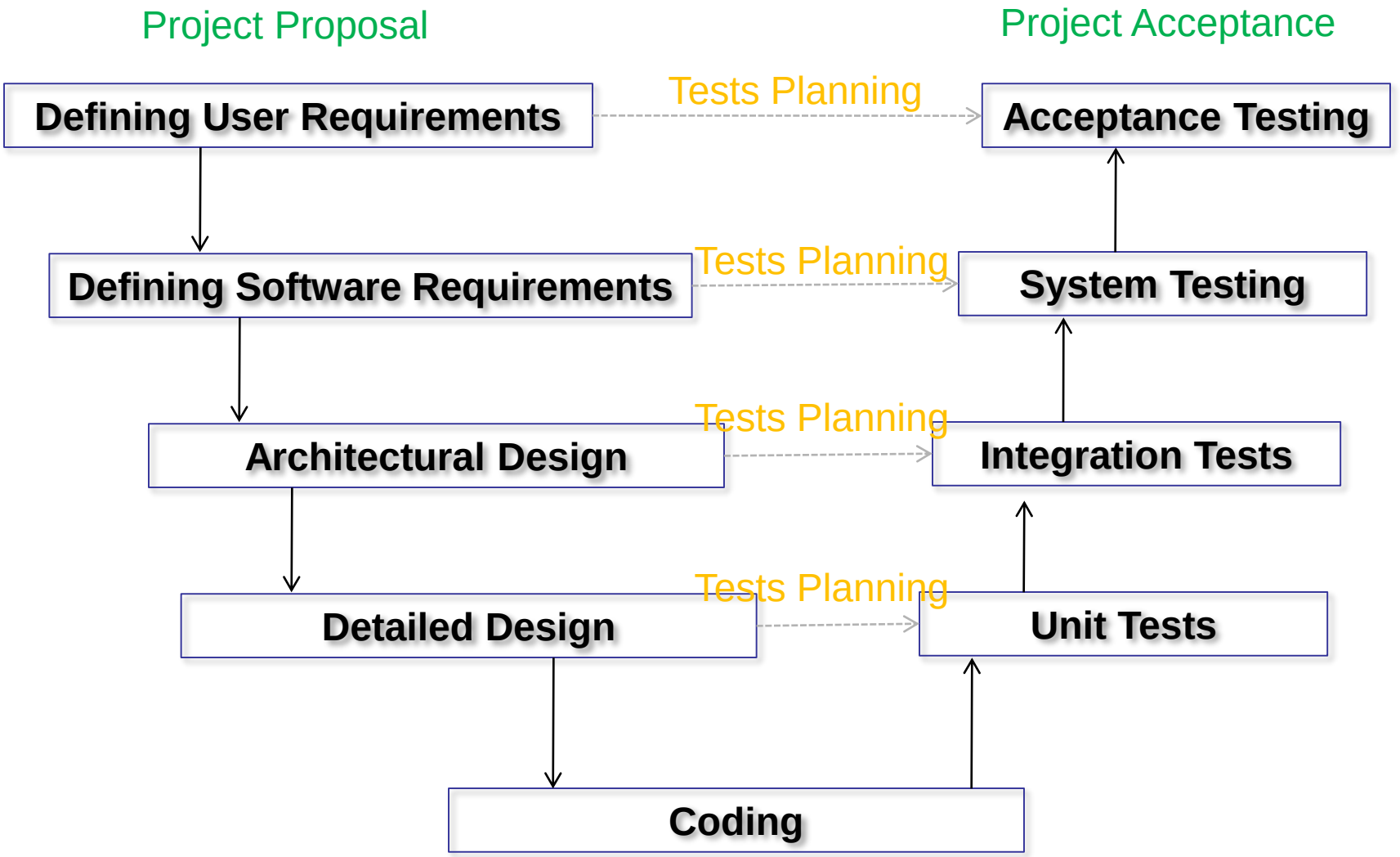
Modelarea sistemelor informatice

Modelarea: parte esentiala in orice proiect software, in special in proiectele mari.

Modelele: - reprezentari abstracte ale sistemului

- create in etapele care preced codificarea:
  - Analiza si specificarea cerintelor,
  - Proiectarea arhitecturala
  - Proiectarea de detaliu
- utilizate:
  - inainte de codificare: pentru a verifica daca toate cerintele utilizatorilor sunt acoperite, daca functiile prevazute sunt complete si corect modelate, daca arhitectura este robusta si extensibila.
  - dupa codificare, pentru verificarea si validarea sistemului.

“V” MODEL



Models created

Models used

Utilizarea modelelor

Modelele de analiza:

- Exprima cerintele impuse sistemului
- Corespund unei vederi externe asupra sistemului
- Create de persoanele implicate in analiza
- Folosite ca produs de catre designeri

Modelele de proiectare:

- Redau arhitectura sistemului, alocarea cerintelor pe subsisteme, distributia proceselor in sistem, sincronizarea lor
- Realizarea fizica a sistemului, echipamentele din componenta sa si repartitia
- componentelor program pe diferite componente hardware

UML - Unified Modelling Language

- **limbaj de modelare OO. NU este o metoda OO**
- **independent de procesul de dezvoltare folosit.**
- **aparut din necesitatea unei standardizari a elementelor de modelare folosite in metodele de dezvoltare orientata obiect**
- **Istorie:**
  - Prima versiune UML a fost publicata in 1996 (rezultatul colaborarii intre cei trei lideri recunoscuti in domeniul metodologiilor orientate obiect: Booch, Rumbaugh, and Jacobson).
  - Ianuarie 1997: UML 1.0
  - Noiembrie 1997: UML 1.1
  - Iunie 1998:UML 1.2
  - 1999: UML 1.3
  - 2002: UML 1.4
  - Octombrie 2004: UML 2.0
  - Stadiu actual: UML 2.4 (2.5 publicat informal)

WWW.UML.ORG

Utilitatea UML

- **Specificare**
- **Construire**
- **Vizualizare**
- **Comunicare**
- **Documentare**

Categorii de diagrame UML

Modelare comportamentala

- Diagrame de cazuri de utilizare
- Diagrame de interactiune
- Diagrame de stari
- Diagrame de activitati

Modelare structurala

- Diagrame de clase
- Diagrame de obiecte
- Pachete
- Diagrame de componente
- Diagrame de distributie etc.

Lecturi suplimentare

- **Alin Moldoveanu et al., UML Practic, Editura Matrix Rom (COD CNCSIS: 39), ISBN 978-606-25-0118-1. 168 pagini. 2014**

- **http://my.safaribooksonline.com/book/software-engineering-and-development/uml/0321193687**

UML

O perspectivă pragmatică

?

Ce este UML

Limbaj de modelare software prin diagrame
(cu notație, semantică și utilitate specifică)

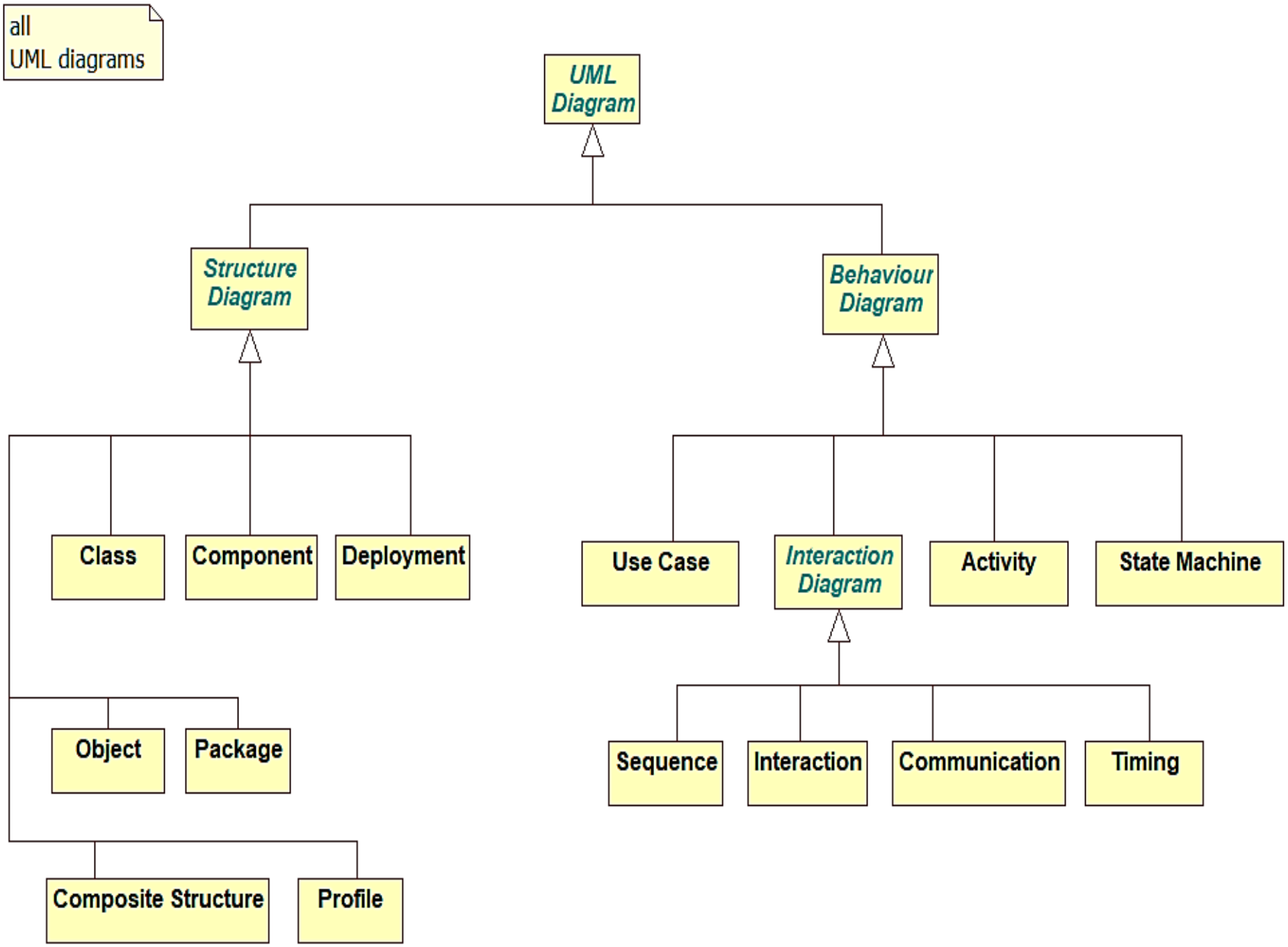
- 14 tipuri de diagrame
- folosite frecvent: 9
- esențiale: 6

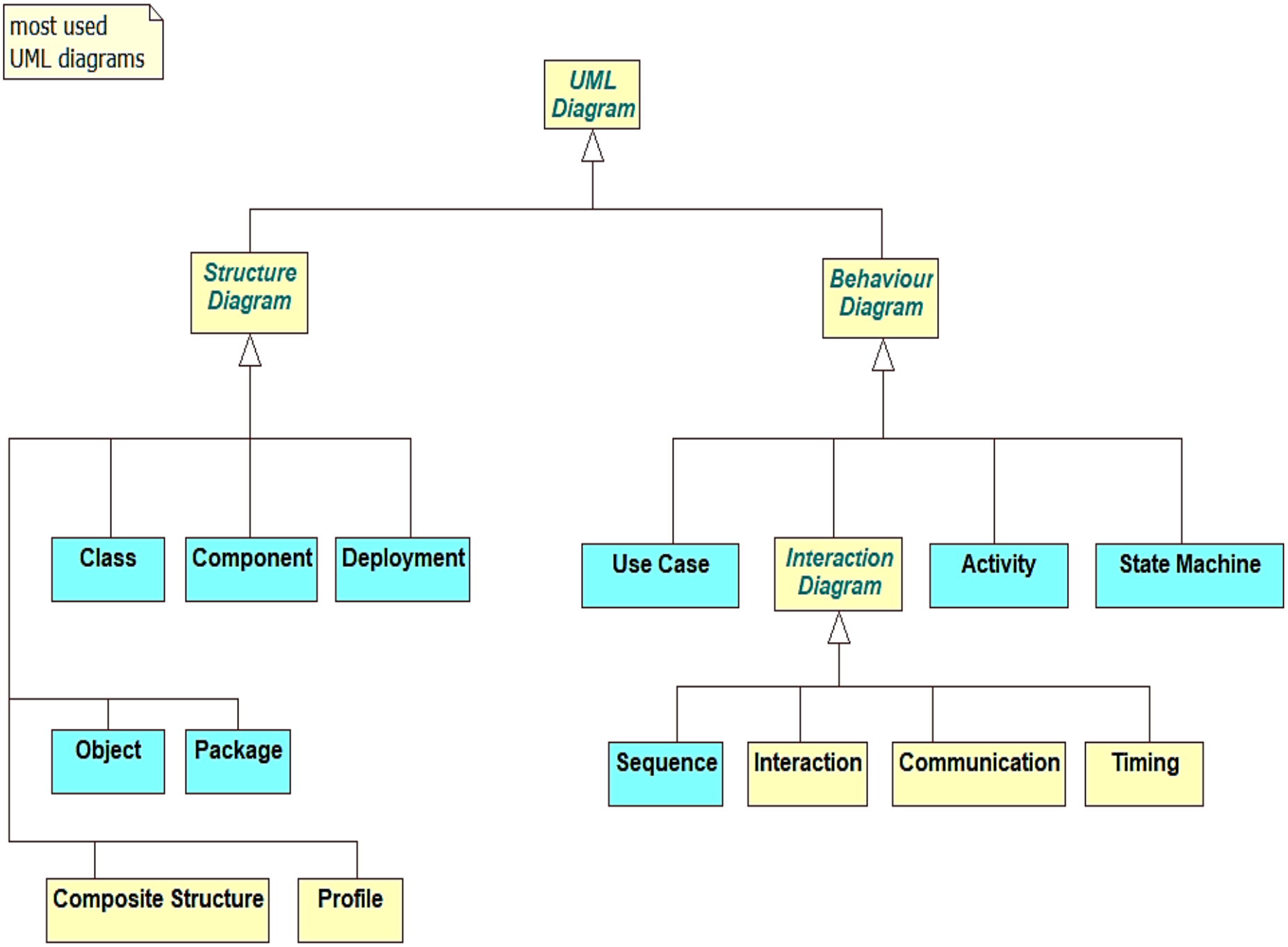
Unde îl folosim

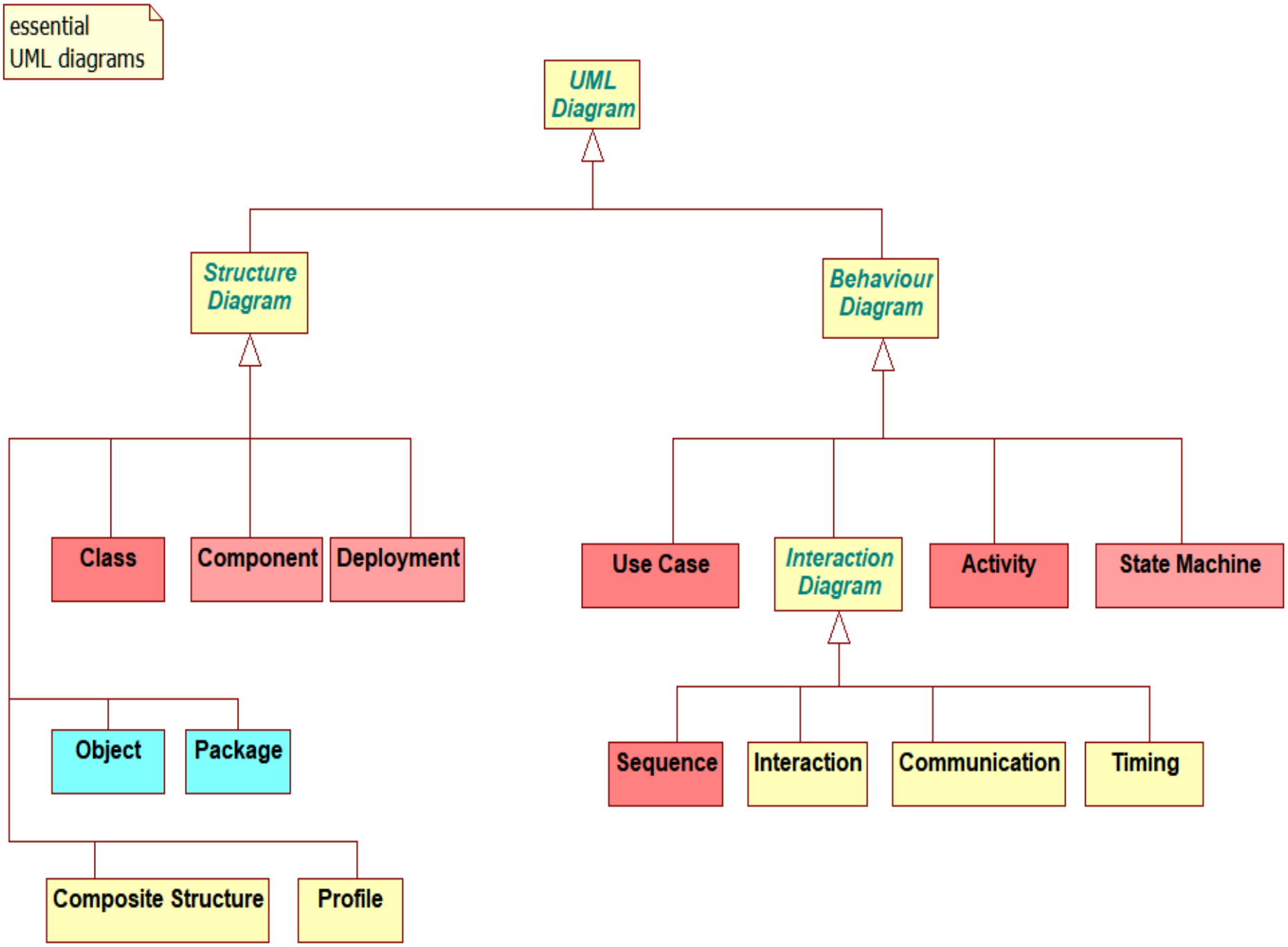
Analiză : comunicare, specificatii
Proiectare : comunicare, arhitectura
Implementare : ghid)
Mentenanta : [re]intelegere

Si.. chiar e util ?

DA ! daca îl folosim smart


all
UML diagrams
UML Diagram
Structure Diagram
Behaviour Diagram
Class
Component
Deployment
Object
Package
Composite Structure
Profile
Use Case
Interaction Diagram
Activity
State Machine
Sequence
Interaction
Communication
Timing


most used
UML diagrams
UML Diagram
Structure Diagram
Behaviour Diagram
Class
Component
Deployment
Object
Package
Composite Structure
Profile
Use Case
Interaction Diagram
Activity
State Machine
Sequence
Interaction
Communication
Timing


essential
UML diagrams
UML Diagram
Structure Diagram
Behaviour Diagram
Class
Component
Deployment
Use Case
Interaction Diagram
Activity
State Machine
Object
Package
Sequence
Interaction
Communication
Timing
Composite Structure
Profile

OK.. Să trecem în revistă..

Cazuri de utilizare

Secventa

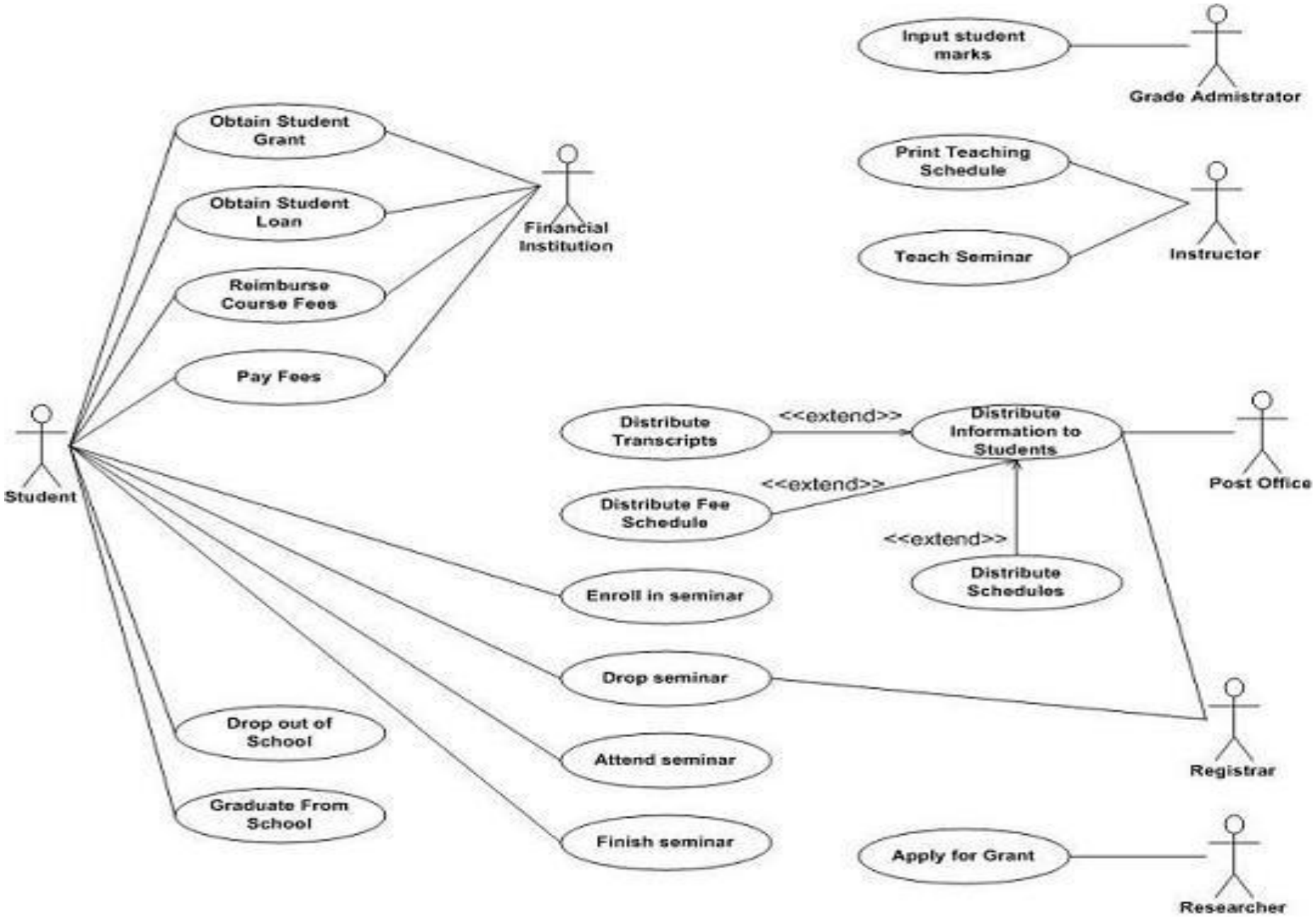
Clase

Activitate

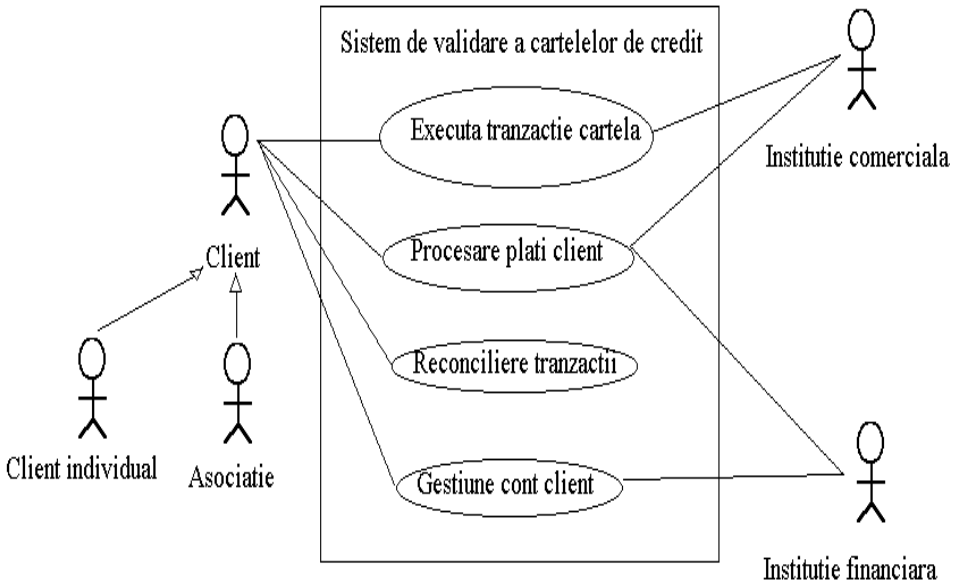
Stari

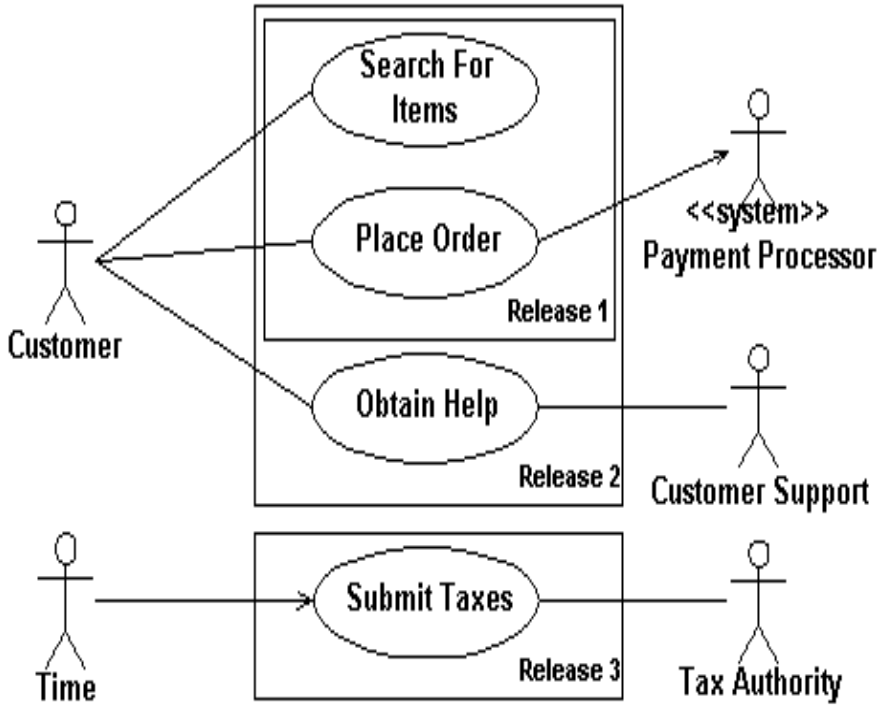
Componente si distributie

Cazuri de utilizare (use case d.)

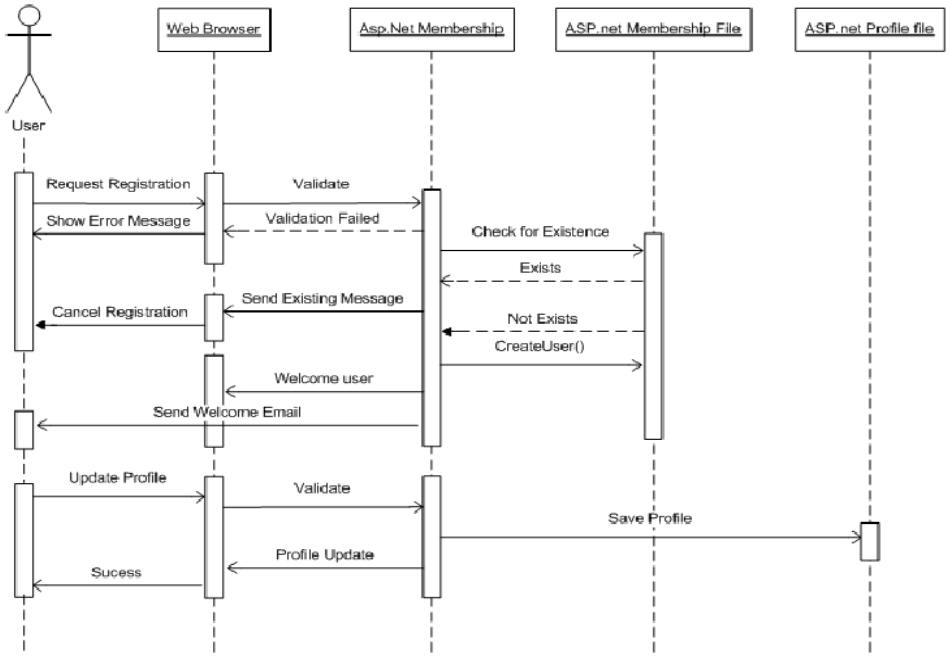


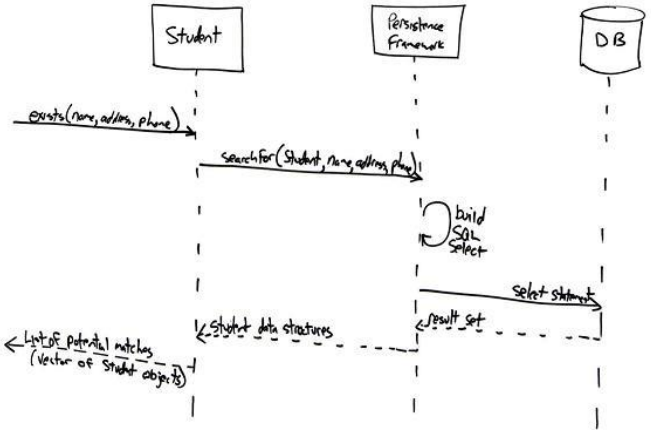
..Cazuri de utilizare (continuare)

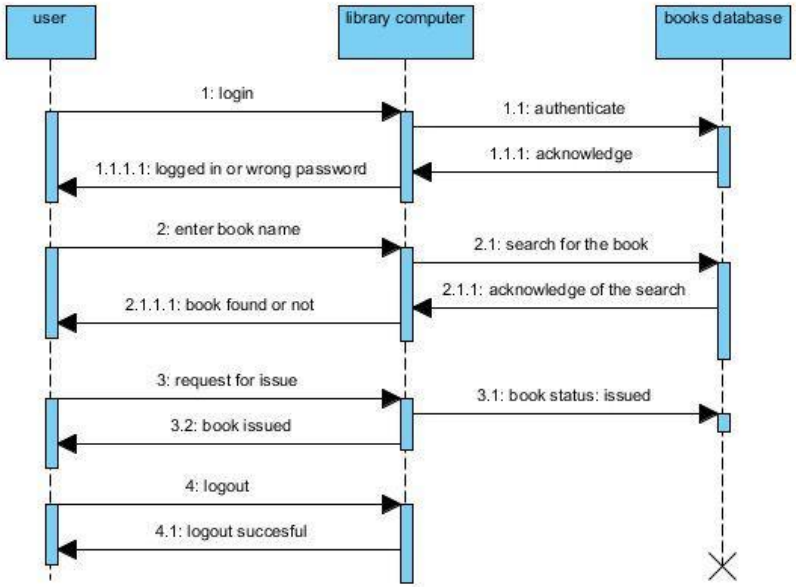


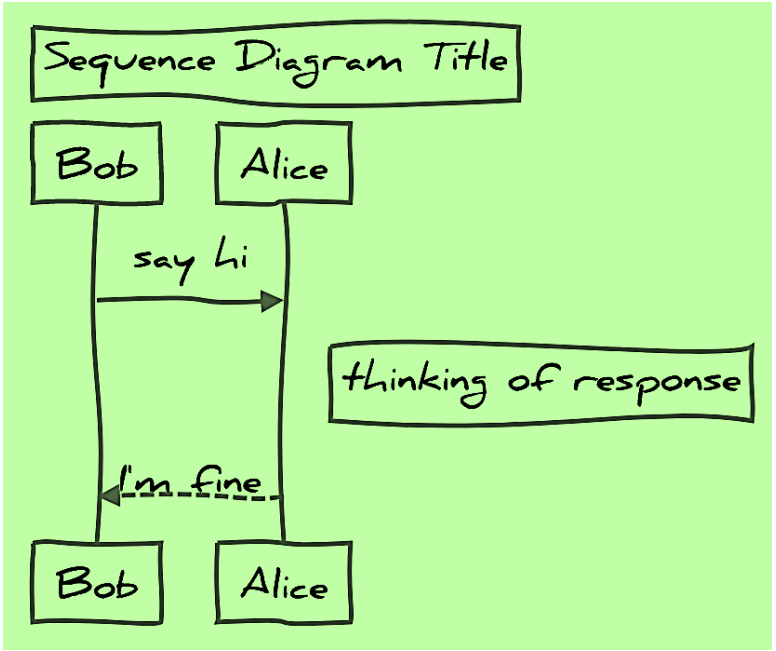


Secventa


User
Web Browser
Asp.Net Membership
ASP.net Membership File
ASP.net Profile file
Request Registration
Validate
Show Error Message
Validation Failed
Check for Existence
Exists
Send Existing Message
Cancel Registration
Not Exists
CreateUser()
Welcome user
Send Welcome Email
Update Profile
Validate
Save Profile
Profile Update
Sucess

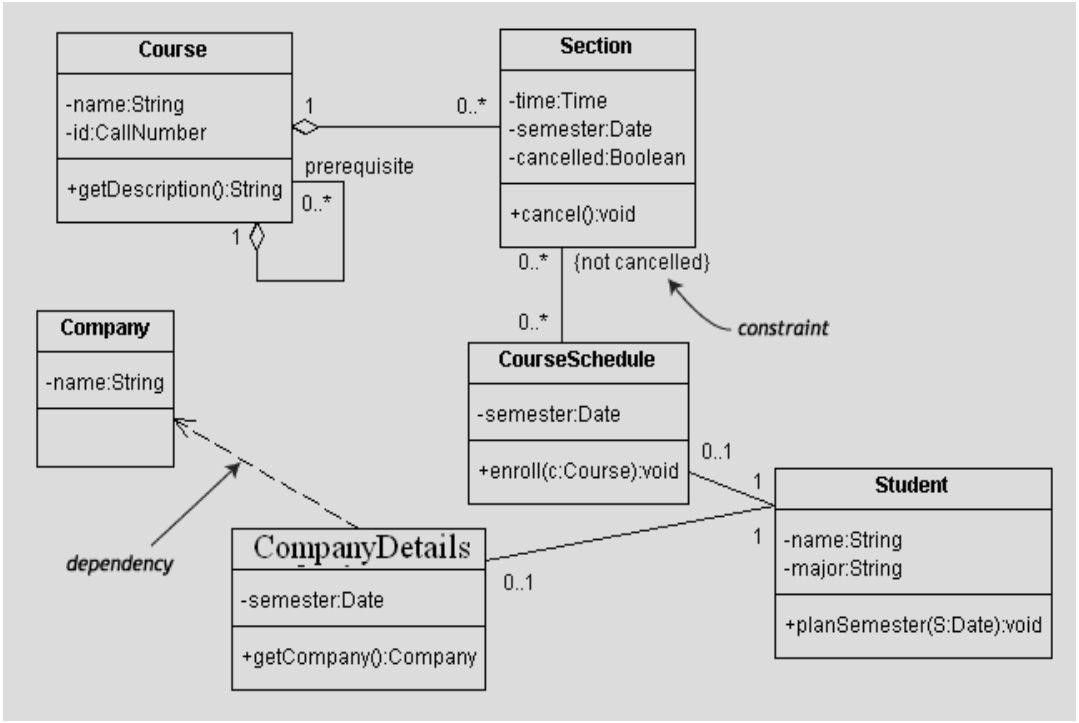

Student
Persistance Framework
DB
exists(name, address, phone)
searchFor(Student, name, address, phone)
build SQL select
select statement
result set
Student data structures
List of Potential matches (Vector of Student objects)

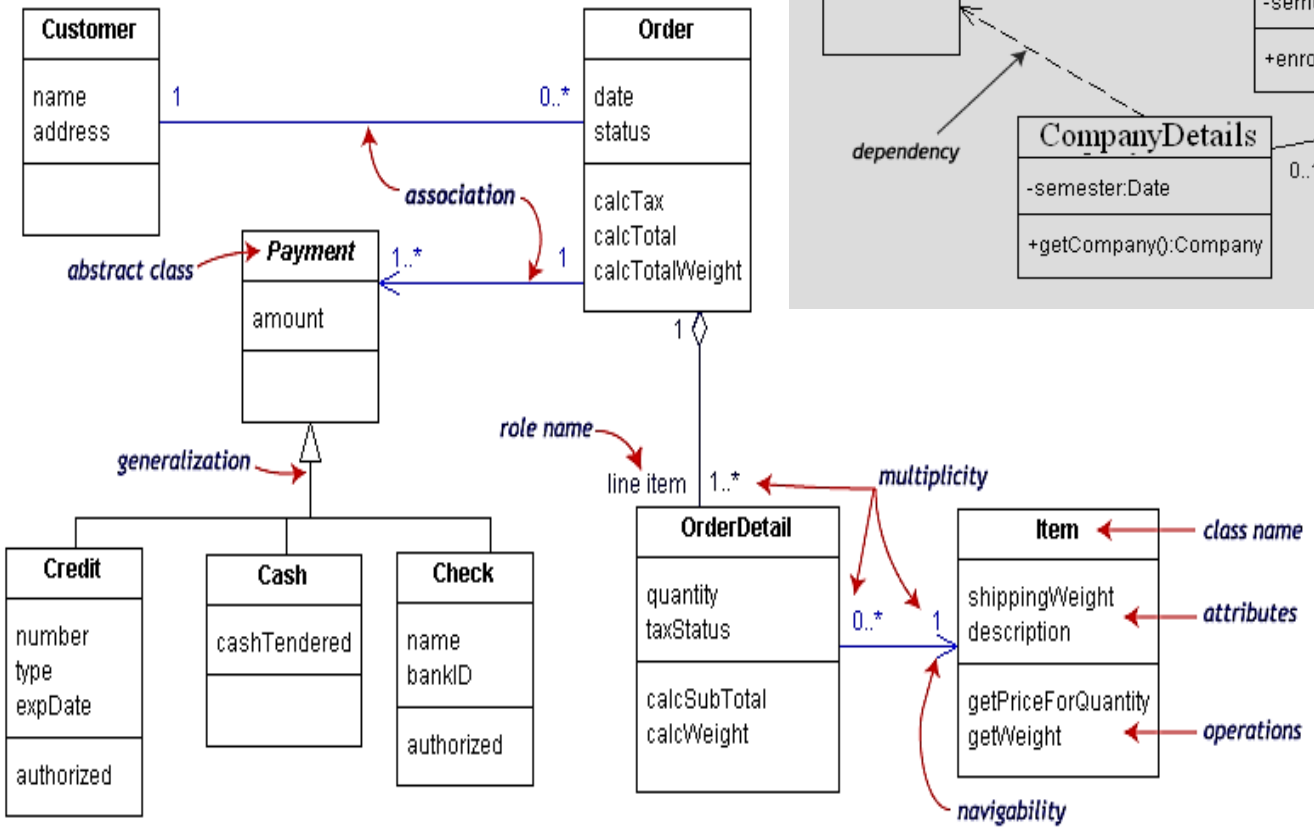

user
library computer
books database
1: login
1.1: authenticate
1.1.1: acknowledge
1.1.1.1: logged in or wrong password
2: enter book name
2.1: search for the book
2.1.1: acknowledge of the search
2.1.1.1: book found or not
3: request for issue
3.1: book status: issued
3.2: book issued
4: logout
4.1: logout succesful


Sequence Diagram Title
Bob
Alice
say hi
thinking of response
I'm fine
Bob
Alice

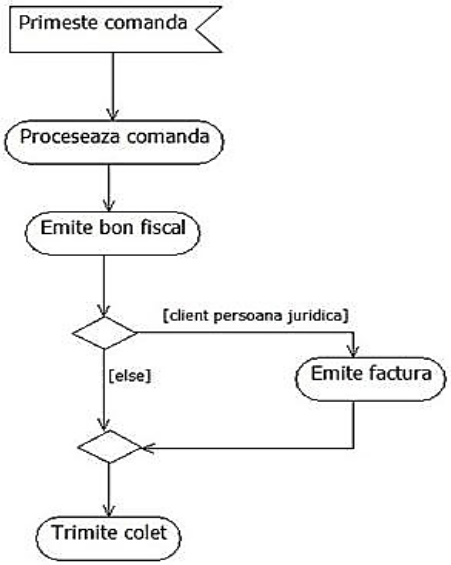
Clase

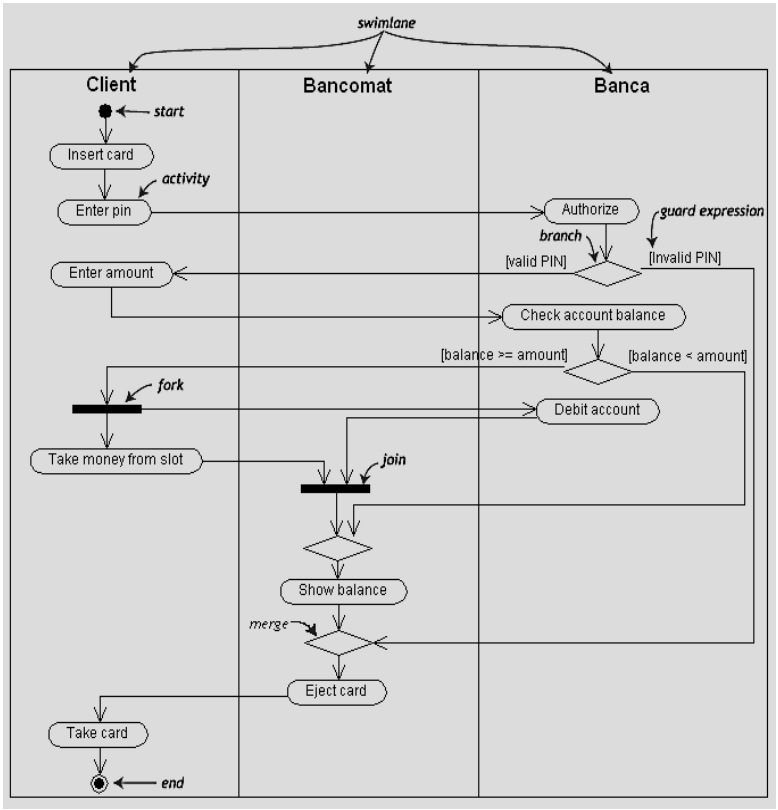
we should know these, right ?

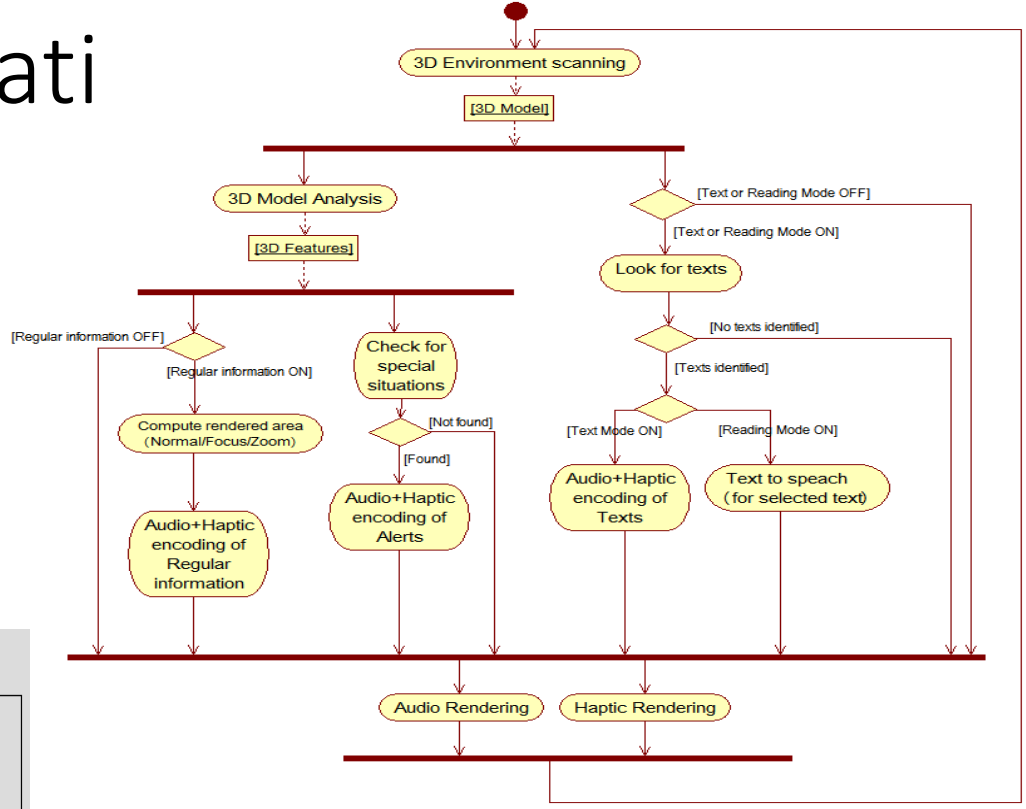




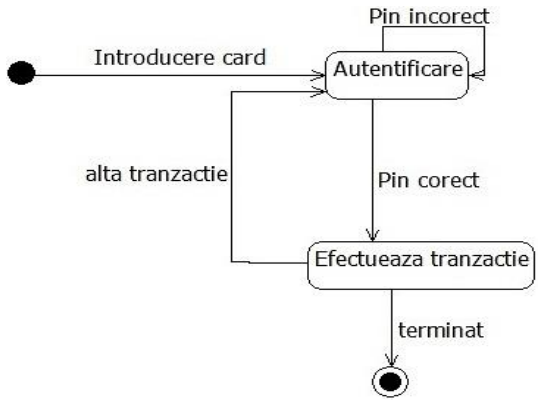
Activitati

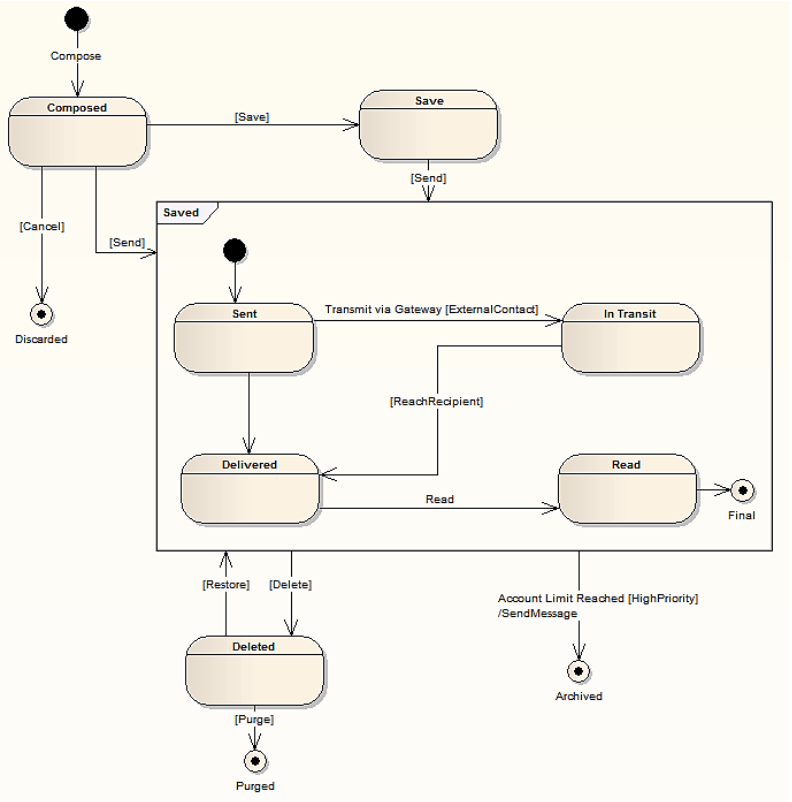

Primeste comanda
Proceseaza comanda
Emite bon fiscal
[client persoana juridica]
[else]
Emite factura
Trimite colet

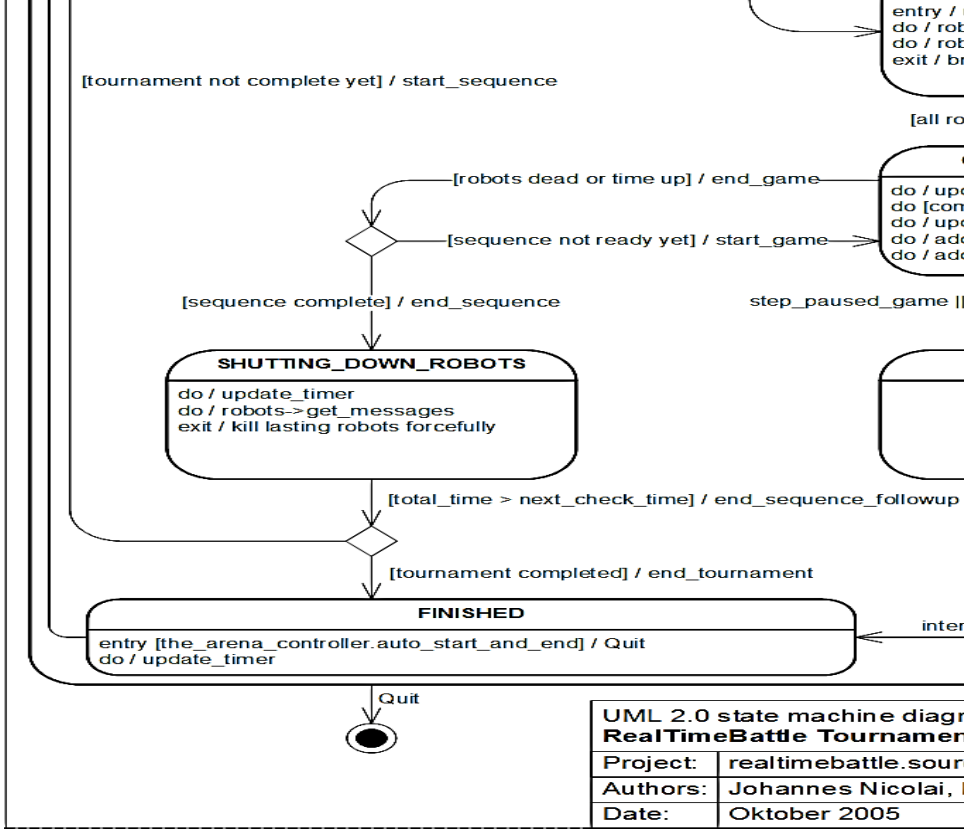

swimlane
Client
Bancomat
Banca
start
Insert card
activity
Enter pin
Authorize
guard expression
branch
[valid PIN]
[Invalid PIN]
Enter amount
Check account balance
[balance >= amount]
[balance < amount]
fork
Debit account
Take money from slot
join
Show balance
merge
Eject card
Take card
end

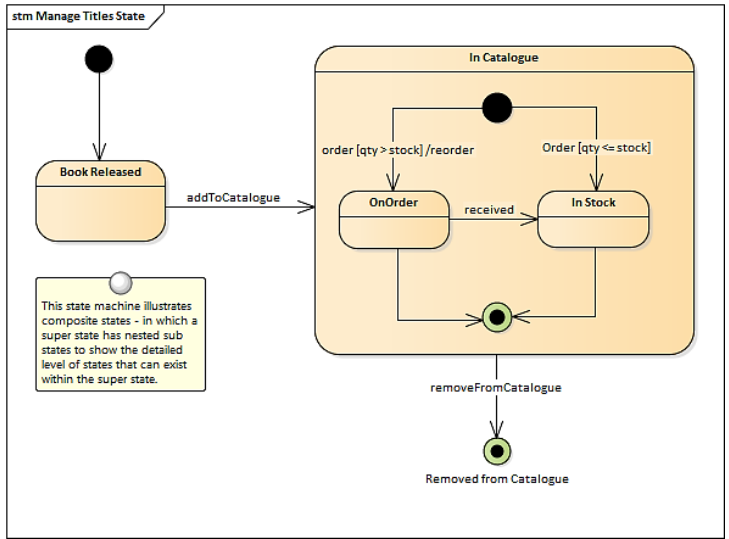

3D Environment scanning
[3D Model]
3D Model Analysis
[3D Features]
[Regular information OFF]
[Regular information ON]
Compute rendered area (Normal/Focus/Zoom)
Audio+Haptic encoding of Regular information
Check for special situations
[Not found]
[Found]
Audio+Haptic encoding of Alerts
[Text or Reading Mode OFF]
[Text or Reading Mode ON]
Look for texts
[No texts identified]
[Texts identified]
[Text Mode ON]
[Reading Mode ON]
Audio+Haptic encoding of Texts
Text to speach (for selected text)
Audio Rendering
Haptic Rendering

Stari (state machine)

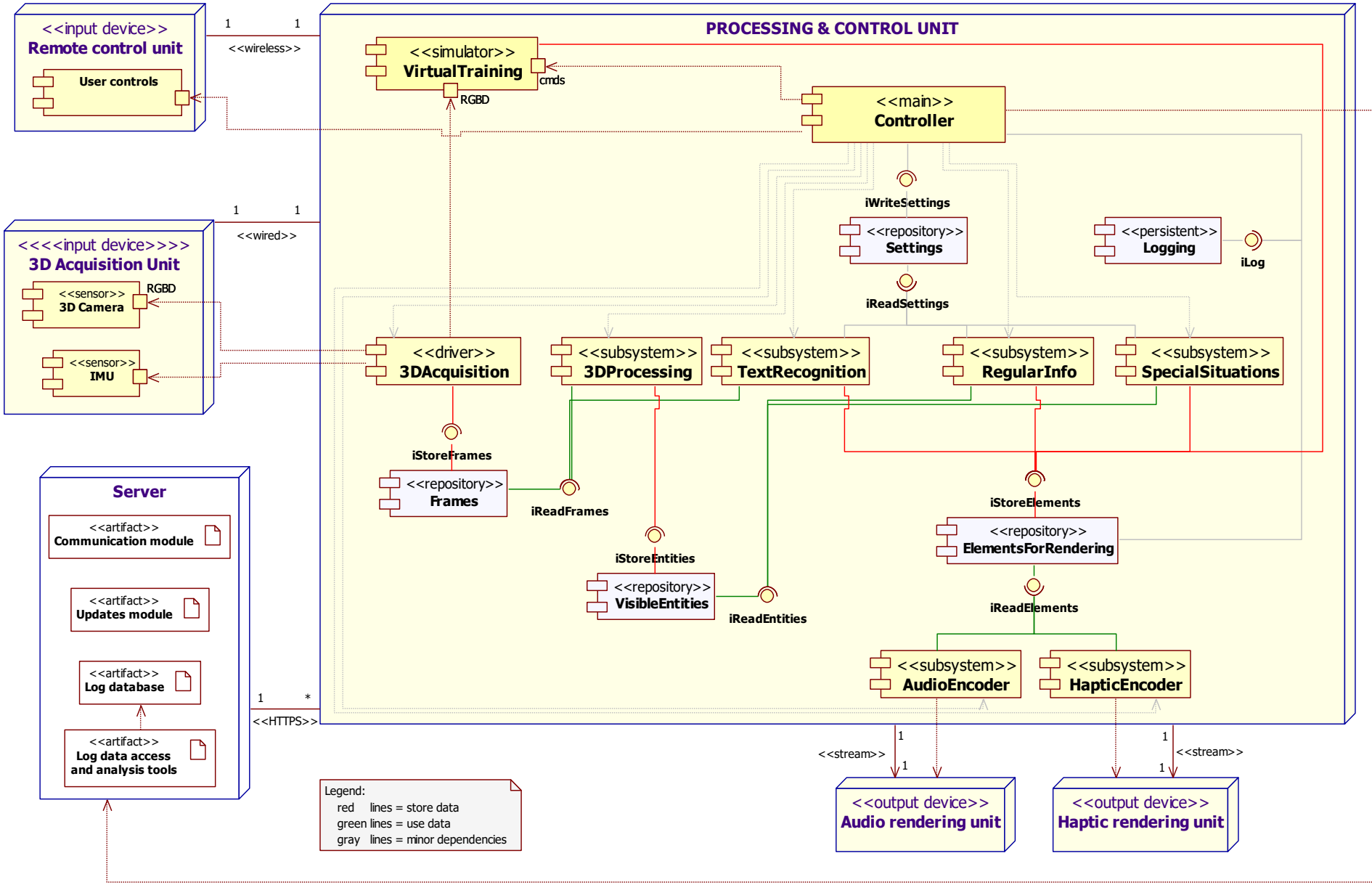








Componente si distributie



PRAGMATIC ?

- E un limbaj, nu o metoda (i.e. facem ce & cum vrem cu el)
- E complex, e puternic…. E si util ?!

Thumb rules.. Work most of the time.. Yet sometimes the opposite is justified

Good UML ? (focused, agile)	Bad UML ? (classical)
Efficient	**Overkill**, drain too many resources
Draw the essentials	Diagram everything
Basic notations	Advanced notations
Read at a glance	Hard to understand
Facilitate communication	Slows down communication
Focus	**Get lost in details**
Easy to maintain	Hard to maintain

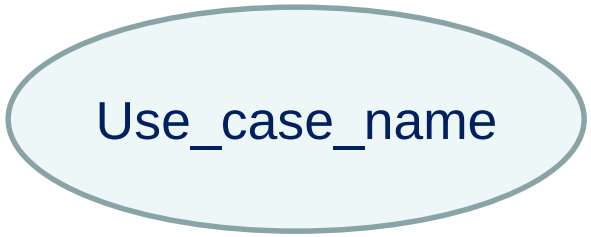
UML in Specificarea Cerintelor

- Cazuri de utilizare
- Diagrame de cazuri de utilizare
- Diagrame de secventa
- Diagrame de stari
- Diagrame de clase conceptuale

Modelul cazurilor de utilizare

- **Se elaboreaza in fazele initiale ale analizei**
- **Exprima interactiunea**
  - SISTEM <->UTILIZATOR

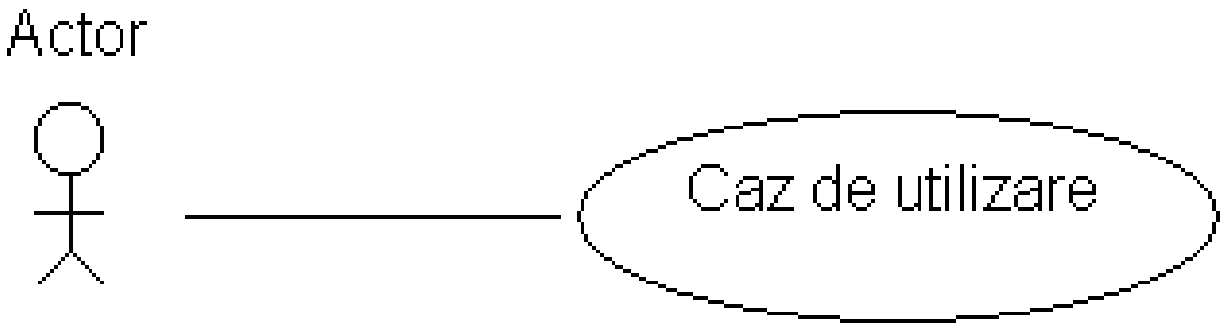
  sau
  - SISTEM<->componenta EXTERNA
- **Modelul cazurilor de utilizare include:**
  - actorii
  - scenarii
  - cazurile de utilizare
  - diagramele de cazuri de utilizare



Actor

Actor = rol pe care o entitate externa il joaca in raport cu sistemul:

- Utilizatorii directi ai sistemului (un utilizator poate juca mai multe roluri)
- Un echipament extern sau alt sistem cu care sistemul analizat comunica



Scenarii & Cazuri de utilizare

SCENARIU = o secventa de pasi care descrie o posibila interactiune dintre un sistem si un actor

CAZ DE UTILIZARE = abstractizare a unei interactiuni dintre un actor si sistem:

- Generalizeaza toate scenariile pentru o anumita interactiune
- Descrie interactiunea la modul general, fara a intra in detaliile fiecarui scenariu

Studiu de caz:

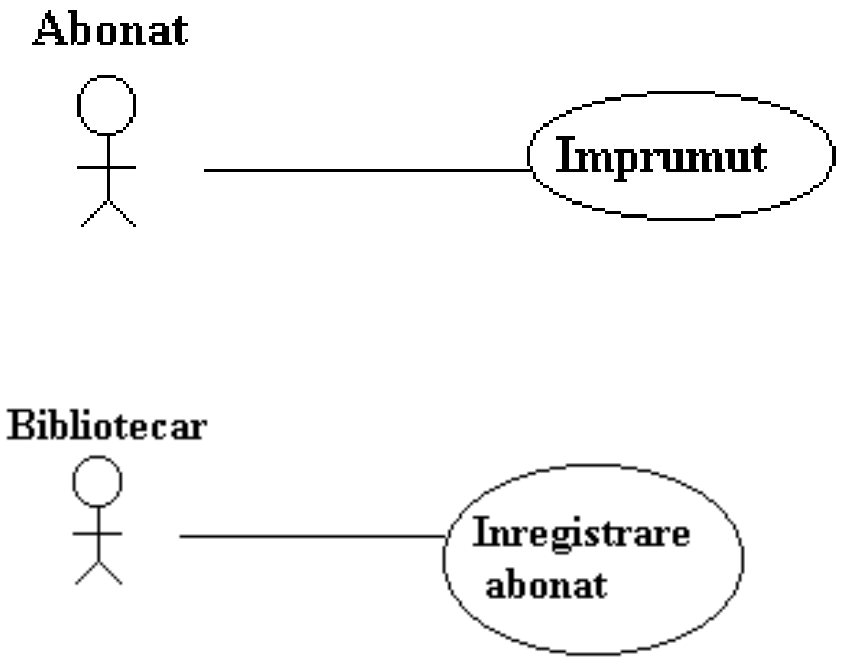
Sistem de gestiune electronica a cartilor din o biblioteca:

- **Sistemul urmeaza sa fie utilizat de doua categorii de utilizatori: bibliotecarii si abonatii.**
- **Bibliotecarii acceseaza sistemul pentru a inregistra abonati si carti noi sau pentru a elimina carti din evidenta.**
- **Abonatii pot cere informatii despre diferite carti si pot cere imprumutarea unei carti.**
- **Sistemul trebuie sa pastreze evidenta abonatilor, a cartilor imprumutate de fiecare abonat si alte informatii.**

Actorii:

- Abonat
- Bibliotecar

Cazurile de utilizare:



SCENARII ale cazului de utilizare “Imprumut”:

(1)

- Un utilizator care doreste sa imprumute o carte completeaza rubricile afectate numelui si prenumelui sau apoi apasa butonul “Submit”.
- Sistemul preia datele si verifica daca utilizatorul este inregistrat ca abonat.
- Utilizatorul primeste mesajul: „Nu sunteti inregistrat ca abonat. Efectuati procedura de inregistrare”.

(2)

- Un utilizator care doreste sa imprumute o carte completeaza rubricile afectate numelui si prenumelui sau apoi apasa butonul “Submit”.
- Sistemul preia datele si verifica daca utilizatorul este inregistrat ca abonat.
- Utilizatorul primeste mesajul: „Ati depasit numarul maxim de carti imprumutate. Restituiti o parte dintre ele”.

(3)

- Un utilizator care doreste sa imprumute o carte completeaza rubricile afectate numelui si prenumelui sau apoi apasa butonul “Submit”.
- Sistemul preia datele si verifica daca utilizatorul este inregistrat ca abonat.
- Sistemul afiseaza urmatoarea pagina, continand formularul de imprumut.
- Abonatul completeaza formularul de imprumut, cu titlul cartii, numele si prenumele autorului si codul ISBN al cartii apoi apasa butonul “Submit”.
- Sistemul preia datele si cauta cartea.
- Utilizatorul primeste mesajul: „Cartea nu exista in bibliotecile noastre”.

etc. (se descriu toate scenariile)

Descriere caz de utilizare “Imprumut”: (abstractizare scenarii corelate; descriere prin secventa tipica de pasi si alternativele):

Secventa tipica:

1. Un utilizator care doreste sa imprumute o carte completeaza rubricile afectate numelui si prenumelui sau apoi apasa butonul “Submit”.
2. Sistemul preia datele si verifica daca utilizatorul este inregistrat ca abonat.
3. Sistemul afiseaza urmatoarea pagina, continand formularul de imprumut.
4. Abonatul completeaza formularul de imprumut, cu titlul cartii, numele si prenumele autorului si codul ISBN al cartii apoi apasa butonul “Submit”.
5. Sistemul preia datele si cauta cartea.
6. Sistemul inregistreaza imprumutul.
7. Sistemul afiseaza datele necesare imprumutului.

Alternative:

- Alternativa “Acces ne-autorizat”:
  - La pasul 2: Utilizatorul nu este inregistrat ca abonat si atunci sesiunea este incheiata de sistem cu un mesaj in care utilizatorul este invitat sa se inregistreze.
- Alternativa “Imprumutul nu este posibil”
  - La pasul 5:
    - 5a) Cartea nu este gasita. Sistemul afiseaza un mesaj si sesiunea este incheiata.
    - 5b) Cartea este gasita dar abonatul a imprumutat deja numarul maxim admis de carti.

    Sistemul afiseaza un mesaj si incheie sesiunea.

Ce trebuie sa precizeze un C.U.

UML admite variatii in privinta descrierii cazurilor de utilizare.

Descrierea unui caz de utilizare trebuie sa precizeze:

- cum si cand incepe si se termina cazul de utilizare (evenimente care marcheaza inceputul si sfarsitul cazului)
- cand au loc interactiunile cu actorii si informatiile schimbate (comunicate intre actori – sistem) in timpul fiecarei interaciuni
- fluxul de baza (comportamentul de baza) si alternativele fiecarei interactiuni;
- repetarile de comportament, care pot fi descrise prin formulari de tipul:

  ciclu

  ----

  sfarsit ciclu

  sau

  cat timp

  ----

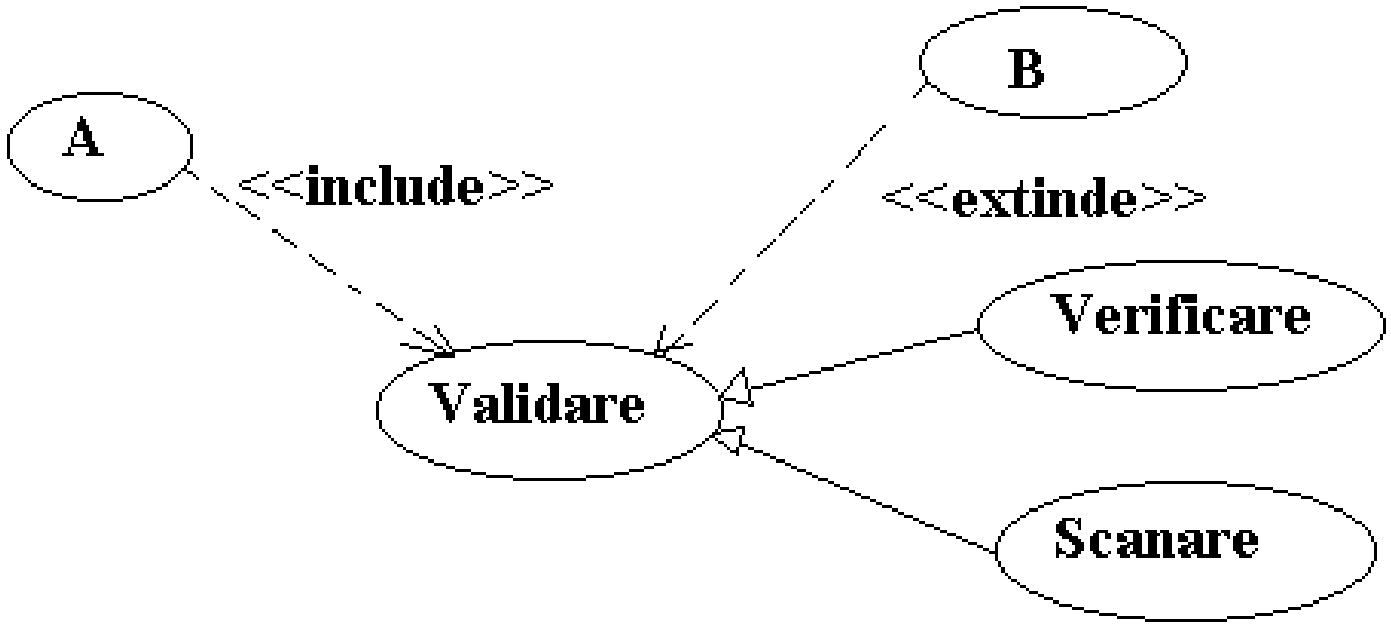
  sfarsit cat timp

Diagrame UML de C.U.

- **Actori**
  - multiplicitati
  - Relatii intre actori: doar generalizare (nu exista interactiuni intre actori)
- **Cazuri de utilizare**
  - Relatii intre cazuri de utilizare: Generalizare, Includere, Extindere
- **Asocieri intre actori si cazuri de utilizare**
- **System boundary boxes:**
  - Evidentierea limitelor sistemului
  - Precizarea release-urilor sau a unor componente ale sistemului
- **Pachete de cazuri de utilizare**
  - organizarea in grupuri de functionalitati
  - simplificarea diagramelor

Exemple de d.C.U.

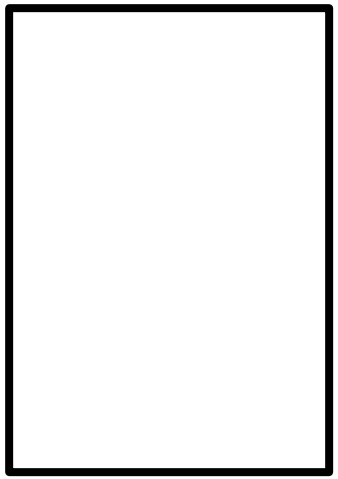
Relatii de reutilizare intre C.U.



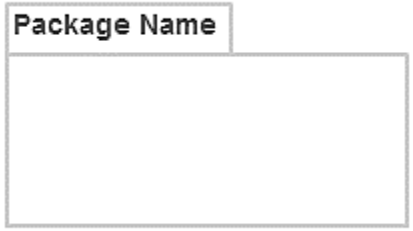
“Sistem”, Pachet

Sistem: dreptunghi ce incadreaza toate cazurile, sau un grup din ele:

- ✓ Definirea sistemului dezvoltat in mediul sau de operare
- ✓ Indicarea release-urilor



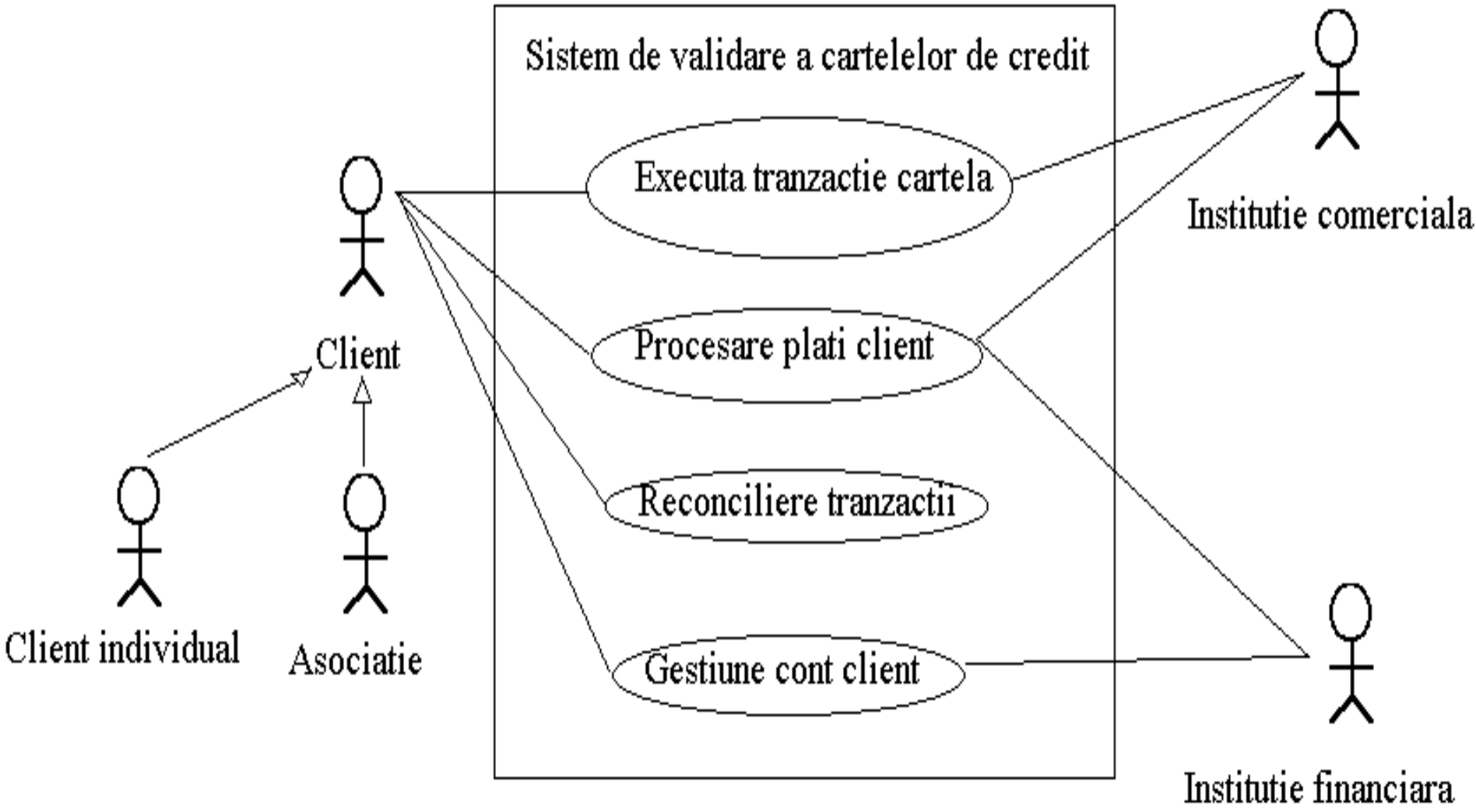
Package: grupeaza cazuri de utilizare strans legate



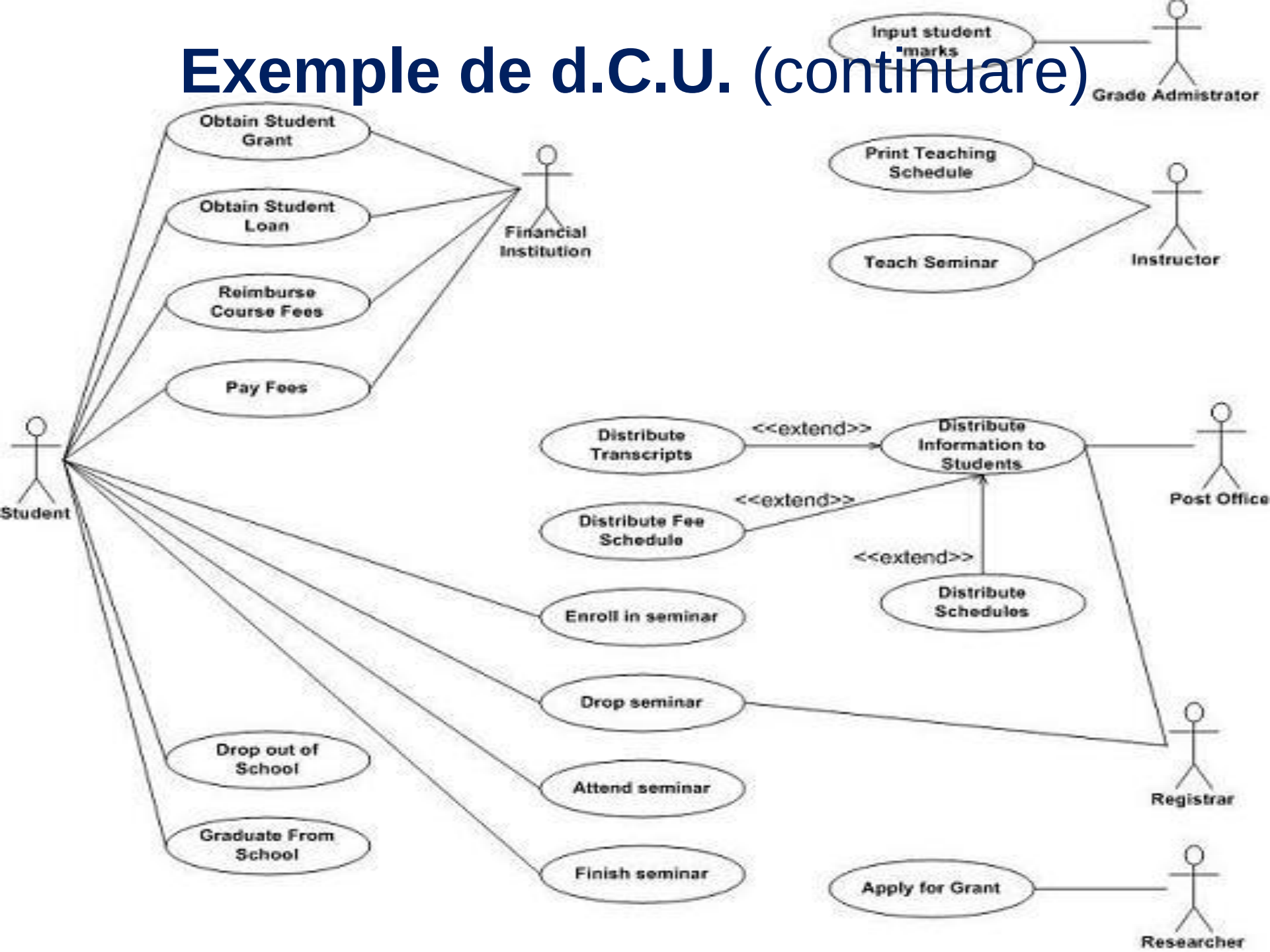
Reguli simple

- **Nume relevante si concise**
  - Actorii – substantive [+adjective]
  - Cazurile de utilizare – incep cu un verb
- **Layout-ul trebuie sa:**
  - fie usor de inteles
  - accentueze importanta elementelor
    - actorii principali in stanga
    - c.U. sa curga in ordinea logica
    - actorii derivati se plaseaza sub cei de baza
- **In primele faze ale analizei, nu trebuie detaliate exagerat**, pentru ca devin greoi de urmarit pentru beneficiar
- **Legaturile**:
  - Tip linie:
    - include si extend intre C.U. sunt cu linie intrerupta
    - restul cu linie normala
  - Orintare:
    - Actor – C. U. nu sunt orientate
    - restul sunt orientate (au sageata la capat)

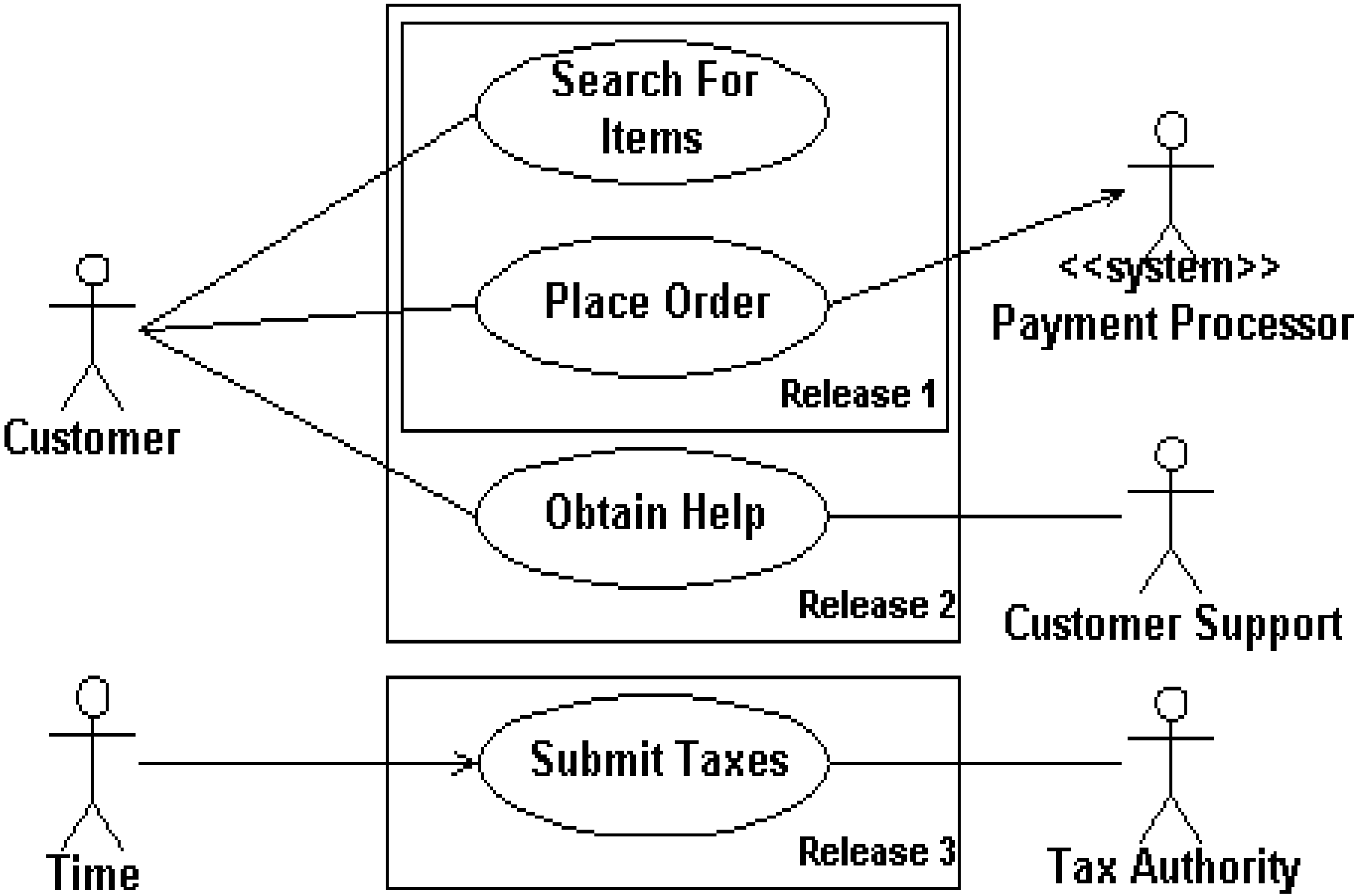
Exemple de d.C.U. (continuare)

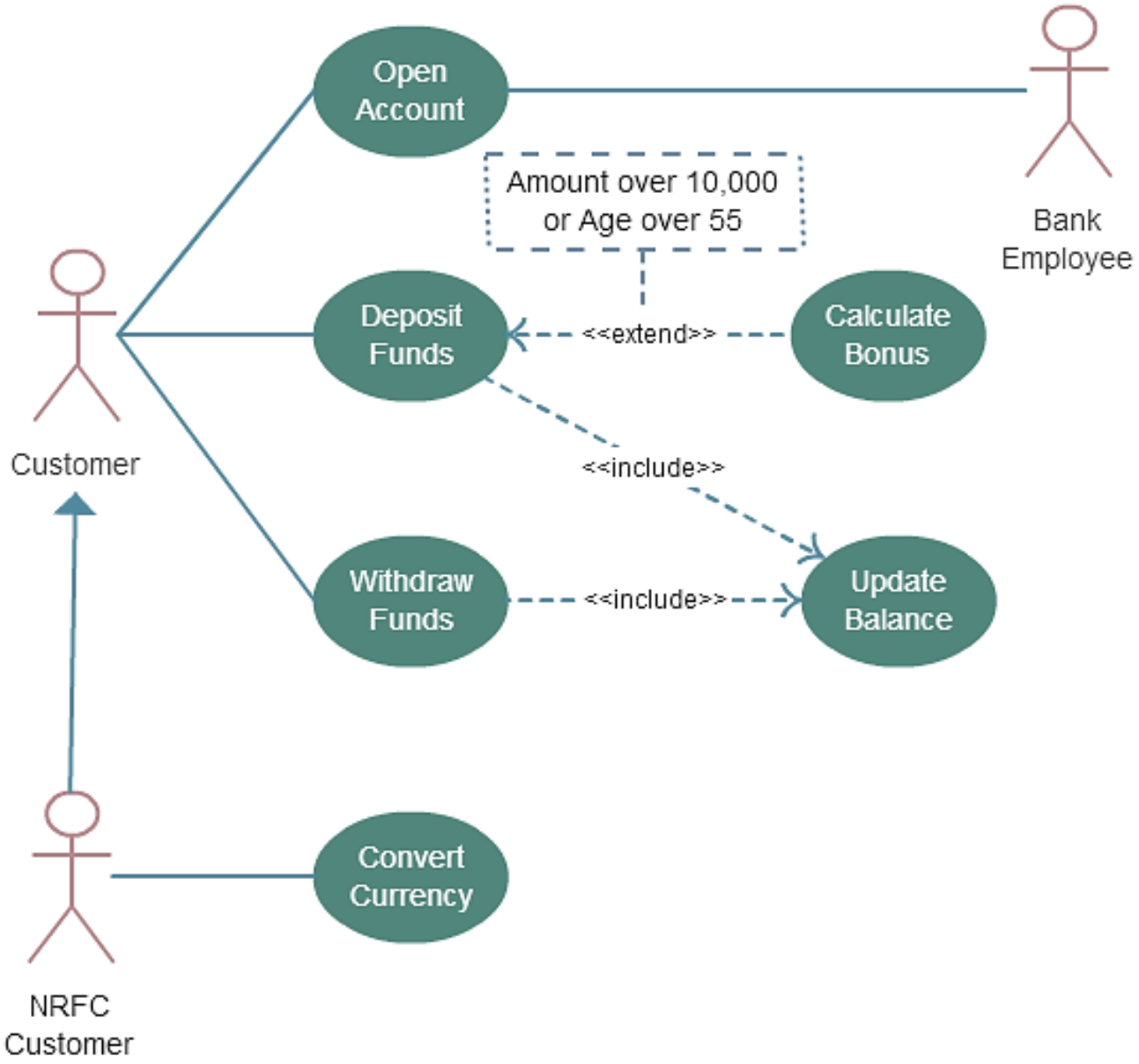


Exemple de d.C.U. (continuare)

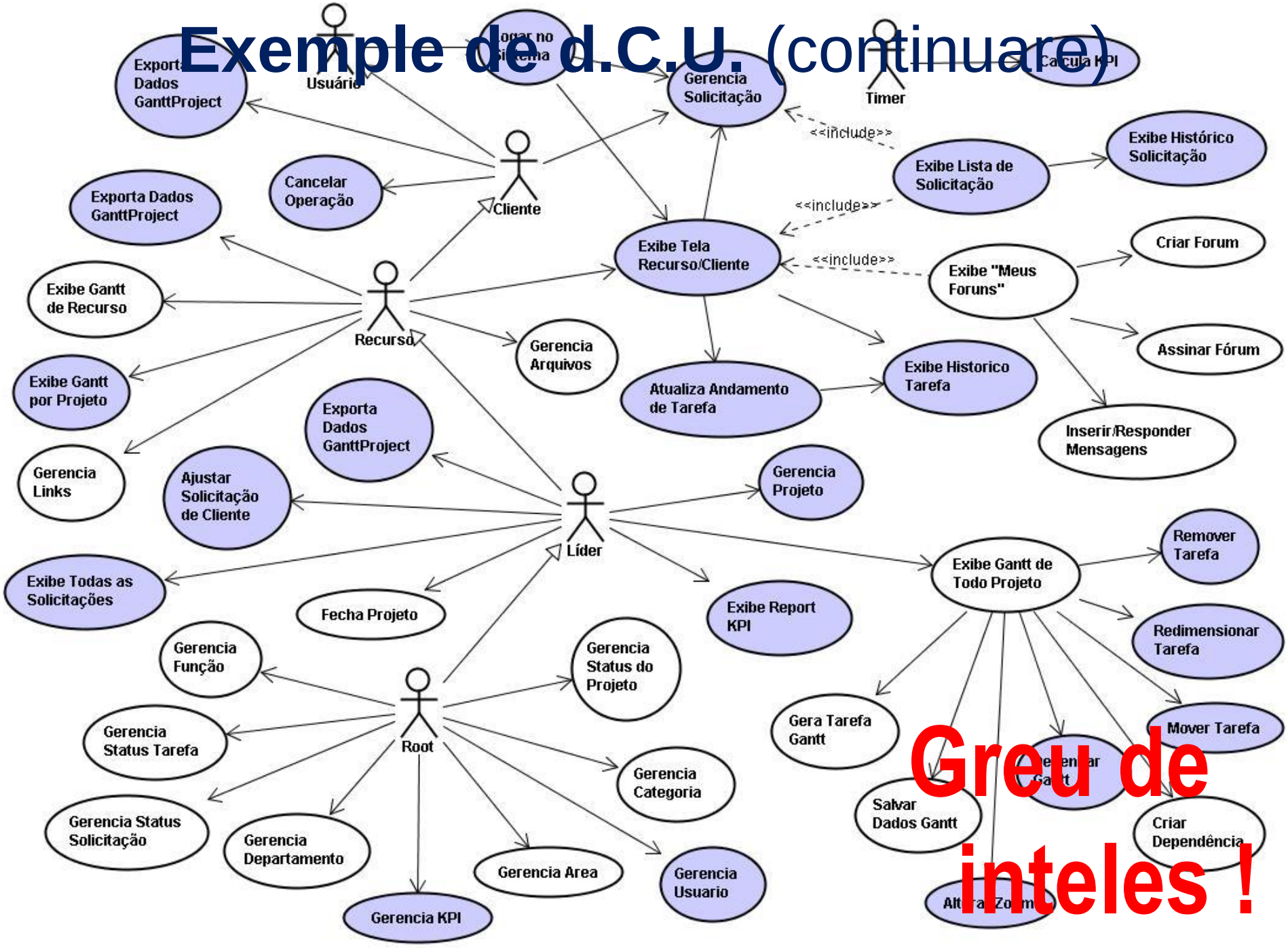


Exemple de d.C.U. (continuare)



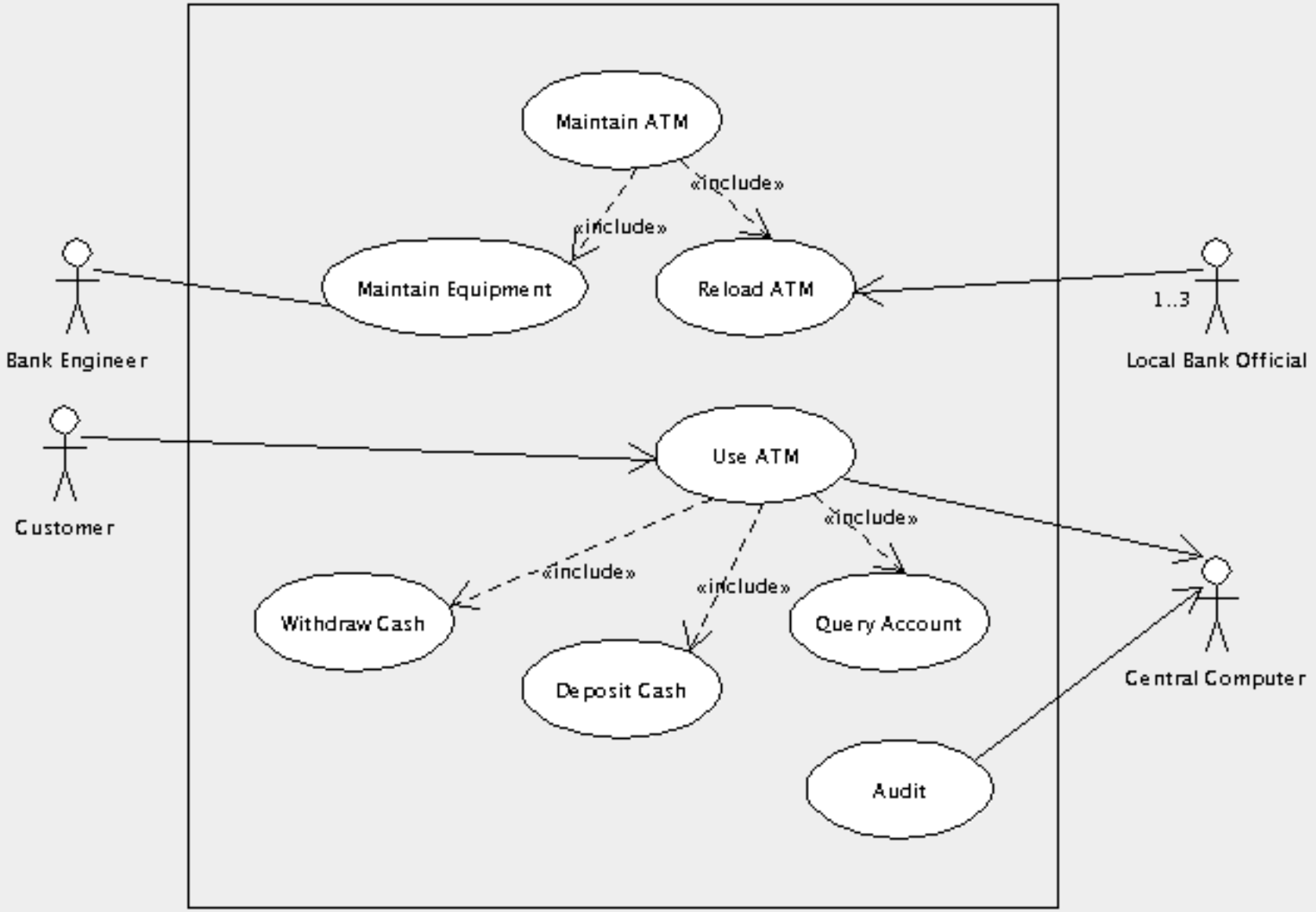

Open Account
Bank Employee
Amount over 10,000 or Age over 55
Deposit Funds
<<extend>>
Calculate Bonus
Customer
<<include>>
Withdraw Funds
<<include>>
Update Balance
Convert Currency
NRFC Customer

Exemple de d.C.U. (continuare)

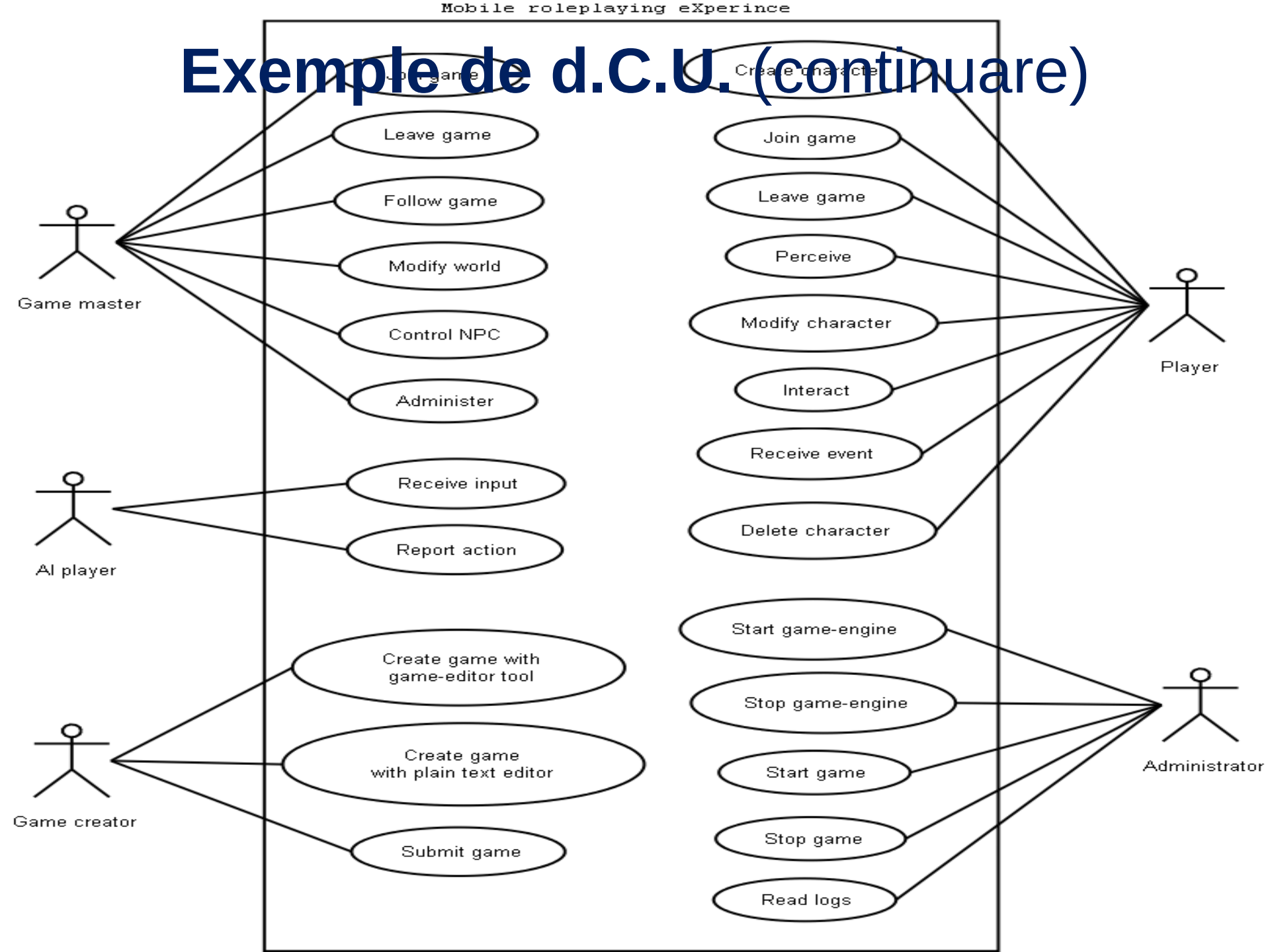


Greu de inteles !

Exemple de d.C.U. (continuare)



Exemple de d.C.U. (continuare)



Utilitatea modelului C.U.

- ❑ Mijloc de comunicare intre analist si client,experti ai domeniului
- ❑ Se construieste in etapa de definire a cerintelor utilizator
- ✓ Evidentierea functiilor
- ✓ Desprinderea cerintelor
- ✓ Determinarea contextului sistemului

Lecturi suplimentare

- http://www.agilemodeling.com/artifacts/useCaseDiagram.htm

- **Use cases vs. User stories**
  - Use cases vs user stories in Agile development
  - Has “Agile” Killed “Use Cases”?

    “No, use cases are *not*dead!

    In fact, when properly utilized, use cases offer the most efficient and clear way to document detailed functional requirements. They can save your team valuable time, while ensuring the delivered software meets the needs of users.”

Diagrame de interactiune

2 categorii, aproape echivalente, diferite prin layout si scop:

- Diagrame de secventa
- Diagrame de colaborare

Se folosesc in mai multe etape ale dezvoltarii:

- definirea cerintelor
- proiectare arhitecturala
- proiectare de detaliu

In etapa de definire a cerintelor, redau:

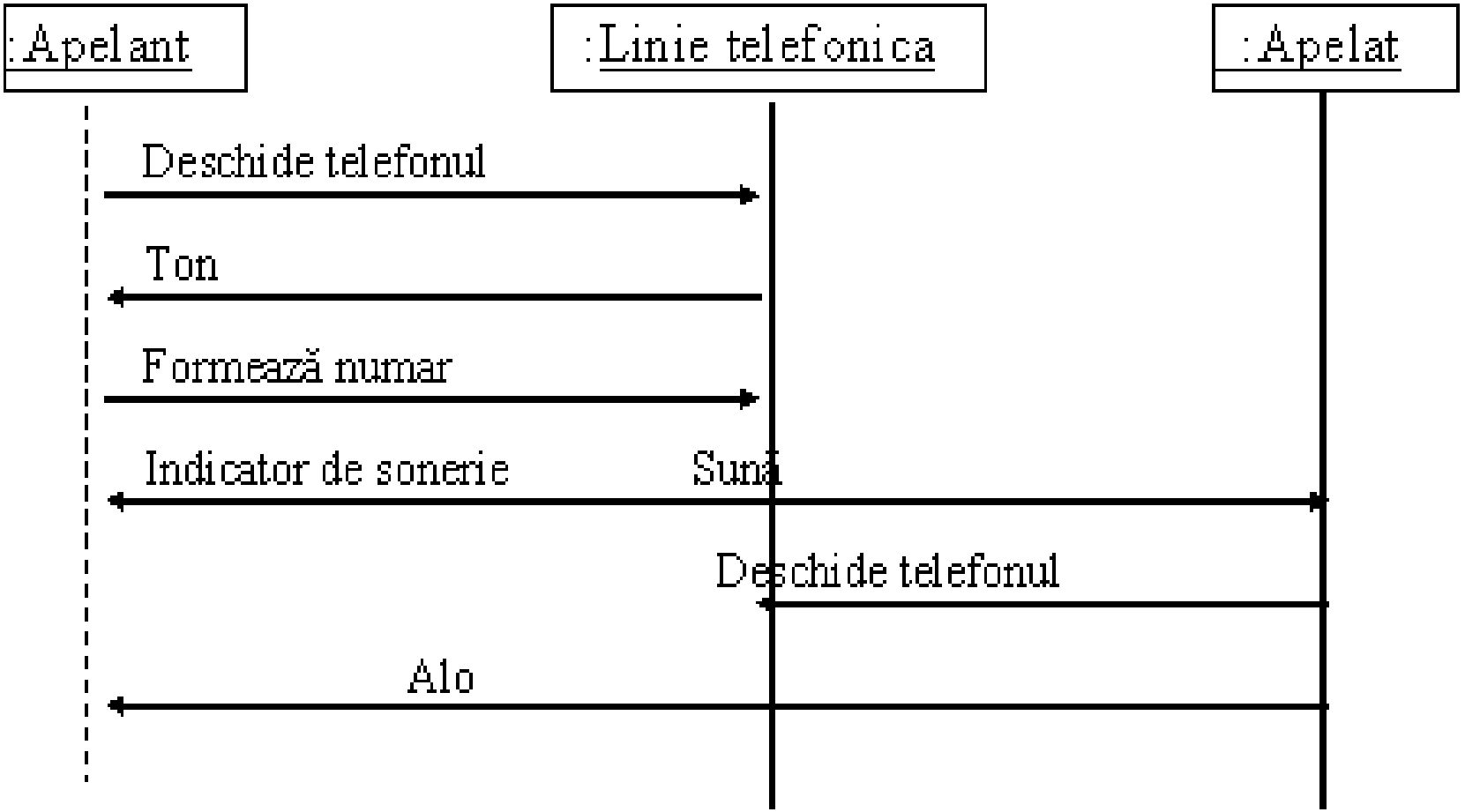
- Interactiunile dintre actori si sistem (scenarii)
- Interactiunile dintre obiectele din universul problemei

Obiecte

Comportamentul unui obiect, ca urmare a unei stimulari externe, este reprezentat prin operatii.

Operatiile unui obiect sunt declansate prin mesaje trimise de actori sau alte obiecte.

Reprezentarea scenariilor prin diagrame de secventa

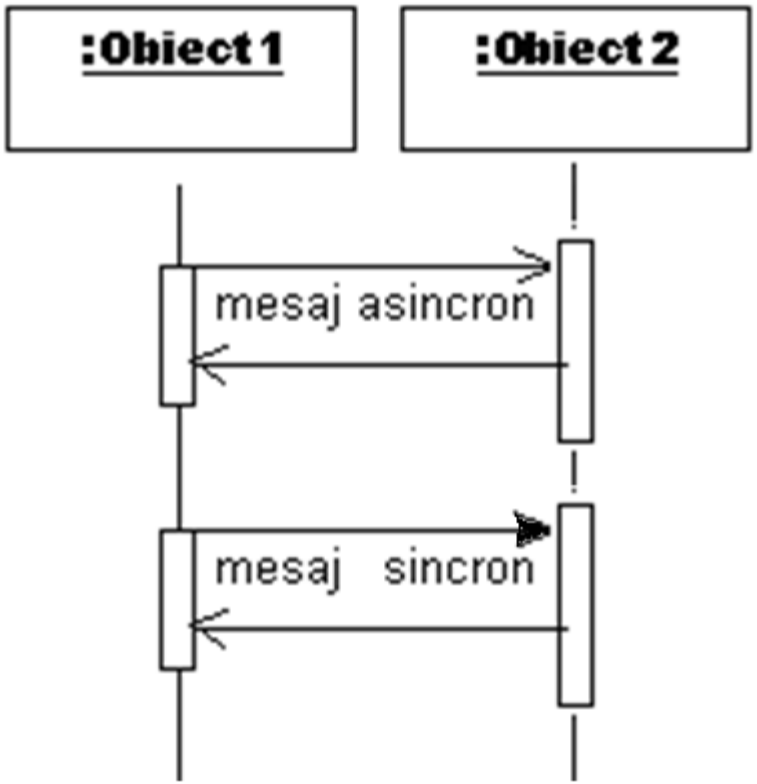


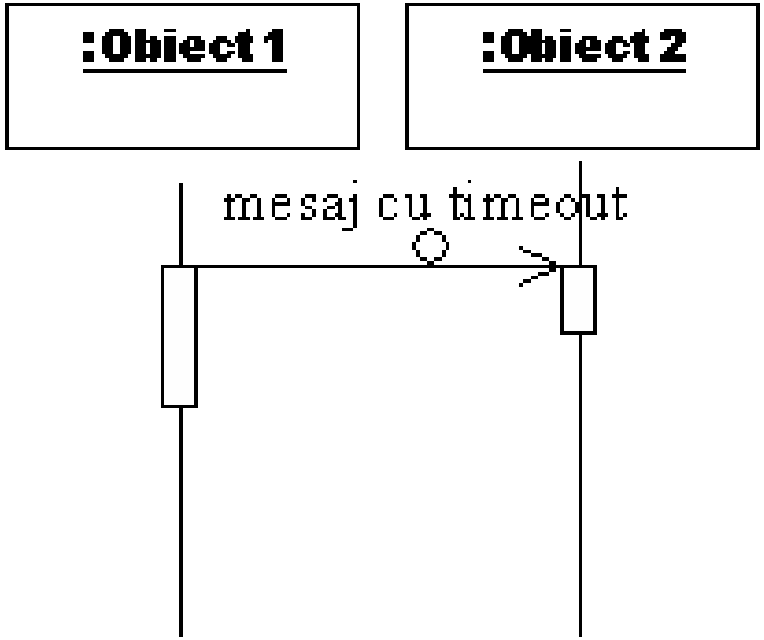
Diagramă de secvență

Elemente diagrame de secventa

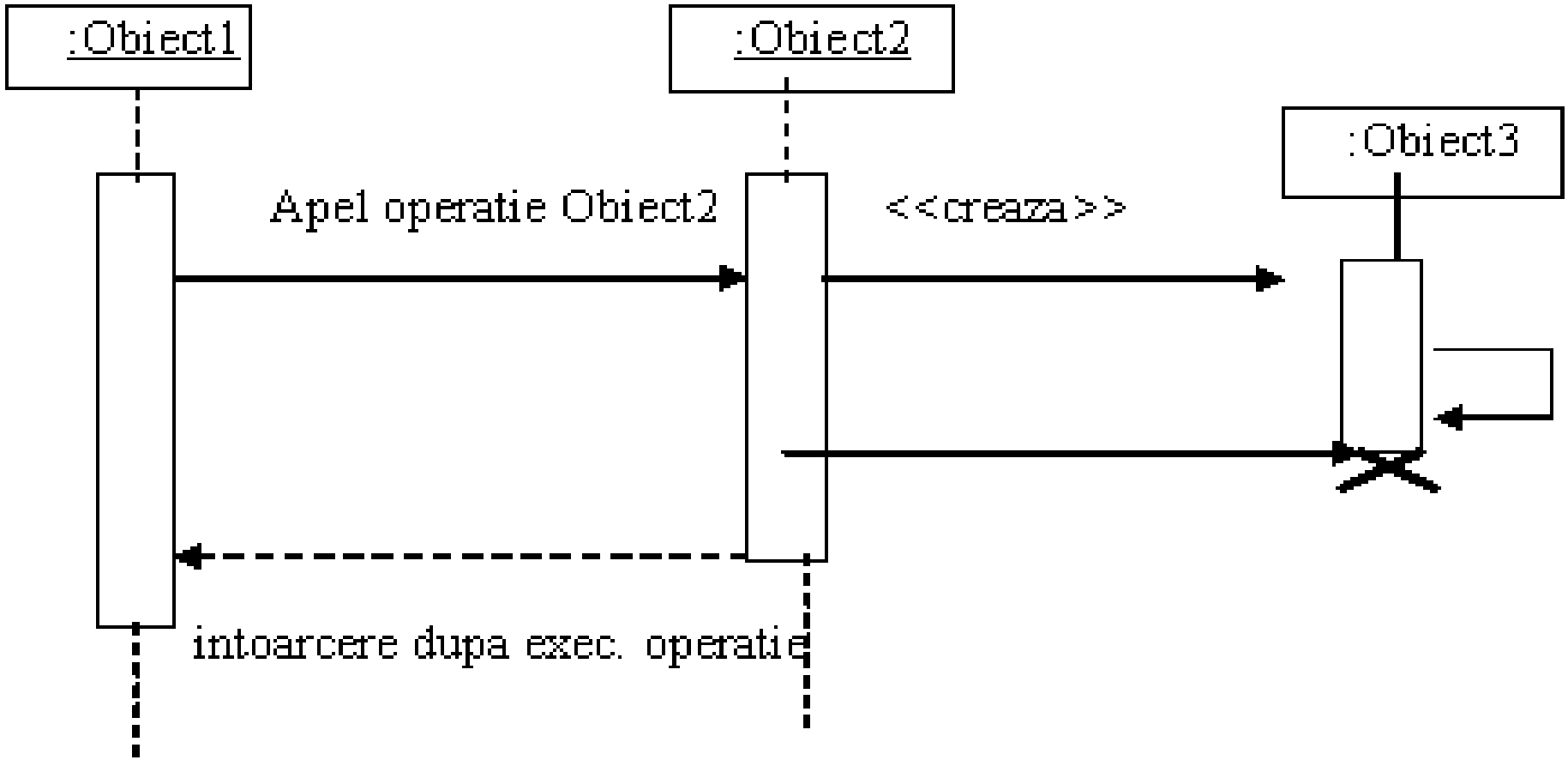
- **Obiecte, actori, sistemul**
- **Linii de viata**
- **Apeluri de operatii (transmitere mesaje)**
- **Perioade de activitate**
- **Creare si distrugere obiecte**
- **Blocuri decizionale (if-else) si repetitive (loop)**

Sincronicitatea apelurilor



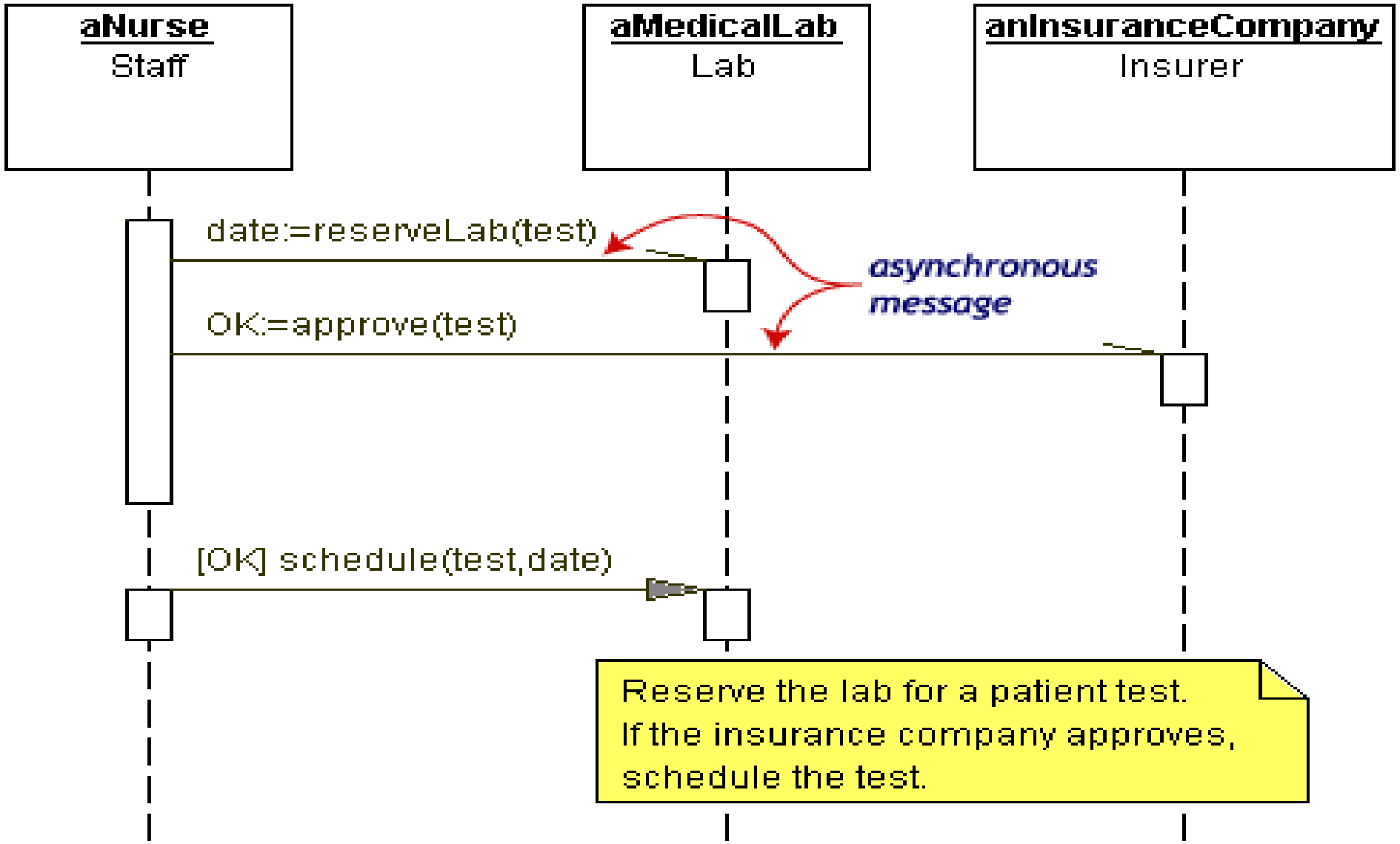


d.Secv - exemple

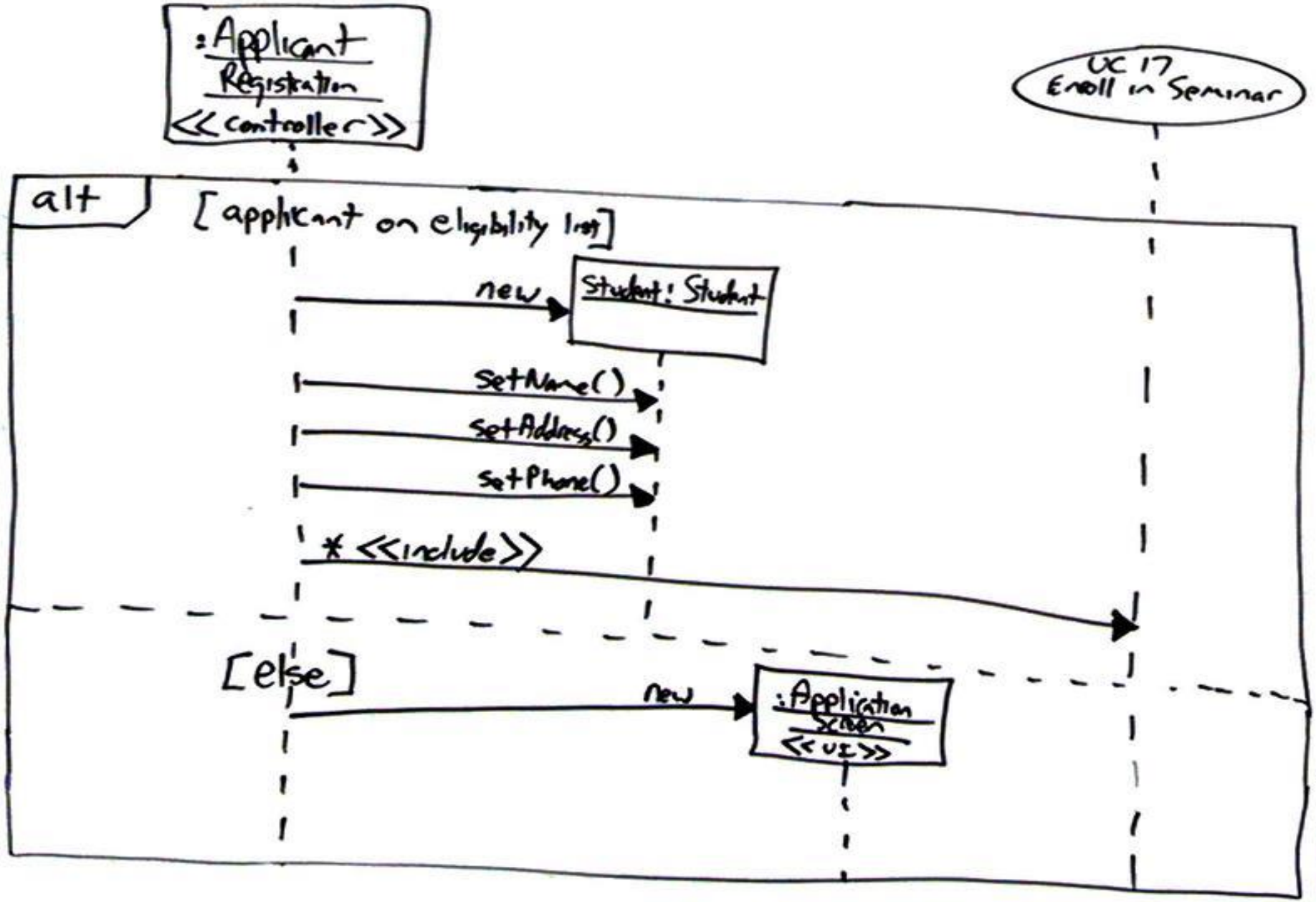


Apeluri de operatii, crearea si distrugerea obiectelor.

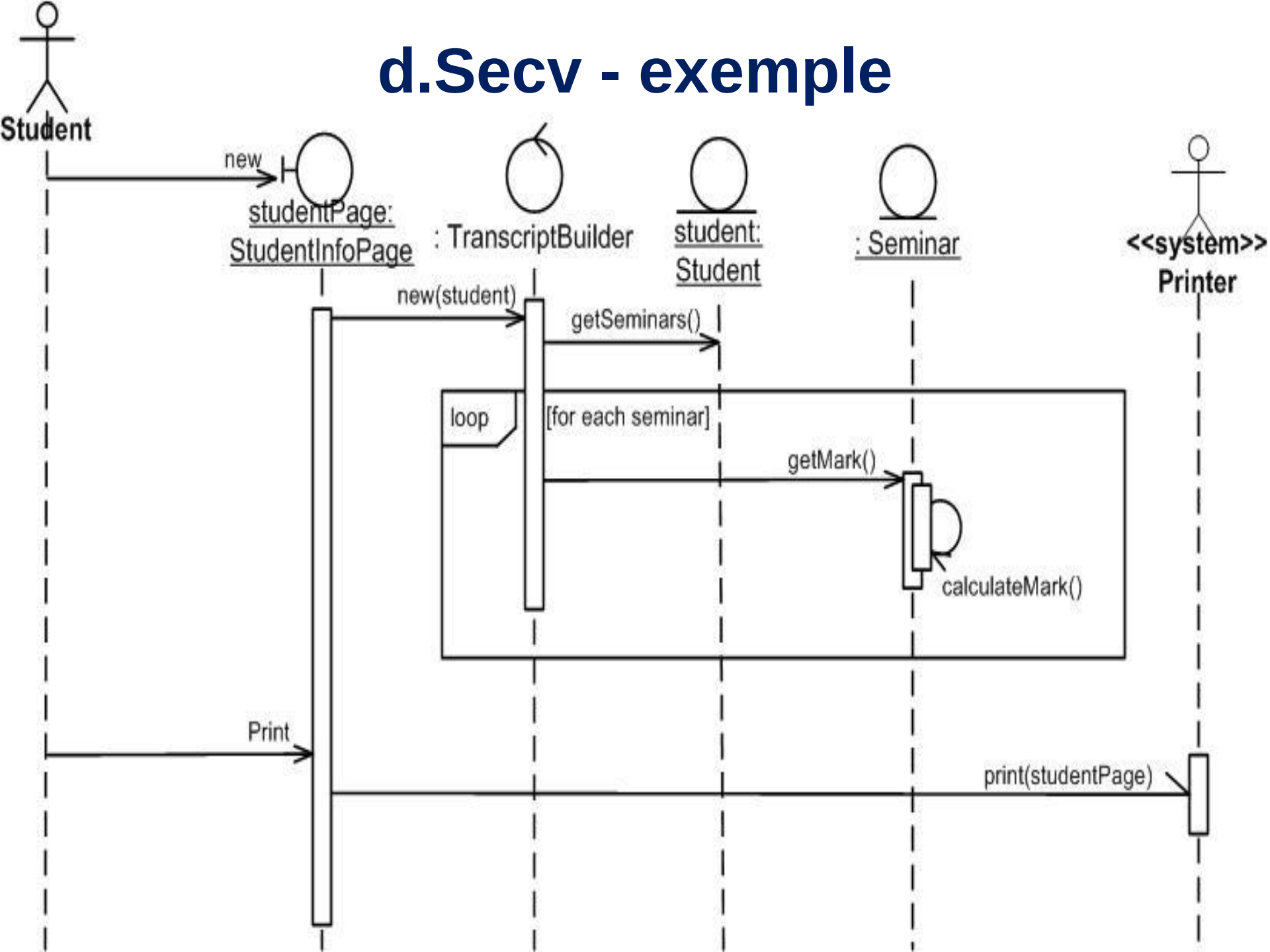
d.Secv - exemple



d.Secv - exemple



d.Secv - exemple



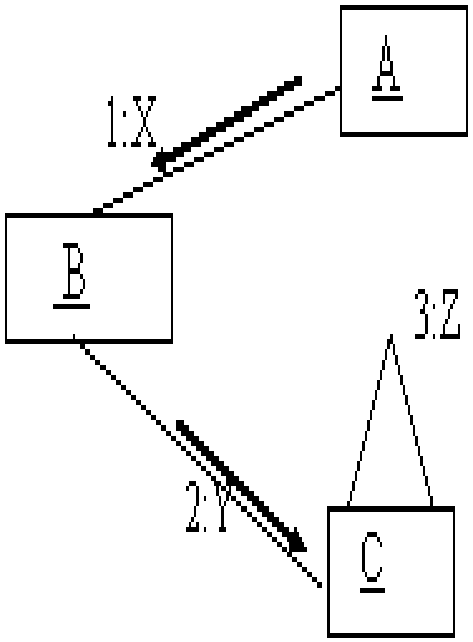
Scop diagrame de secventa

Diagramele de secventa pun accentul pe aspectul temporal:

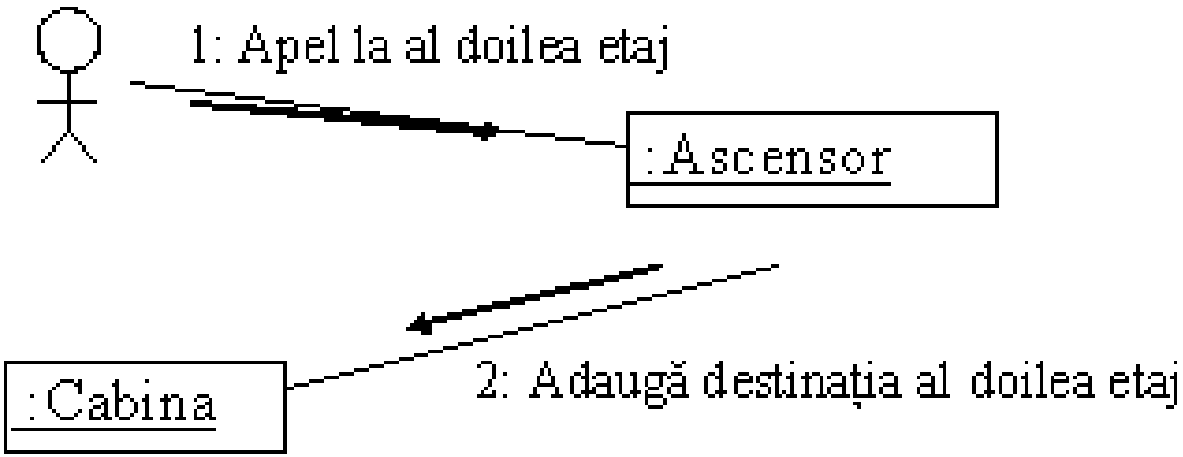
- Scurgerea timpului in timpul unei interactiuni
- Ordinea temporala a interactiunilor
- Interactiuni complexe mai mult sau mai putin paralele intre obiecte

Diagramele de colaborare

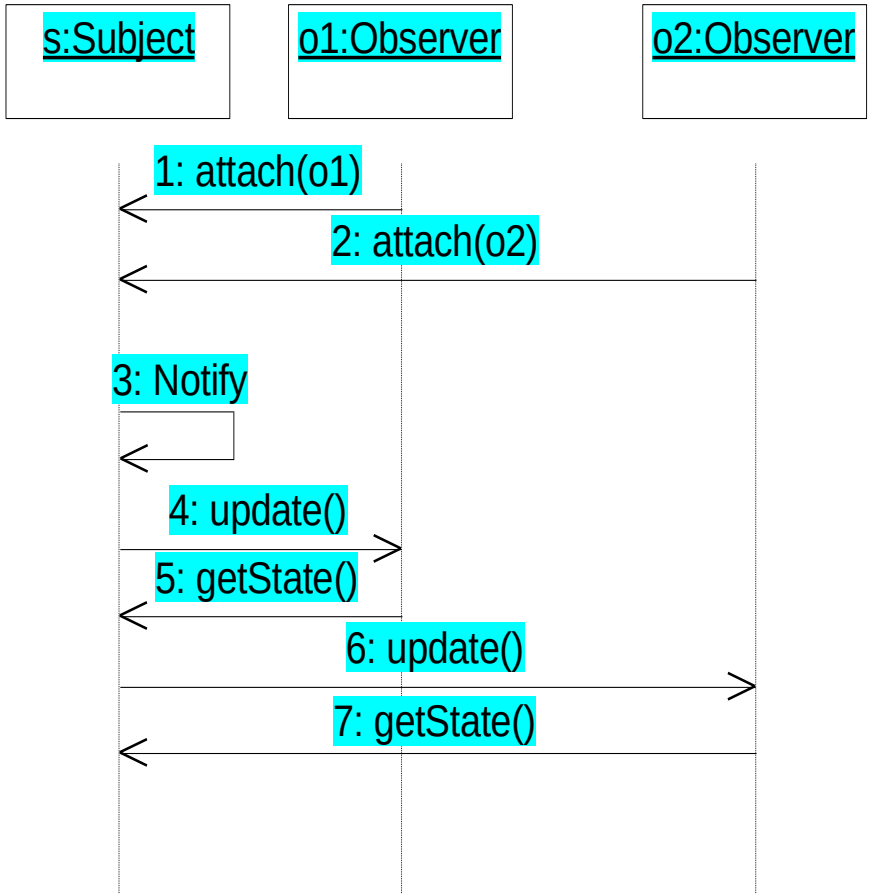
Sunt echivalente cu diagramele de secventa dar evidentiaza relatiile colaborative intre obiecte (legaturile dintre ele)

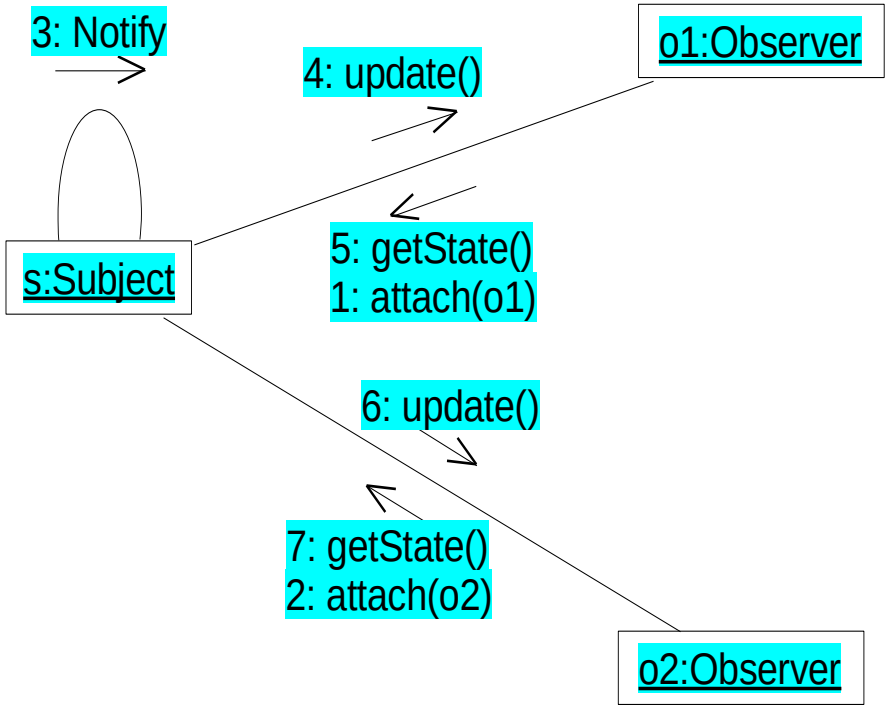


- Relatiile structurale sunt reprezentate prin "legaturi" – linii care conecteaza obiectele.
- Mesajele schimbate între obiecte sunt reprezentate de-a lungul legăturilor.
- Ordinea de trimitere a diferitelor mesaje este indicată printr-un număr amplasat in fata mesajului, ca în figura urmatoare:


1: Apel la al doilea etaj
:Ascensor
:Cabina
2: Adaugă destinația al doilea etaj

Echivalenta d.Secv – d.Colaborare





Diagrame de interactiune in UML 2

Diagrame de secventa

Diagrame de comunicare (corespund diagramelor de colaborare din versiunile anterioare)

Diagrame de evolutie in timp (Timing diagrams)

Diagrame de interactiune generale

- descriu fluxul controlului intr-o maniera generala
- utilizeaza notatii specifice diagramelor de activitate

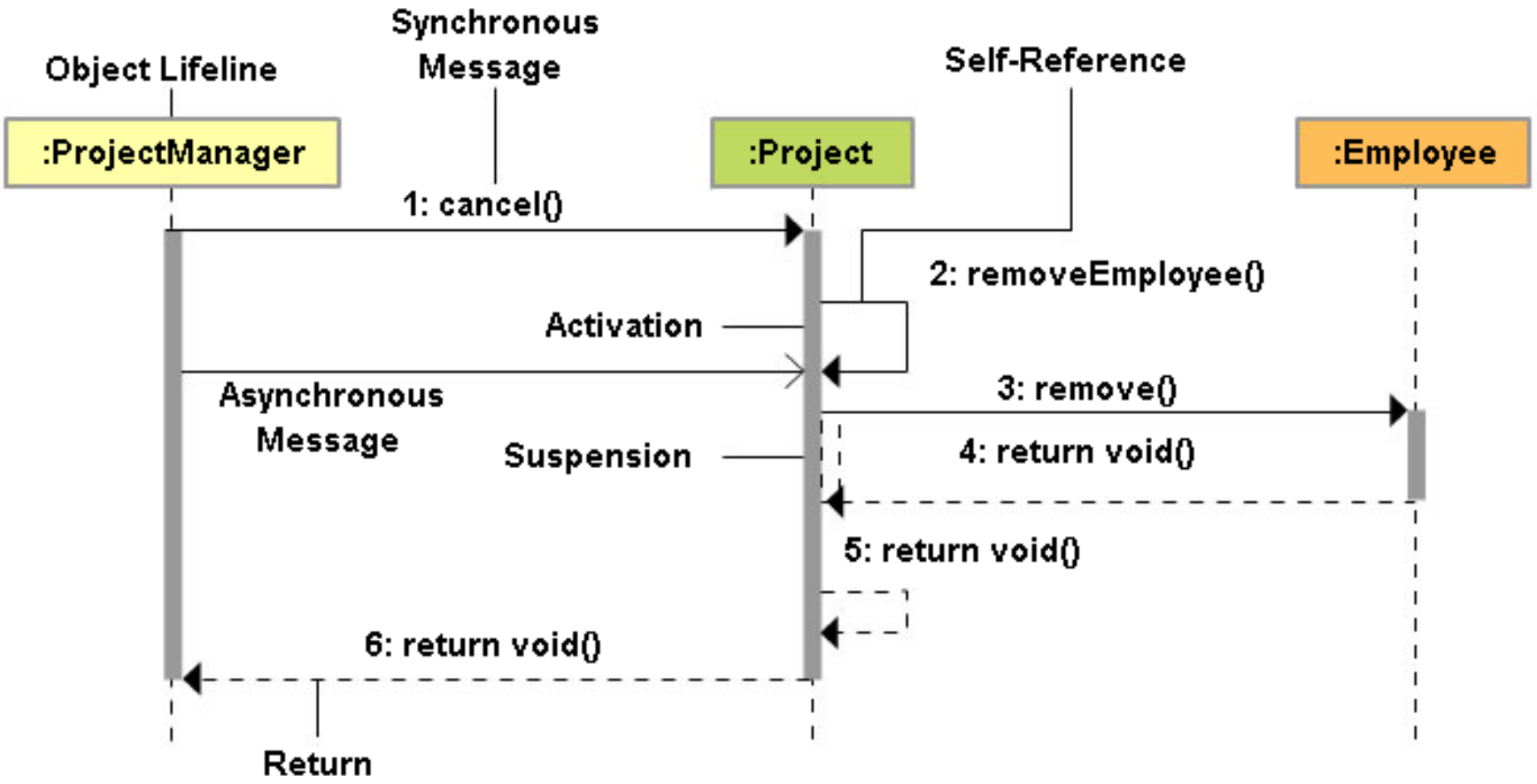


Diagrama de secventa in UML 2.

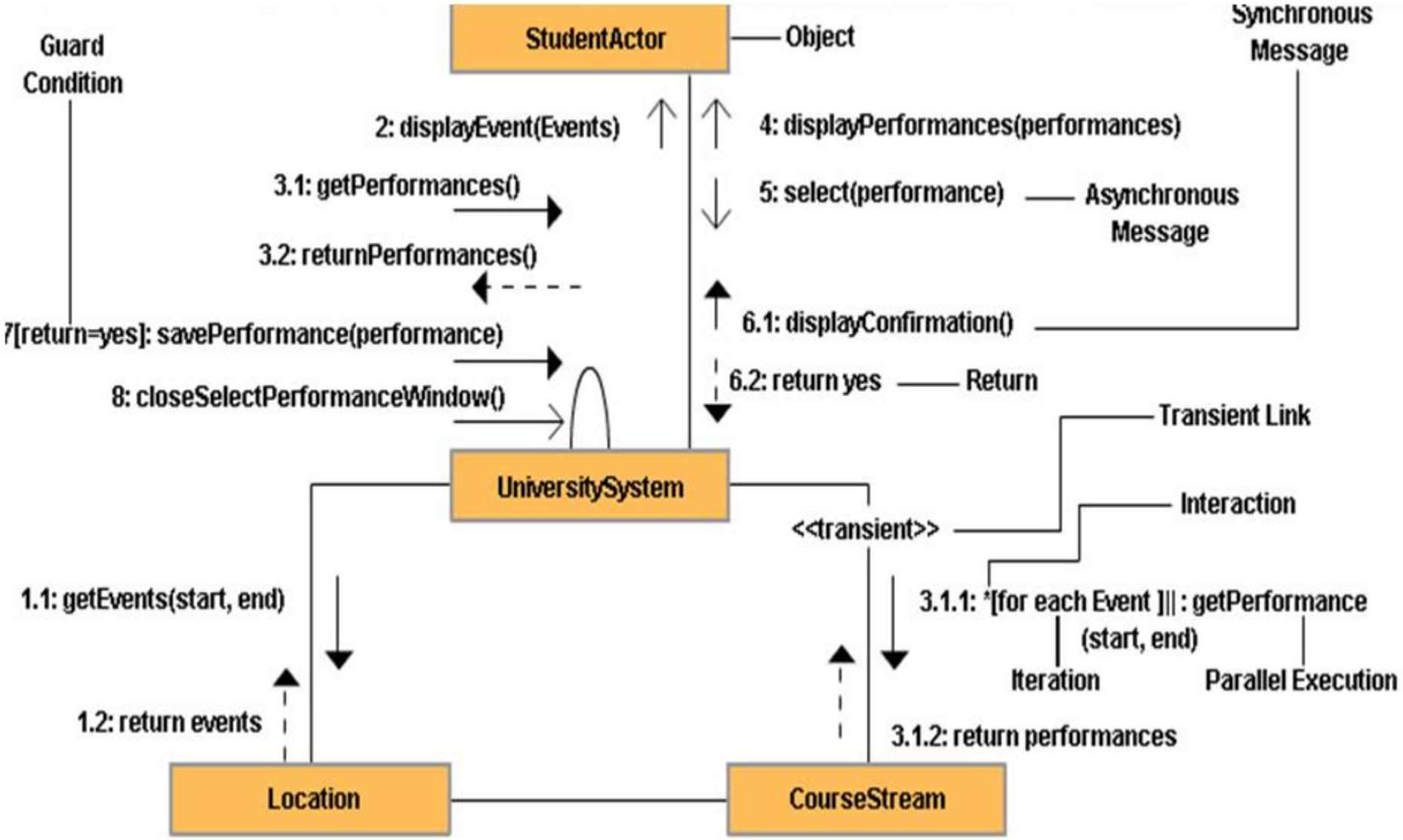


Diagrama de comunicare

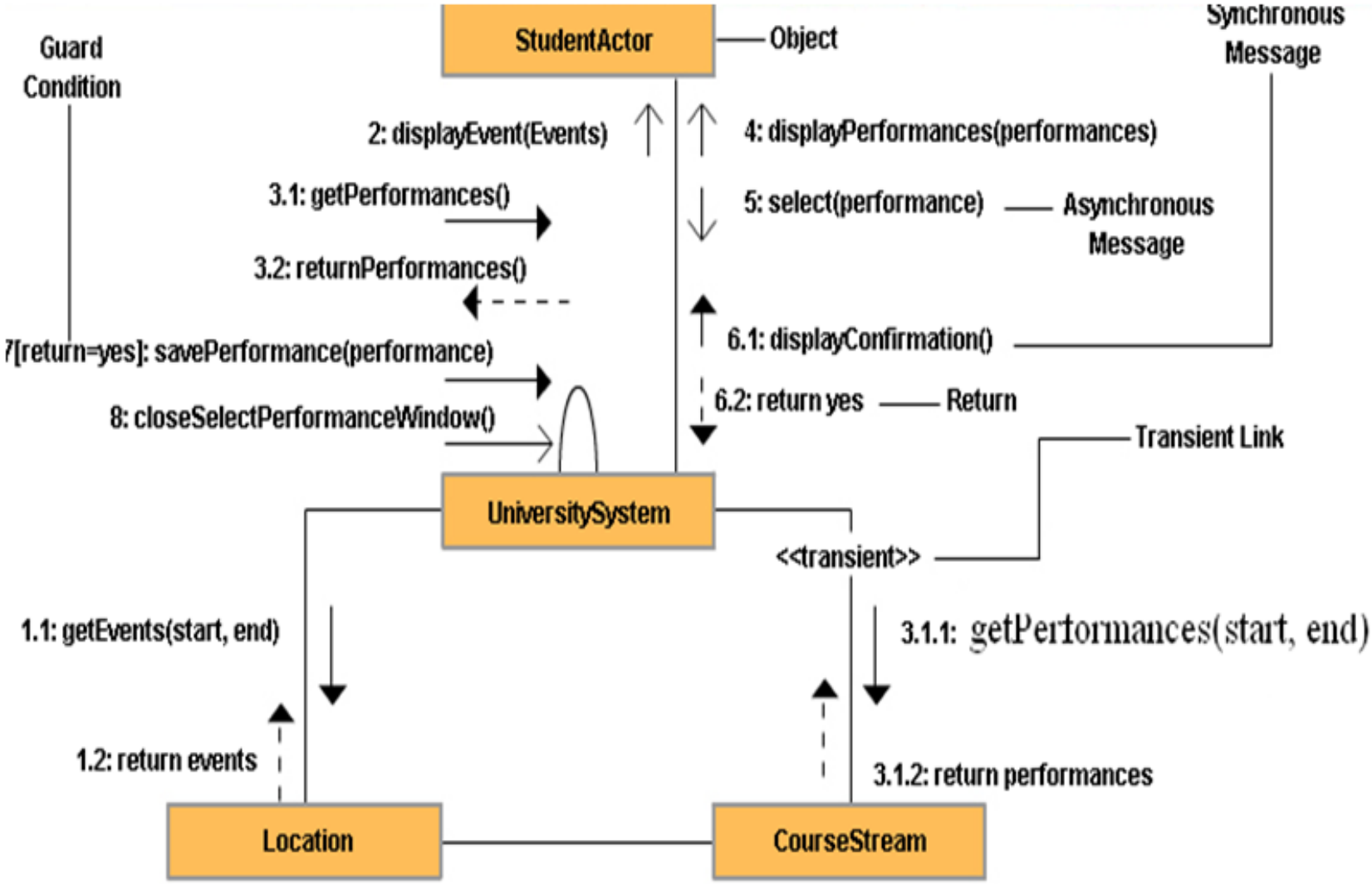


Diagrama de comunicare care refera colaborarea “getPerformances”.

Lecturi suplimentare

- http://www.agilemodeling.com/artifacts/sequenceDiagram.htm

DIAGRAME DE ACTIVITATE

Diagrame de activitate: UTILIZARI

Asemanatoare cu:

- flowcharts (organigrame)
- notatia BPMN etc

Se folosesc pentru modelarea aspectelor dinamice:

- business processes (specificarea cerintelor)
- functionarea sistemului sau a unui modul (specificarea cer.)
- Functionarea interna a modulelor (proiectare)

O diagrama de activitate poate reda:

- un flux de lucru
- pasii unui proces de calcul
- fluxul controlului intr-o operatie
- executia secventiala sau paralela a unor actiuni.

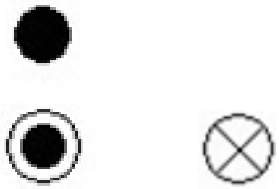
Diagrame de activitate: ELEMENTE DE BAZA

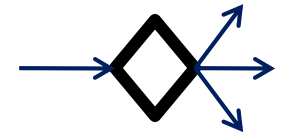
- O diagramă de activitate redă o activitate (care poate sa apara ca o stare intr-o diagrama de stari) descompusă în acţiuni care se pot executa secvenţial sau în paralel.
- Se reprezintă printr-un graf orientat:
  - nodurile corespund acţiunilor: pasi singulari din activitate
  - tranzițiile indică ordinea executiei.
- Actiunile redate intr-o diagrama de activitate pot fi executate de obiecte diferite, care sunt active in acelasi timp. Astfel, o diagrama de activitate poate reda, la un nivel de detaliu mai ridicat, interactiunea dintre obiecte reprezentata printr-o diagrama de secventa.

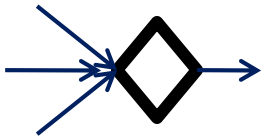
Diagrame de activitate: TIPURI DE NODURI (1)

- Noduri executabile:
  - noduri actiune
  - noduri „tratare exceptie”

- Noduri de control
  - initial
  - final (final activitate, final flux)
  - decizie
  - Merge (unificare ramuri decizie)
  - fork (paralelizare)
  - Join (unificare ramuri paralele)

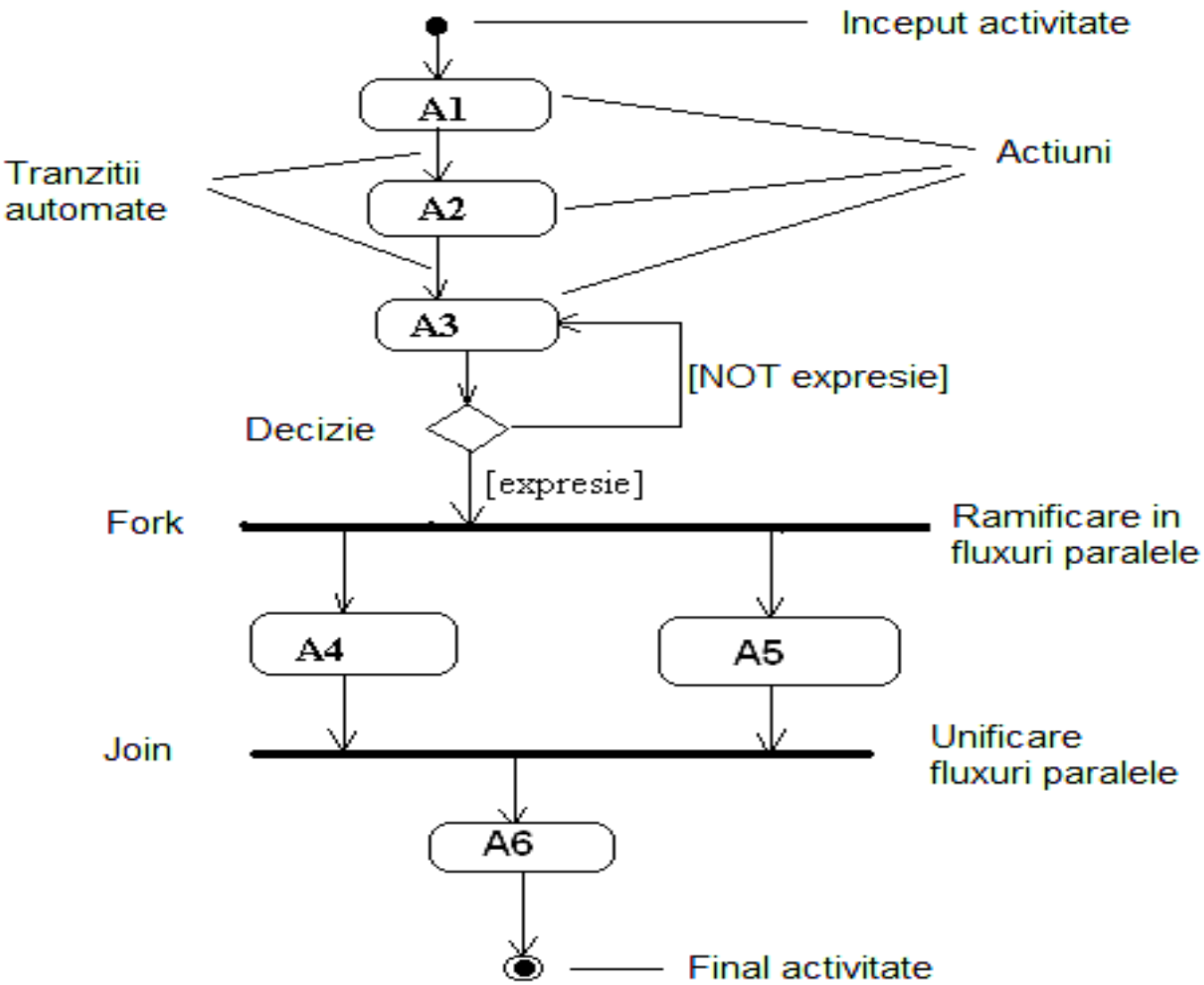




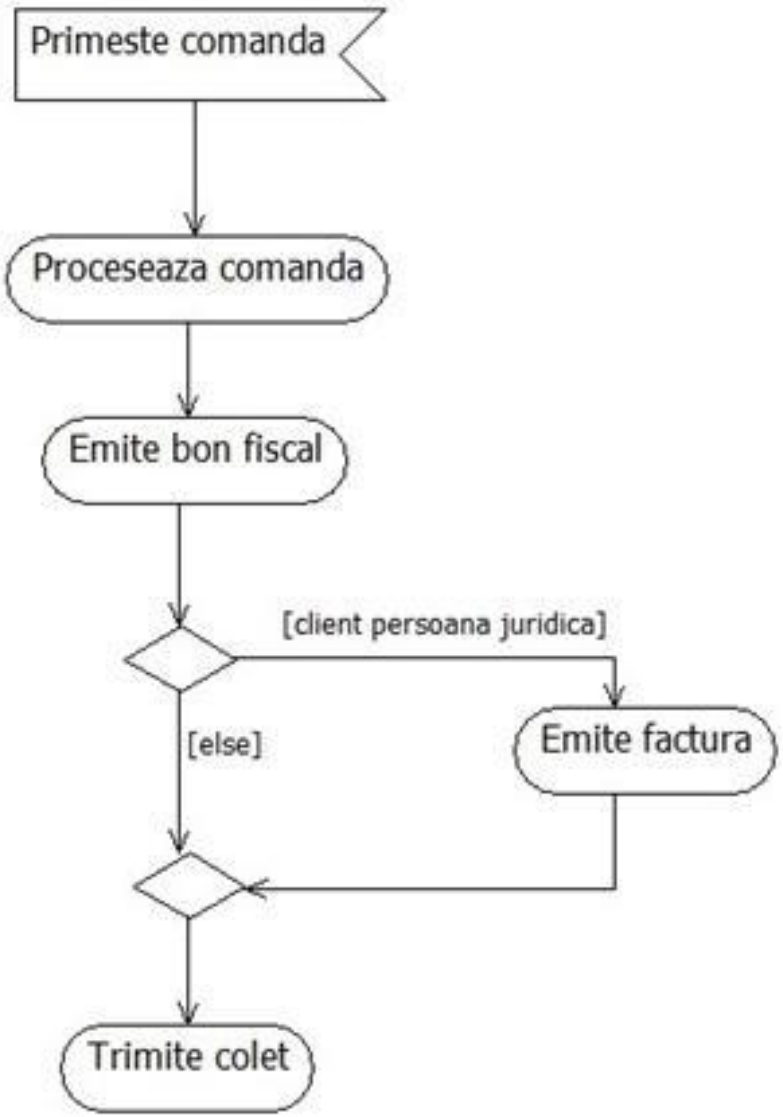


- Noduri obiect

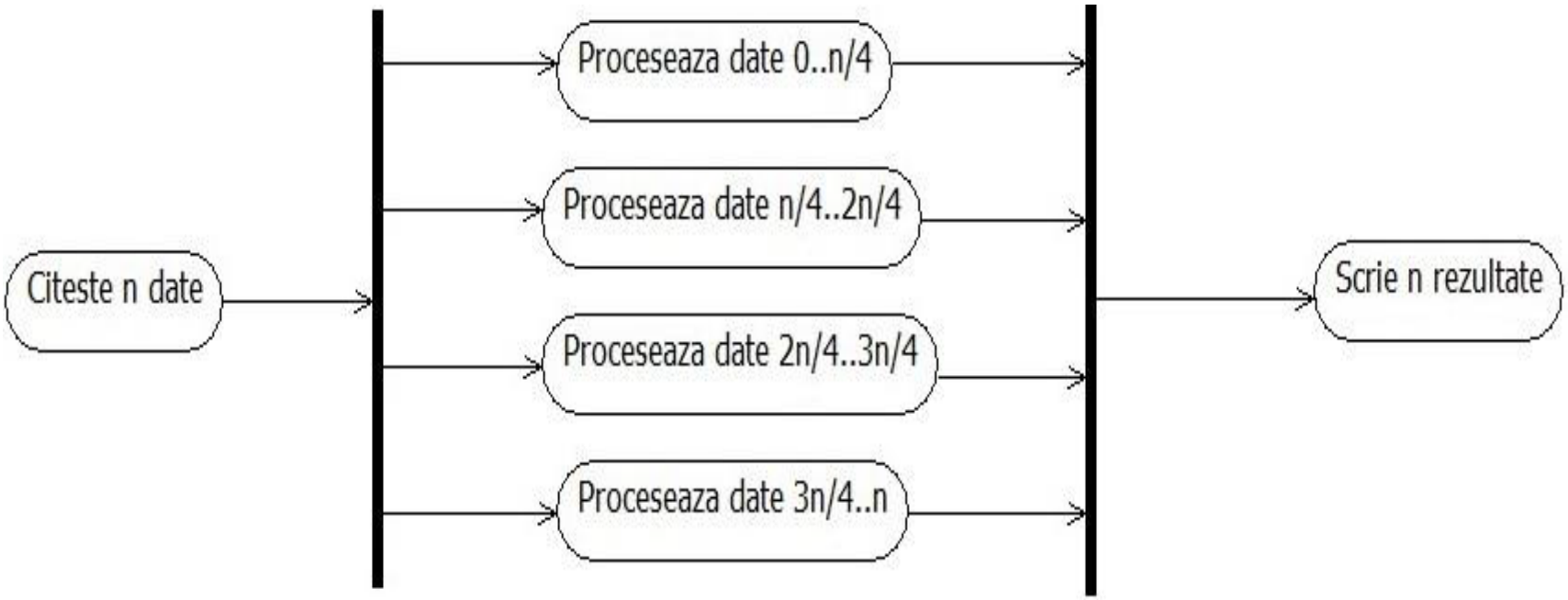
Diagrame de activitate: EXEMPLE



Diagrame de activitate: EXEMPLE



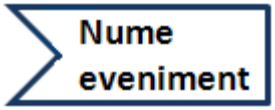
Diagrame de activitate: EXEMPLE



Diagrame de activitate: NODURI EVENIMENT

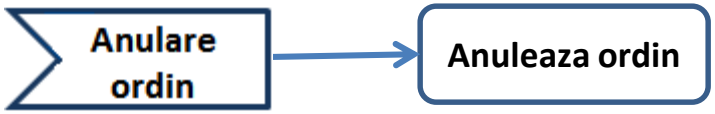
Alte tipuri de noduri actiune:

Actiune care genereaza un eveniment



Actiune care se executa atunci cand se primeste un eveniment

Exemple:

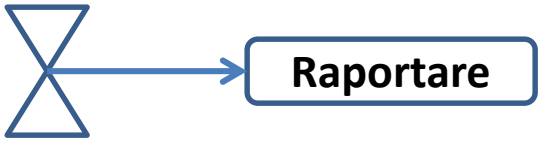


La primirea evenimentului “Anulare ordin”, este invocata operatia “Anuleaza ordin”.

La terminarea actiunii “Prelucreaza ordin” se genereaza evenimentul “Cere plata”, care declanseaza actiunea “Confirmare plata”.

Diagrame de activitate : NODURI TIMP / DURATA

Actiune lansata in urma unui eveniment specificat prin moment de timp sau durata



Sfarsit de luna

La fiecare sfarsit de luna se genereaza o iesire care este intrare pentru actiunea Raportare.

Diagrame de activitate : NODURI OBIECT

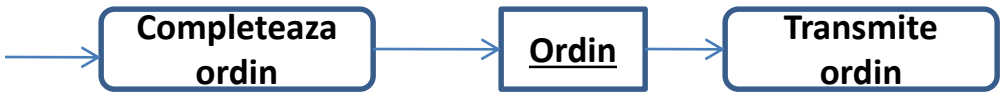
flux de obiecte = tranziție prin care se transmit date

Notatie alternativa (echivalenta)

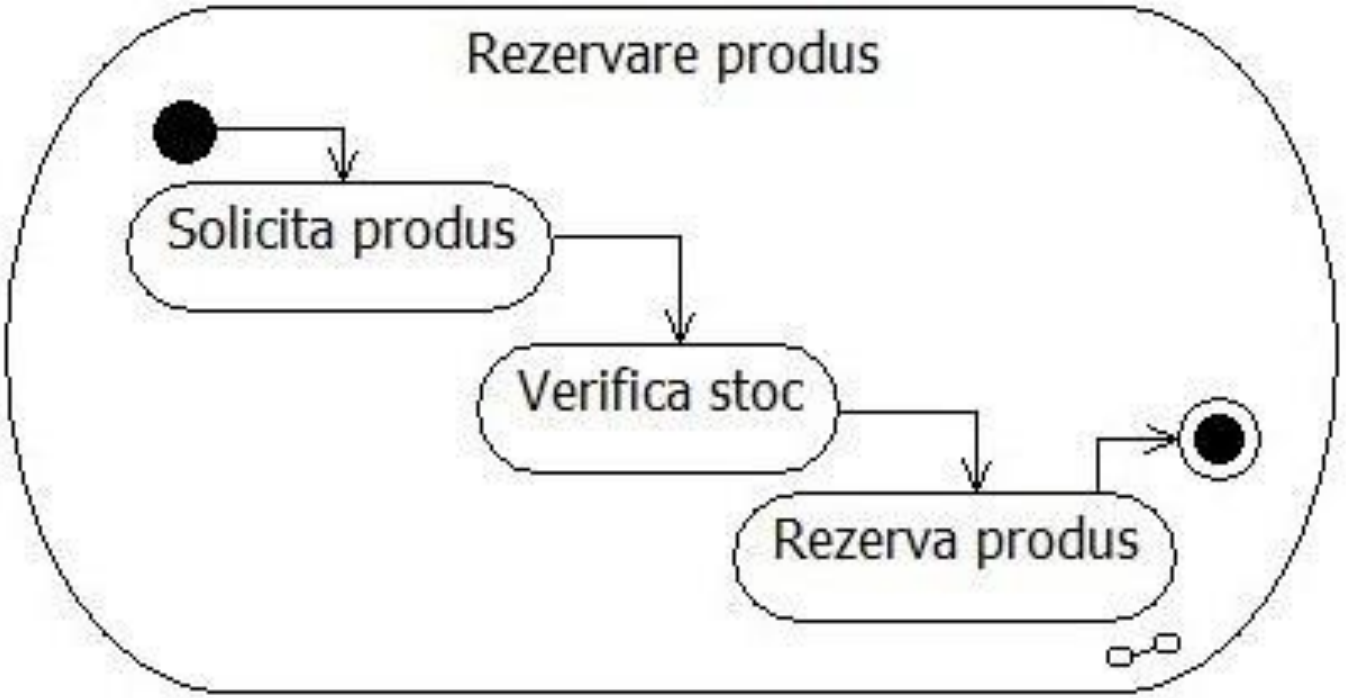
"Pini" de iesire/intrare actiune.

Notatie pentru o baza de date

DIAGRAME DE ACTIVITATE : EXEMPLE



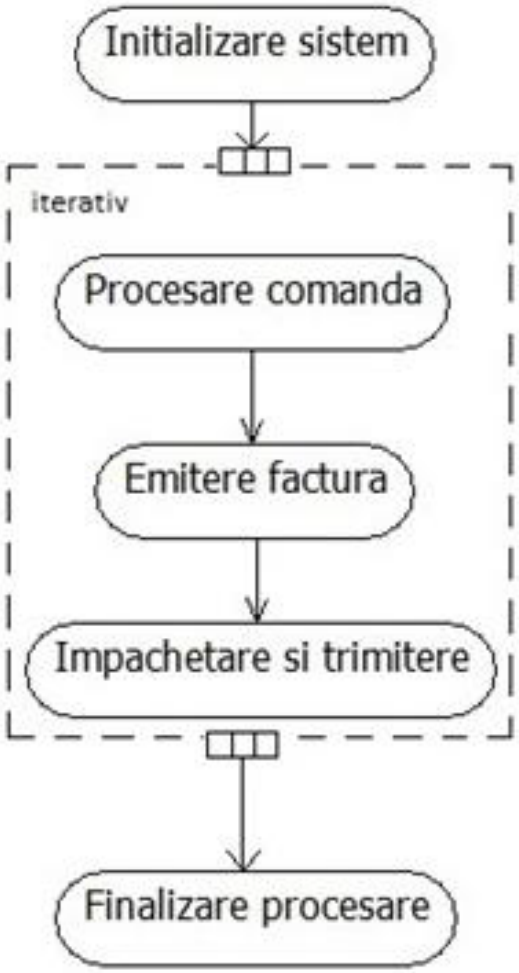
Diagrame de activitate: ACTIVITATI COMPUSE



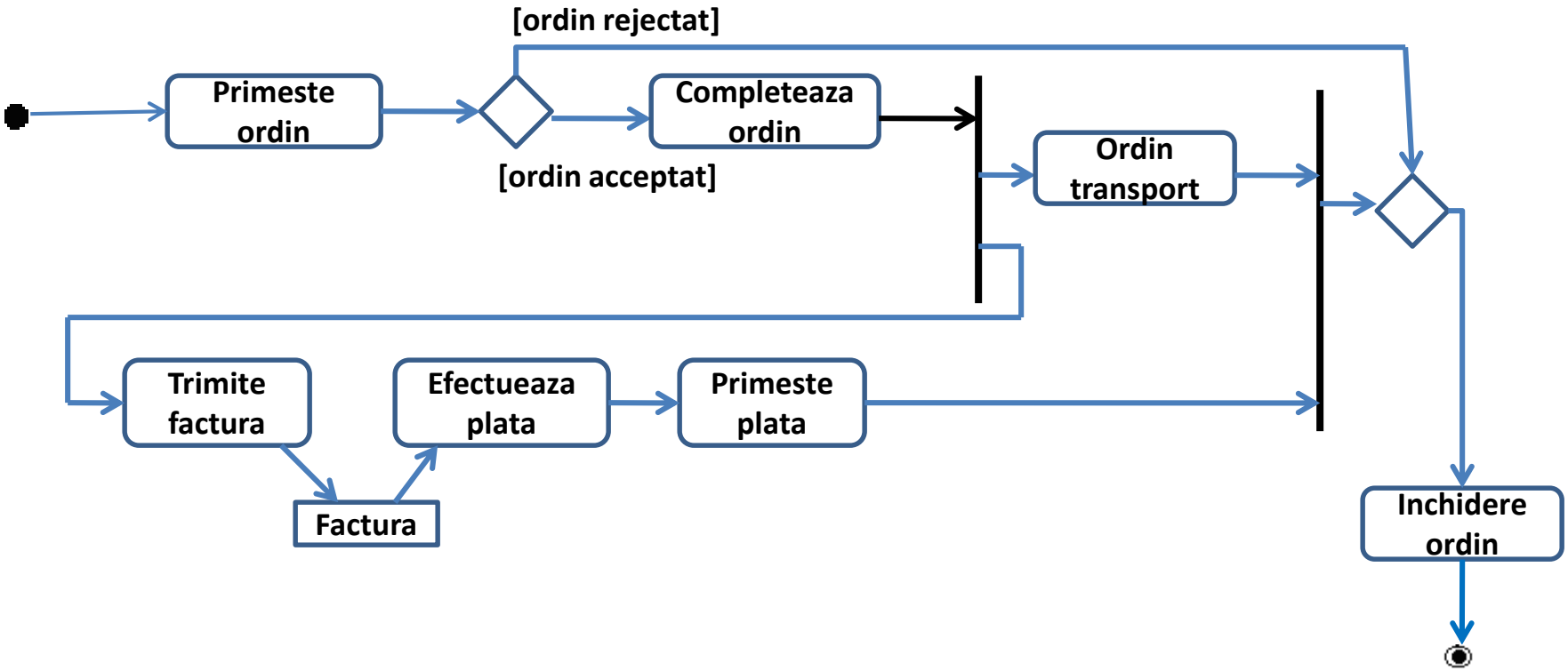
Diagrame de activitate: Prelucrari pe seturi (Regiuni de expansiune)

- O regiune de expansiune este o regiune strict imbricată într-o diagrama de activitate, cu intrări și ieșiri bine stabilite
- Fiecare intrare este o colecție de valori
- Regiunea de expansiune se execută pentru fiecare element al colecției de valori de intrare.
  - *Iterativ*
  - *Paralel*
  - *Stream*
- Regiunea se reprezintă prin încadrarea activităților repetate într-un dreptunghi punctat.
- Intrarile si iesirile se reprezinta prin noduri de expansiune

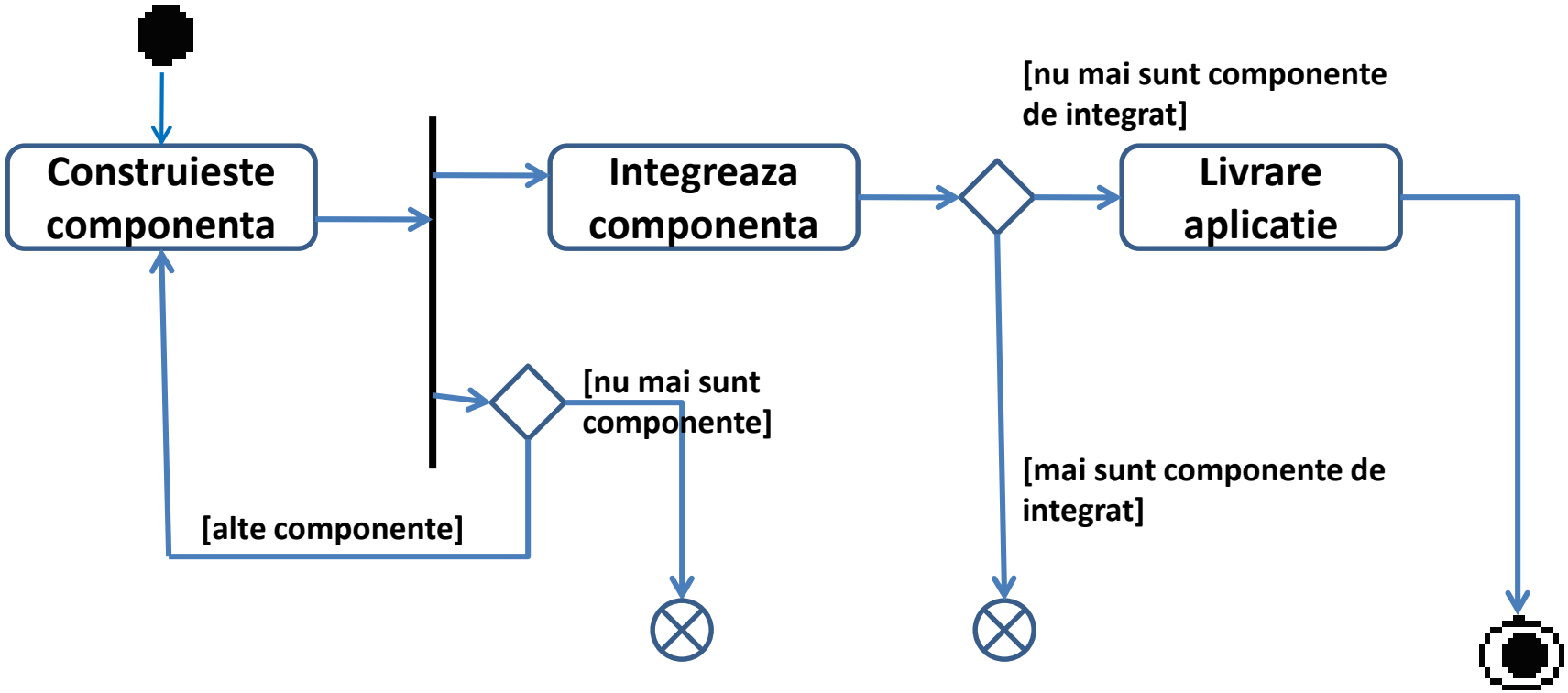
Diagrame de activitate: Prelucrari pe seturi (Regiuni de expansiune)

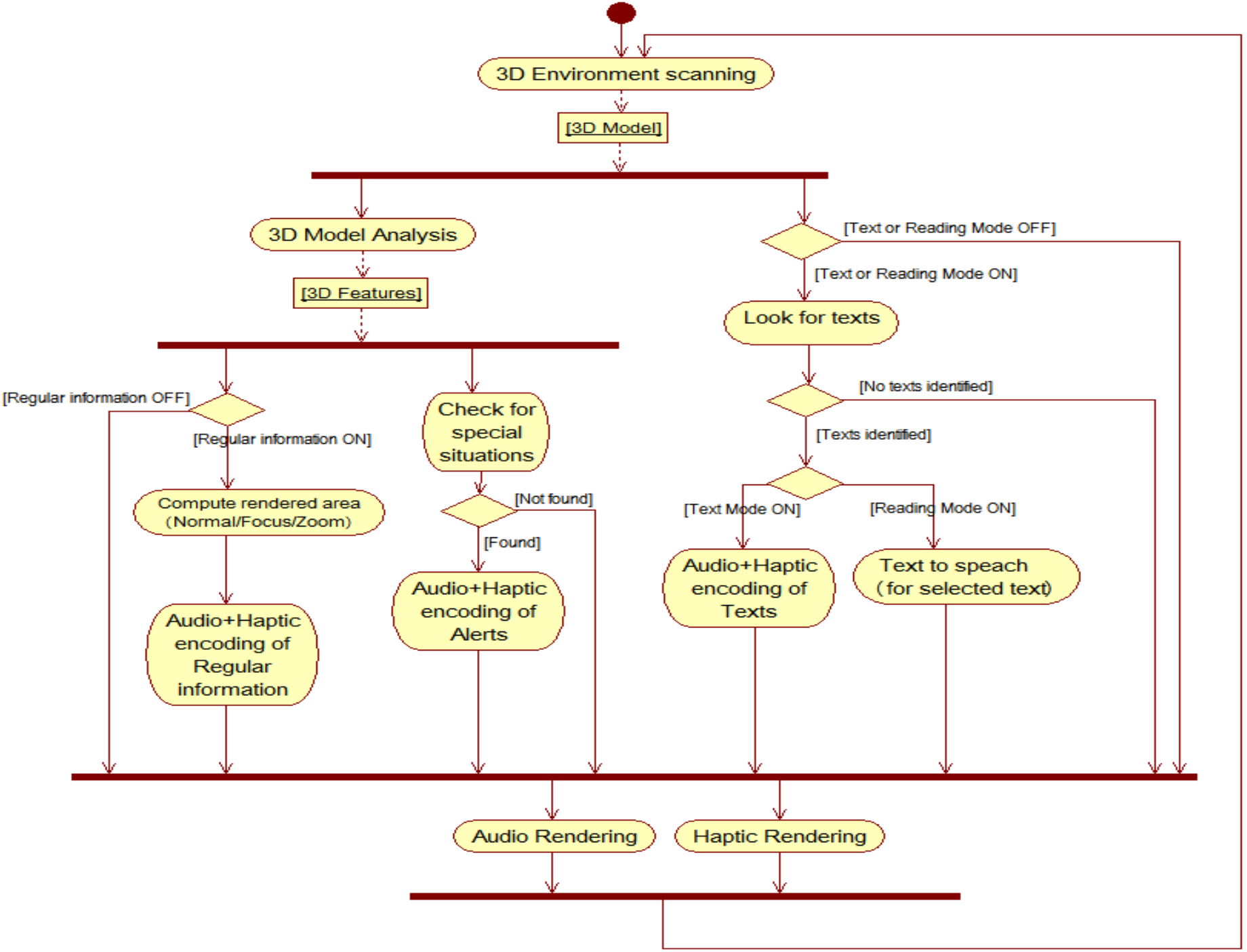


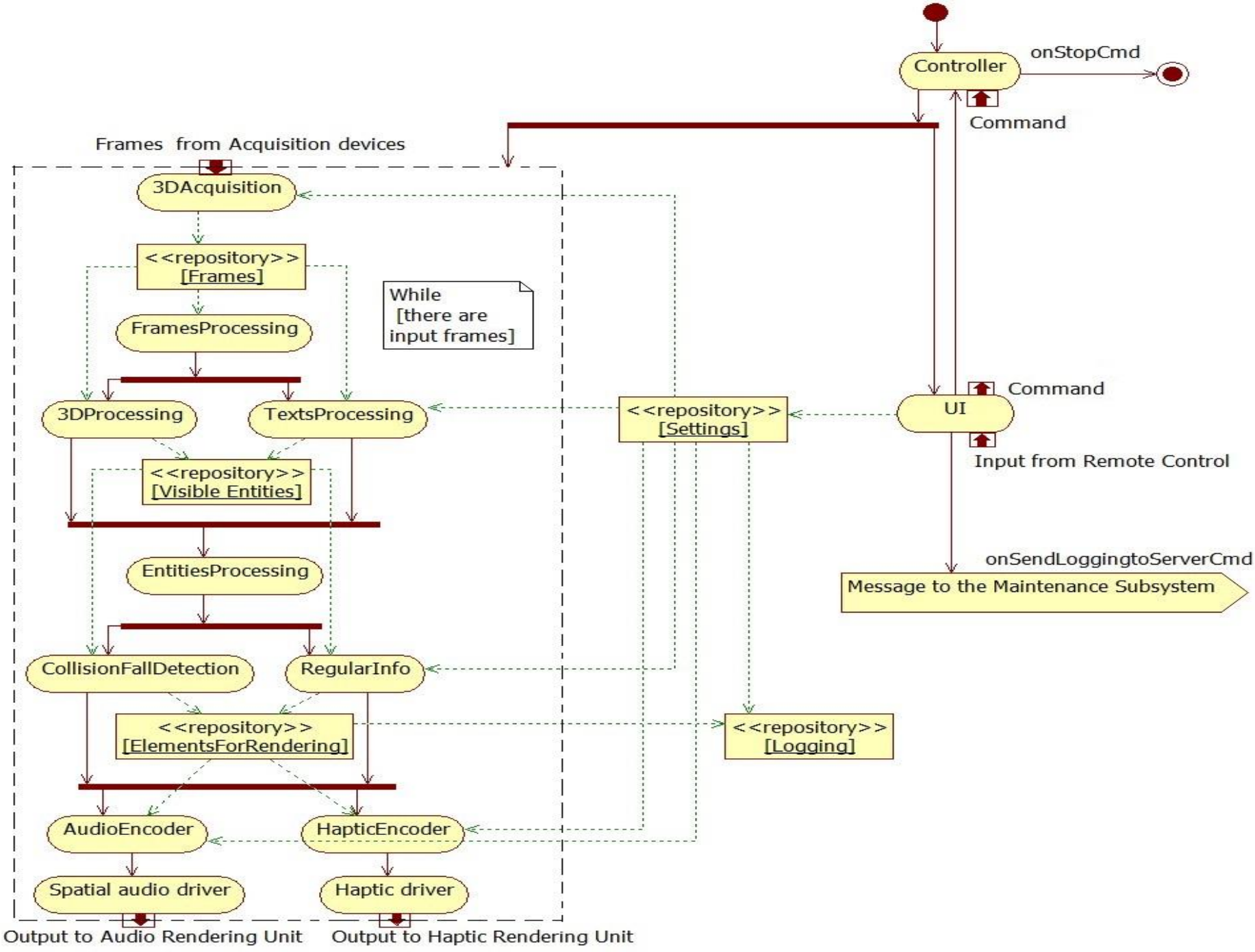
Diagrame de activitate: EXEMPLE



Diagrame de activitate: EXEMPLE




3D Environment scanning
[3D Model]
3D Model Analysis
[3D Features]
[Regular information OFF]
[Regular information ON]
Compute rendered area (Normal/Focus/Zoom)
Audio+Haptic encoding of Regular information
Check for special situations
[Not found]
[Found]
Audio+Haptic encoding of Alerts
[Text or Reading Mode OFF]
[Text or Reading Mode ON]
Look for texts
[No texts identified]
[Texts identified]
[Text Mode ON]
[Reading Mode ON]
Audio+Haptic encoding of Texts
Text to speach (for selected text)
Audio Rendering
Haptic Rendering


Controller
onStopCmd
Command
Frames from Acquisition devices
3DAcquisition
<<repository>>
[Frames]
While
[there are
input frames]
FramesProcessing
3DProcessing
TextsProcessing
<<repository>>
[Visible Entities]
EntitiesProcessing
CollisionFallDetection
RegularInfo
<<repository>>
[ElementsForRendering]
AudioEncoder
HapticEncoder
Spatial audio driver
Haptic driver
Output to Audio Rendering Unit
Output to Haptic Rendering Unit
<<repository>>
[Settings]
<<repository>>
[Logging]
UI
Command
Input from Remote Control
onSendLoggingtoServerCmd
Message to the Maintenance Subsystem

Diagrame de activitate: Partitionare

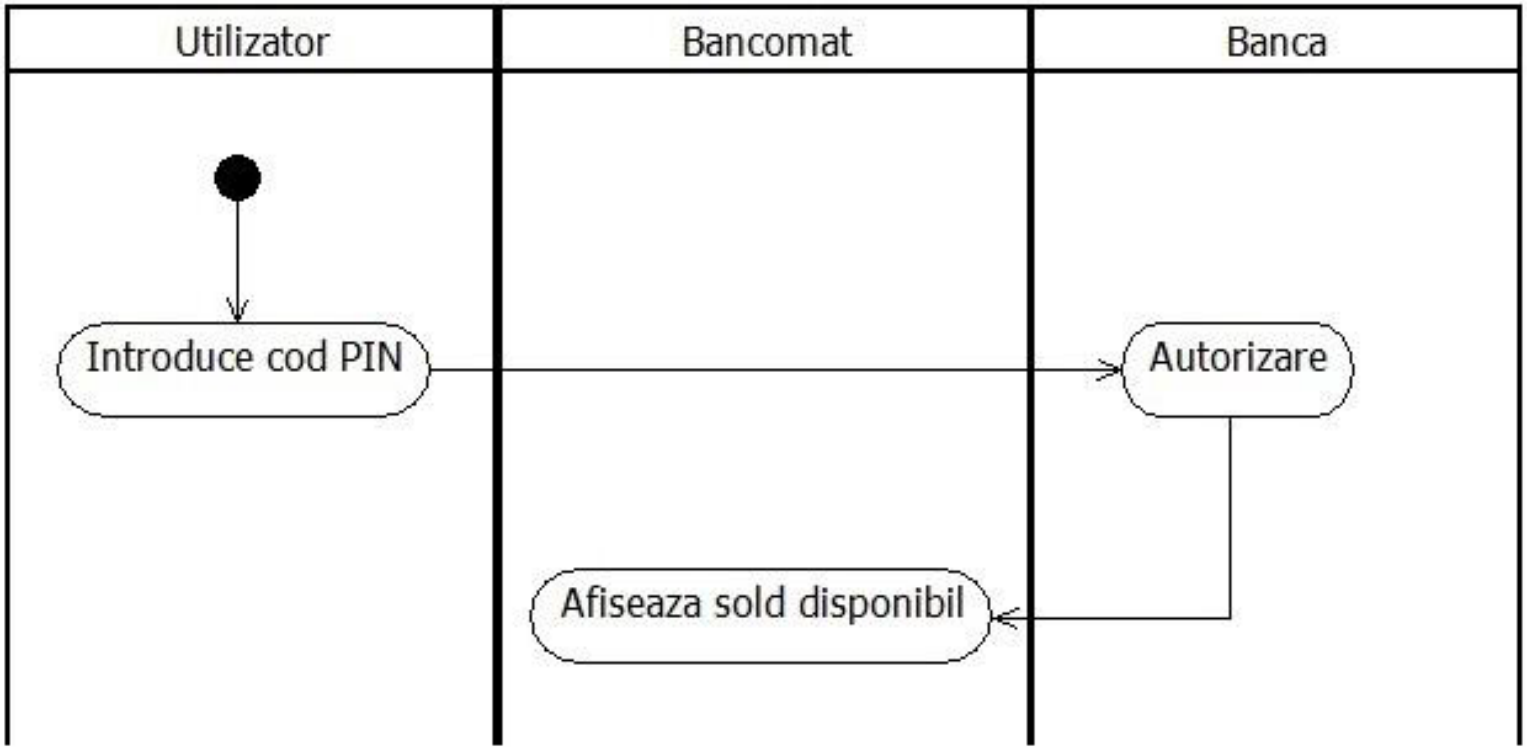
Daca:

- sunt implicate mai multe obiecte
- Se doreste evidentierea actiunilor efectuate de fiecare ob.

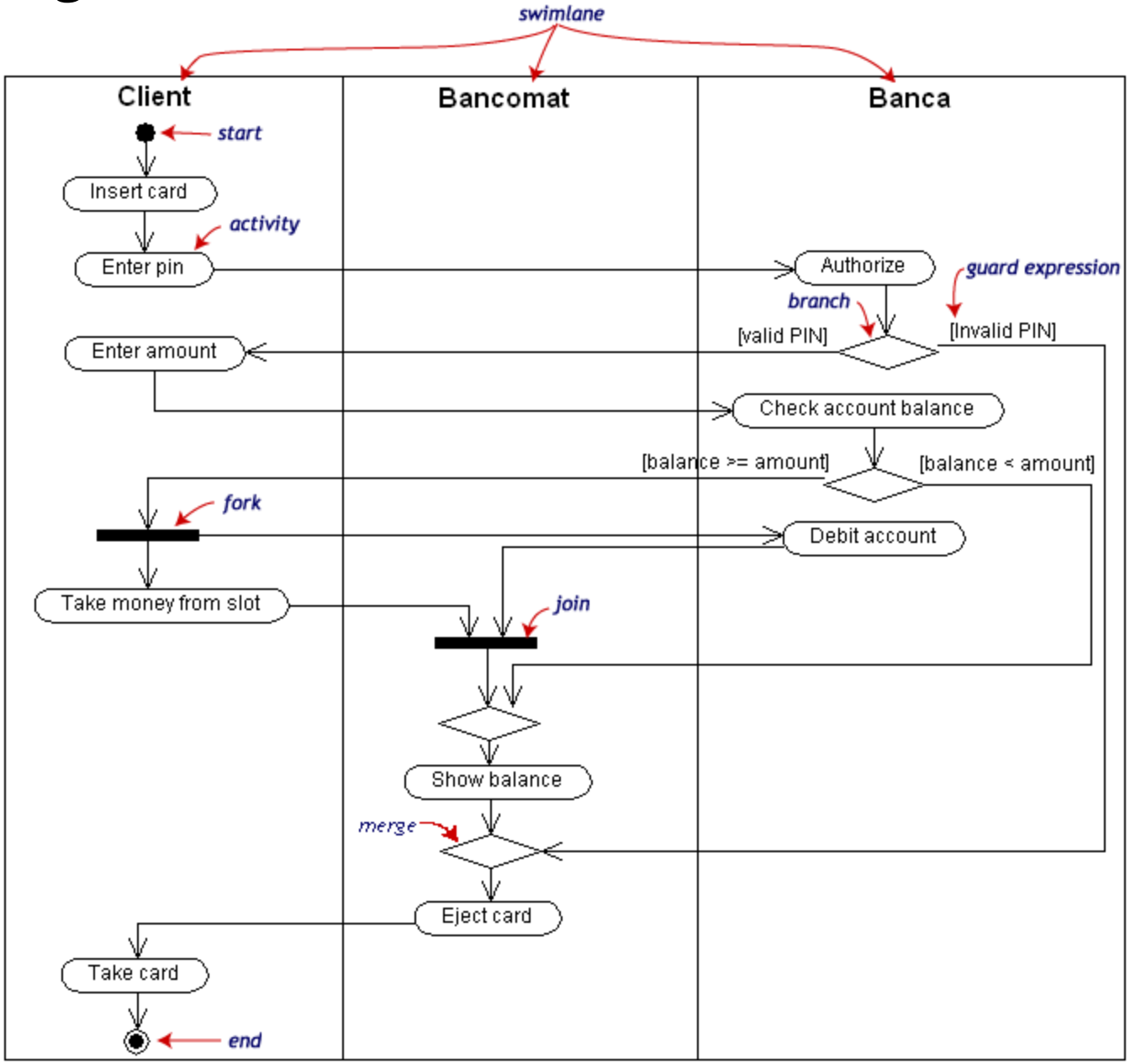
.. Actiunile pot fi aliniate pe “culoare” verticale sau orizontale distincte, corespunzatoare obiectelor care le executa:

===>>> **SWIMLANES**

Diagrame de activitate: EXEMPLE



Diagrame de activitate: EXEMPLE



Diagrame de activitate: CONCLUZII

- Diagramele de activitate pot fi utilizate pentru:
  - Modelarea proceselor din domeniul aplicatiei; la acest nivel se pune accentul pe activitati asa cum sunt ele vazute de actorii care comunica cu sistemul.
  - Modelarea scenariilor.
  - Modelarea proceselor sistemului.
  - Reprezentarea fluxului controlului într-o operatie (metodă a unei clase).
  - Modelarea, la un nivel de detaliu mai ridicat, a aspectelor dinamice ale unei societati de obiecte reprezentata printr-o diagrama de secventa sau de colaborare (schimbul de informatii intre diferite obiecte ale unei aplicatii).
- „forward engineering”: Pornind de la o diagrama de activitate poate fi generat automat cod sursa, atunci cand diagrama reprezinta o operatie
- „reverse engineering” Este posibila generarea diagramei de activitate pornind de la cod sursa

DIAG. DE STARI *VS.* DIAG DE ACTIVITATE

- Diagramele de stare prezintă stările în care se poate afla un obiect de-a lungul vieții sale
- Diagramele de activitate prezintă ordinea în care se execută diferite acțiuni și activități, ce pot implica mai multe obiecte.

DIAG. DE SECVENȚĂ *VS.* DIAG DE ACTIVITATE

- Diagramele de secvență detaliază un caz de utilizare, evidențiind interacțiunea dintre obiecte prin mesajele ce se transmit în cadrul sistemului, punând accentul pe secvențialitatea (ordinea) mesajelor.
- Diagramele de activitate pot reda desfășurarea proceselor din sistem la un nivel de detaliu mai ridicat, prezentând activitățile efectuate de acesta, și fluxul de control dintre ele.

➢ Ambele pot decrie conditii logice de decizie si bucle, insa in cazul celor de activitate notatia este mult mai eficienta si puternica

Diagrame de stari
(State Machine diagrams)

O diagrama de stari modeleaza viata unui obiect prin:

- ❑ starile sale
- ❑ schimbarile de stare care au loc pe parcursul vietii
- ❑ evenimentele / conditiile ce determina aceste schimbarile de stare

diagrama de stari = automat finit determinist.

Cand sunt utile:

pentru obiecte ce raspund la mesaje in mod diferit pe parcursul vietii

Cand NU sunt utile:

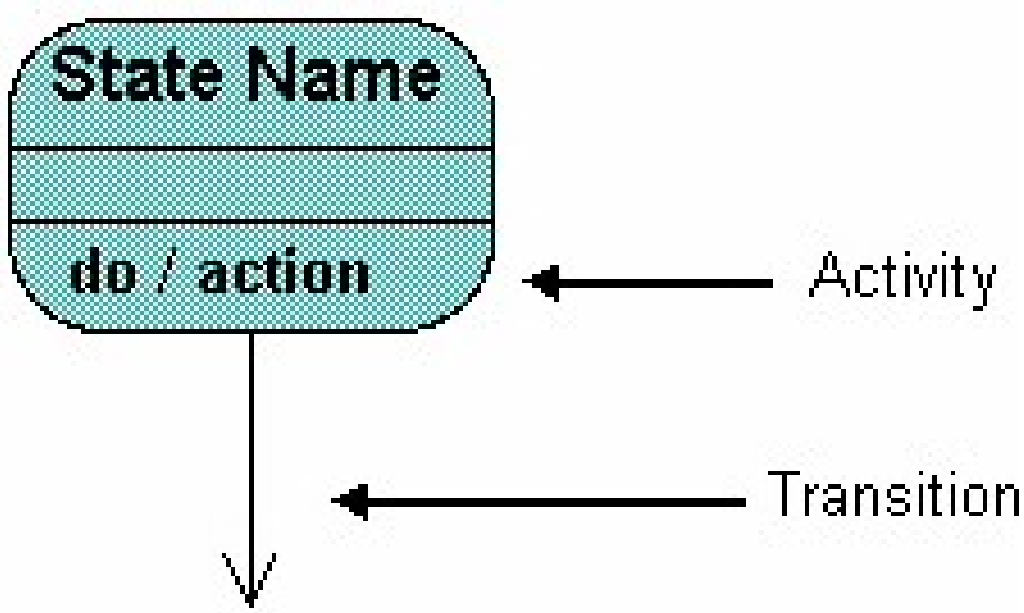
pentru obiecte ce au acelasi comportament in permanenta
(o singura stare intermediara
intre creare si distrugere)

Diferenta

- Diag. de interactiune:
  - Interactiuni intre mai multe obiecte
  - Evidentierea logicii secventiale sau a colaborarilor

- Diag. de stari: comportamentul unui singur obiect reactiv (reactioneaza la semnale), aratand transformarile de stare, actiunile si activitatile obiectului

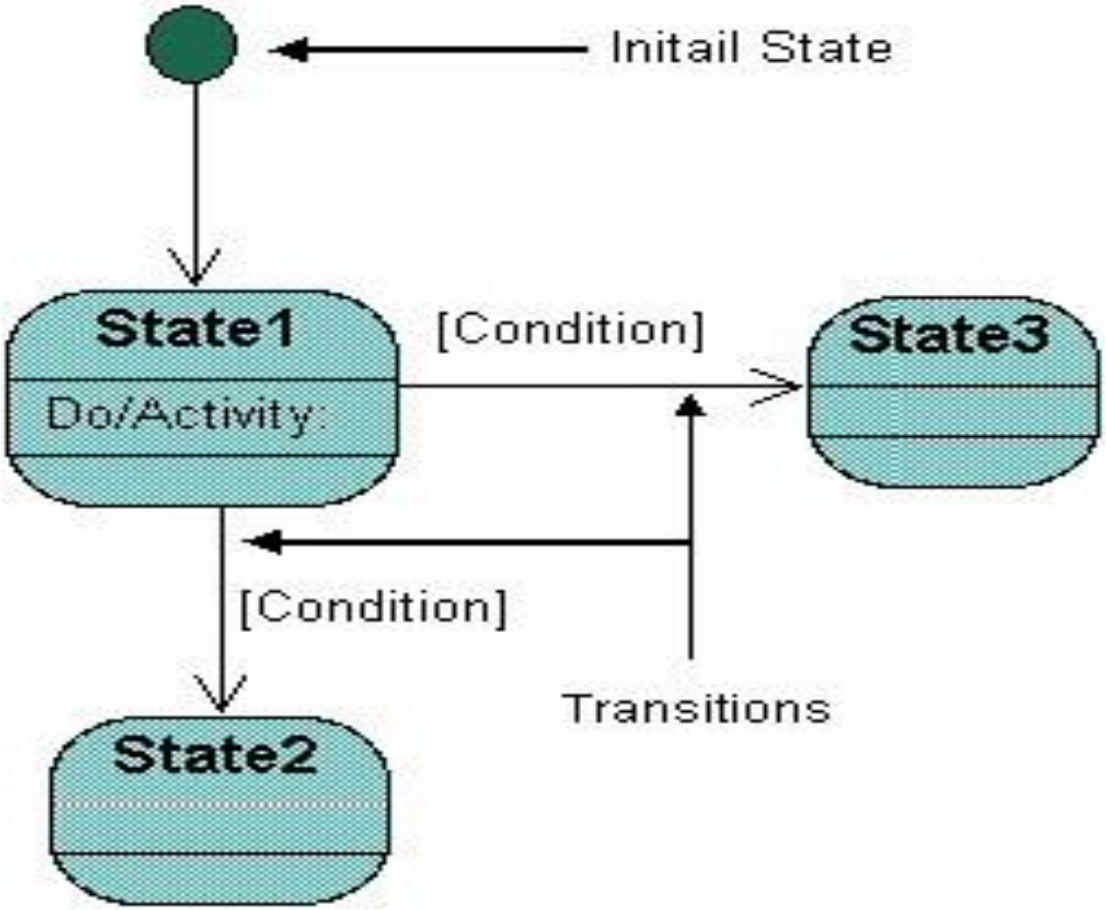
Reprezentare stare:



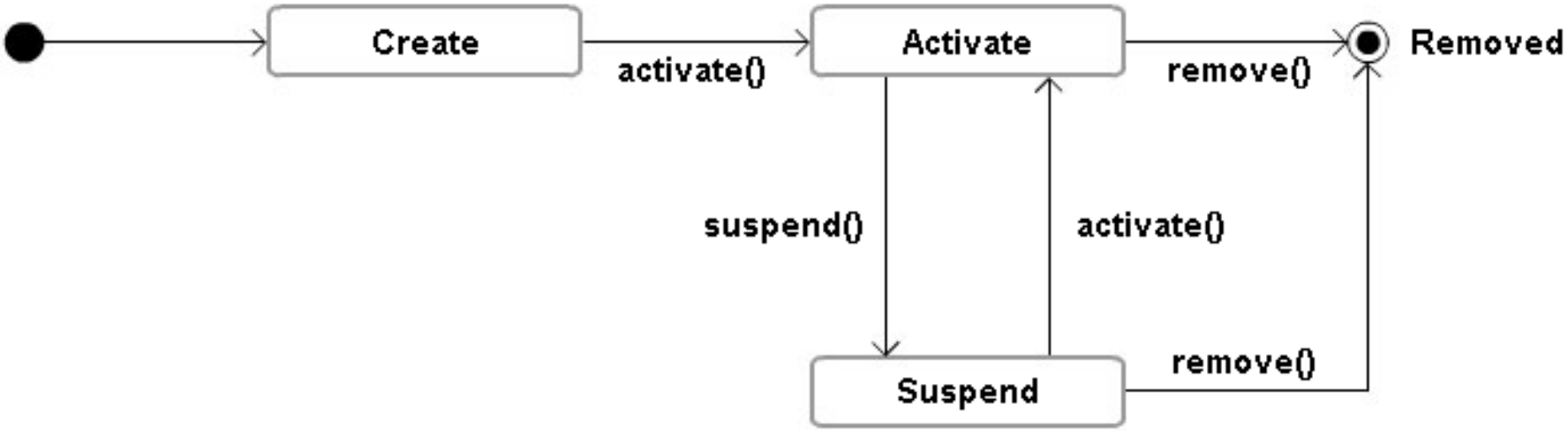
starea initiala

starea finala

Exemplu


Initail State
State1
Do/Activity:
[Condition]
State3
[Condition]
Transitions
State2

(alt) Exemplu


Create
activate()
Activate
remove()
Removed
suspend()
activate()
remove()
Suspend

Starea initiala

- Starea initiala identifica (puncteaza catre) starea in care obiectul este creat
- Notatia standard pentru stara initiala include numai cercul plin
- In practica starea initiala include si sageata care pleca din ea si starea in care obiectul este creat.

Starea finala

- La sfarsitul vietii sale (activitatii sale) obiectul atinge *starea finala* din care nu mai poate iesi
- Starea finala are toate proprietatile unei stari, cu o exceptie: nu poate avea tranzitii de iesire
- Numele starii de iesire este specificat langa simbolul grafic al starii finale.

Starea curenta

Starea curenta a unui obiect consta in:

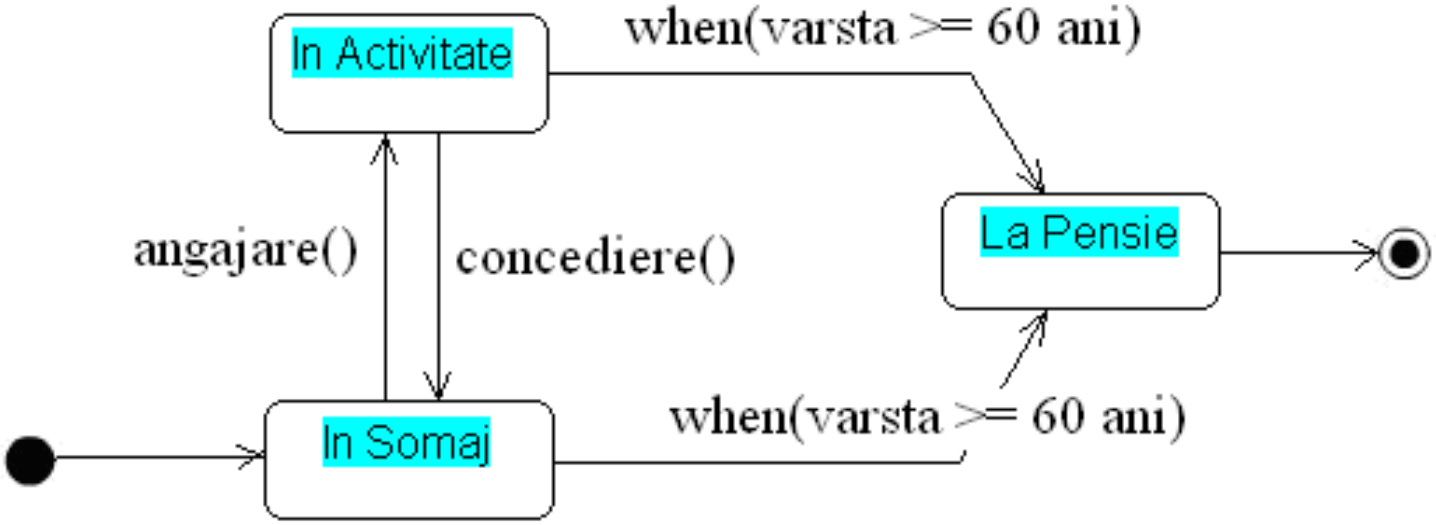
- valorile atributelor
- legaturile existente intre obiect si alte obiecte
- semantica / activitati definitorii

Evenimente

- Receptionarea unui semnal, cum ar fi invocarea unui apel, o exceptie, o notificare, un eveniment generat de interactiunea cu utilizatorul .

- Recunoasterea unei conditii in mediul extern sau in obiectul insusi:
  - conditie predefinita, care este indeplinita la un moment dat, eveniment numit *“change event”(“conditie”).*
  - trecerea unei perioade de timp desemnate, eveniment numit *“elapsed-time event”(“dupa o perioada de timp”).*

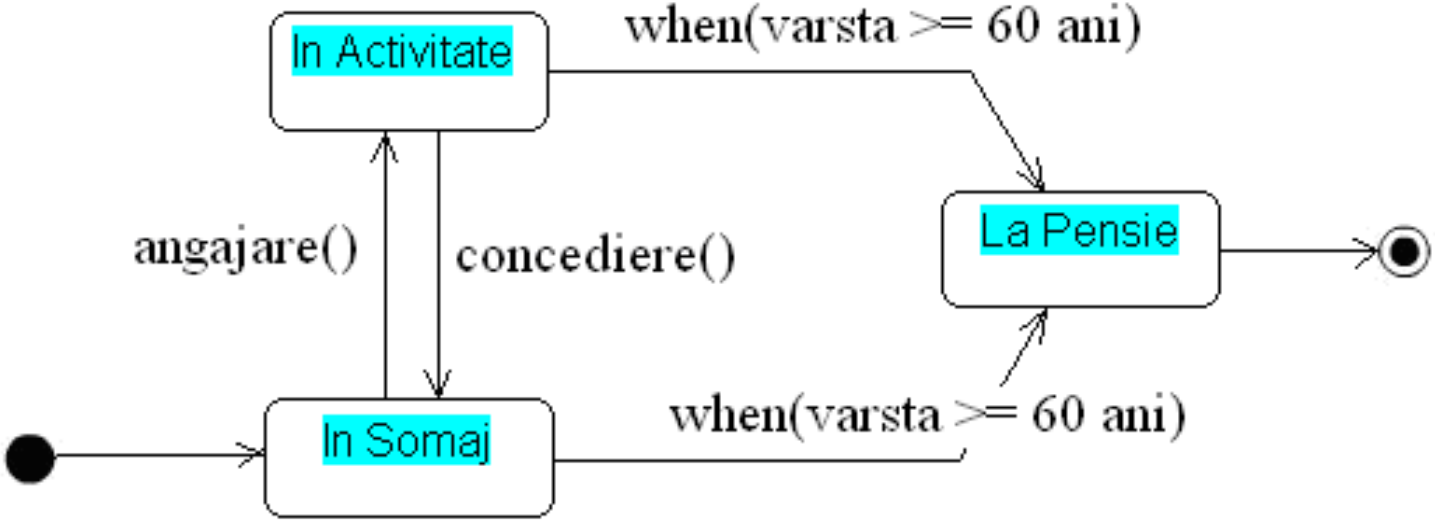
Exemplu evenimente



- angajare() si concediere() sunt evenimente externe, datorate unor mesaje trimise obiectului;
- ***when*** **(**varsta >= 60 ani**)** este un eveniment intern, determinat de valoarea curenta a unui atribut al obiectului.

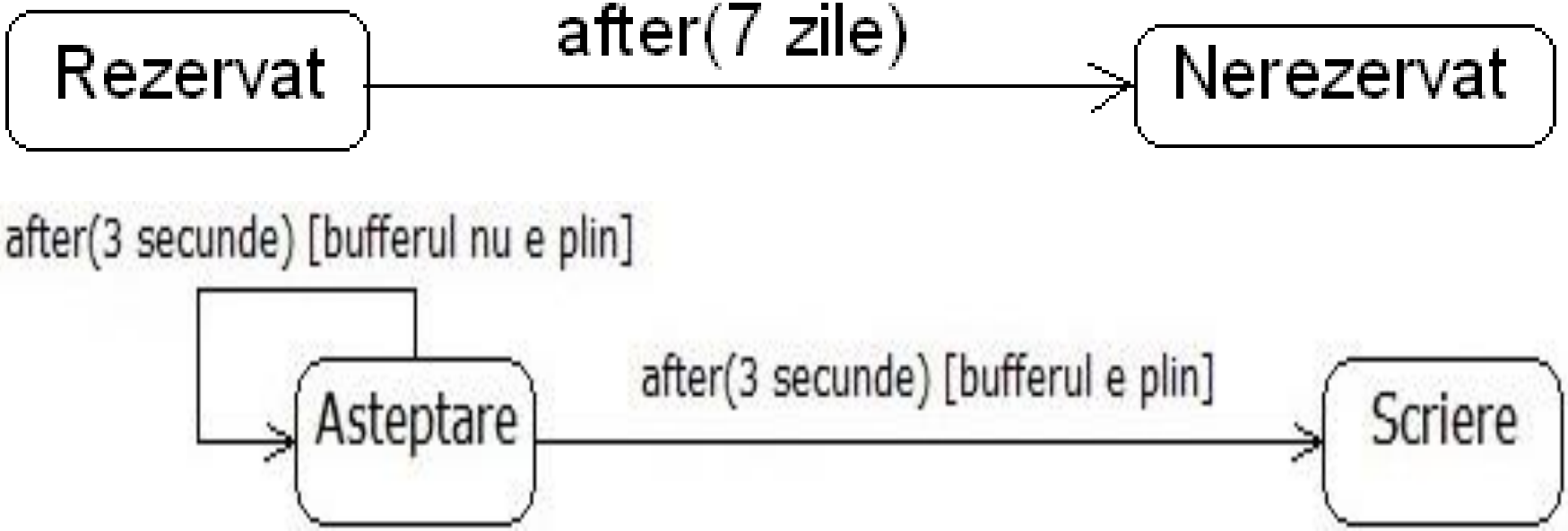
Modelarea evenimentelor de tip “conditie”

modelate prin cuvantul cheie ***when*** urmat de o expresie booleana scrisa intre paranteze



Modelarea evenimentelor de tip “dupa o perioada de timp”

modelate prin cuvantul cheie ***after*** urmat de o expresie, inclusa intre paranteze, care prin evaluare da un rezultat de tip “perioada de timp”



Tranzitie

= trecerea dintr-o stare in alta

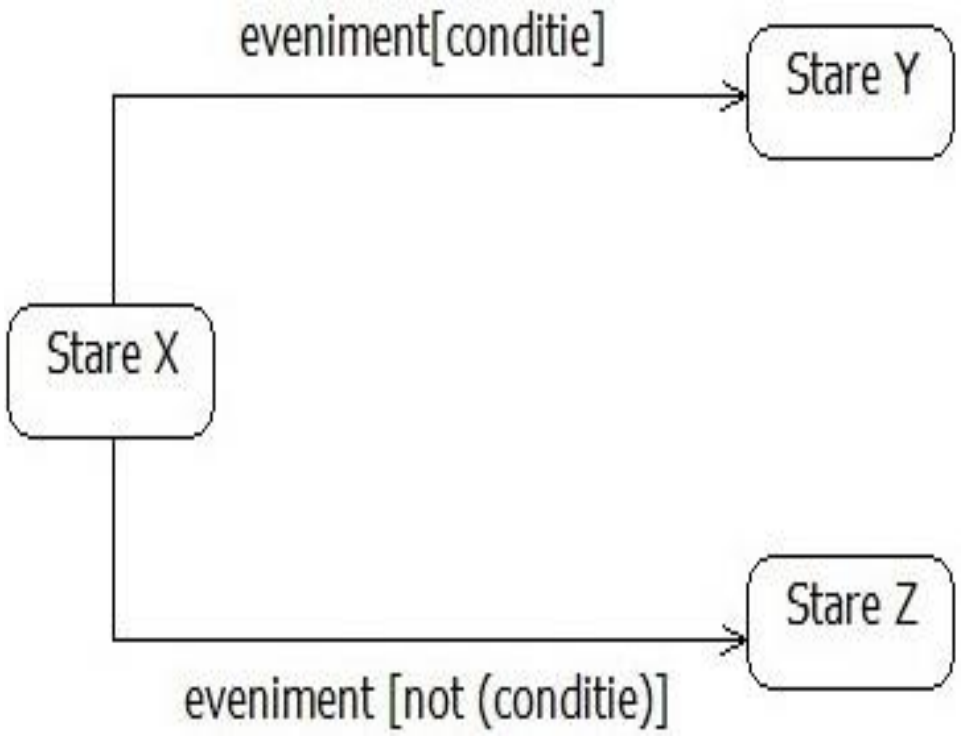
- Reprezentata prin sageata orientata intre stari
- Declansata de evenimente
- (optional) controlata de o garda:

  conditie booleana care valideaza declansarea unei tranzitii in cazul aparitiei unui eveniment

Tranzitie (conditionata)

Nota:

In cazul in care mai multe tranzitii pot fi declansate de acelasi eveniment, garzile trebuie sa fie mutual exclusive



Actiuni

- Efectuate de obiecte in momentul unei tranzitii
- Sunt considerate instantanee

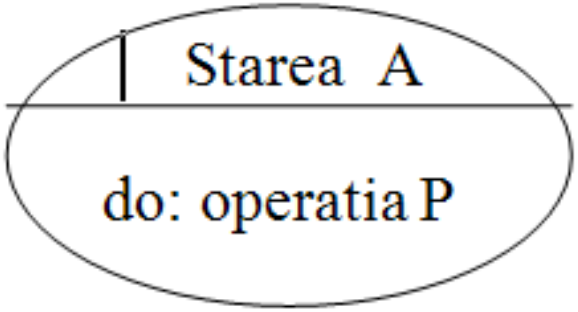
  (timp de executie neglijabil in raport cu dinamica sistemului)
- Exemple:
  - apelul unei operatii
  - trimiterea unui semnal
  - crearea/distrugerea unui alt obiect
  - evaluarea unei expresii etc

Expresia completa pentru o tranzitie

nume eveniment [([lista de parametri])] [conditie garda] / [expresie actiune]

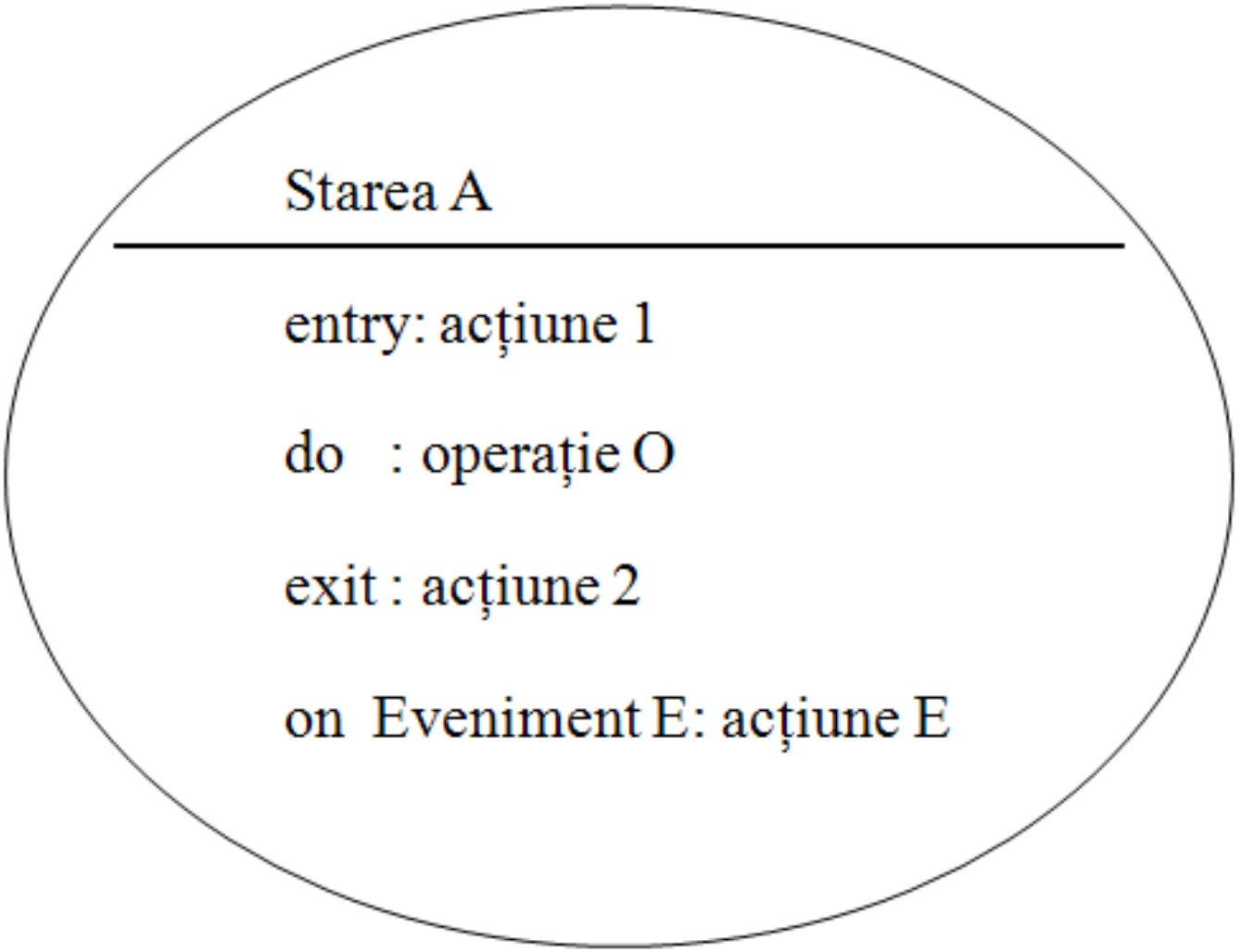
Activitati

- operatii care necesita un anumit timp de executie
- asociate starilor
- pot fi repetitive
- indicate prin cuvantul cheie "do"



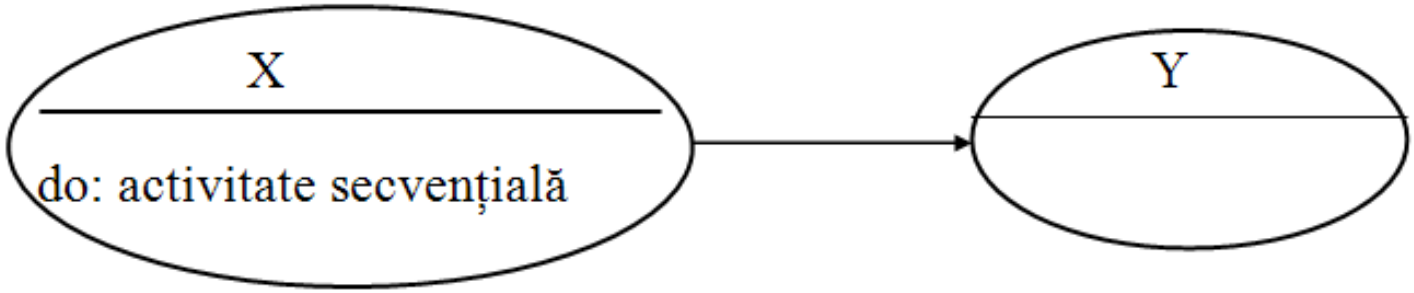
pot fi intrerupte in orice moment, daca se iese din starea corespunzatoare.

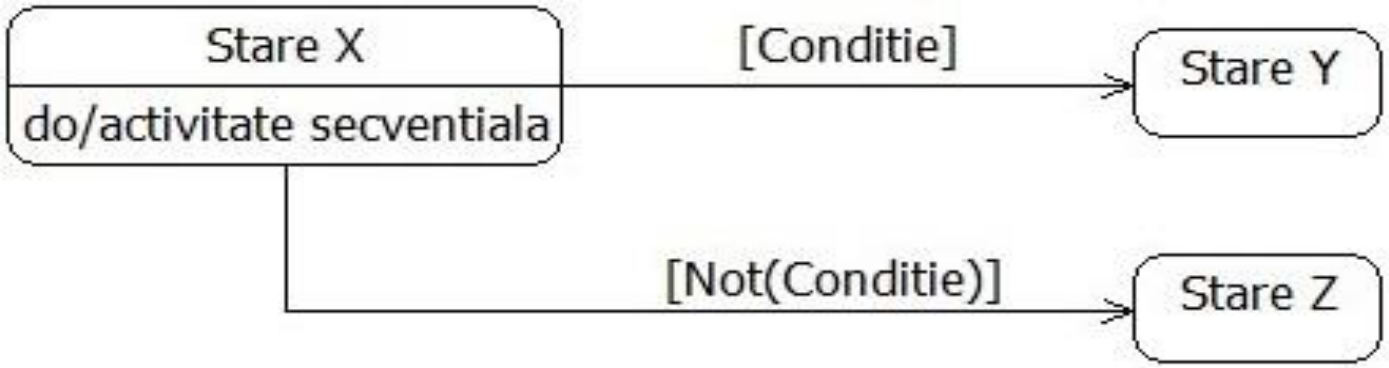
Activitati si actiuni in stari: notatii



Tranzitie automata

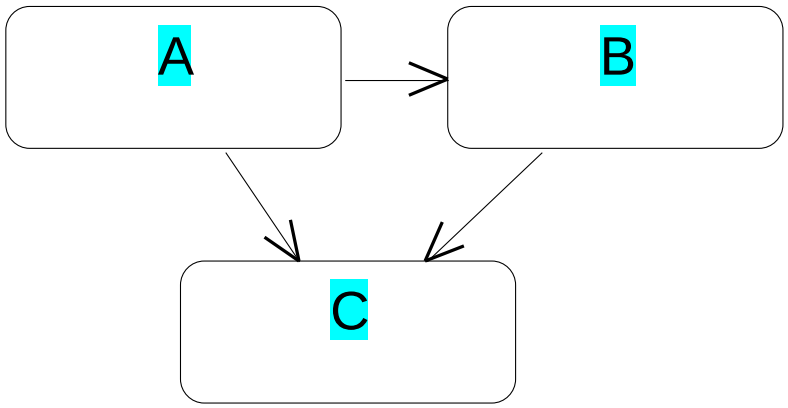
= in cazul incheierii unei activitati secventiale, daca tranzitia nu este marcata printr-un eveniment:

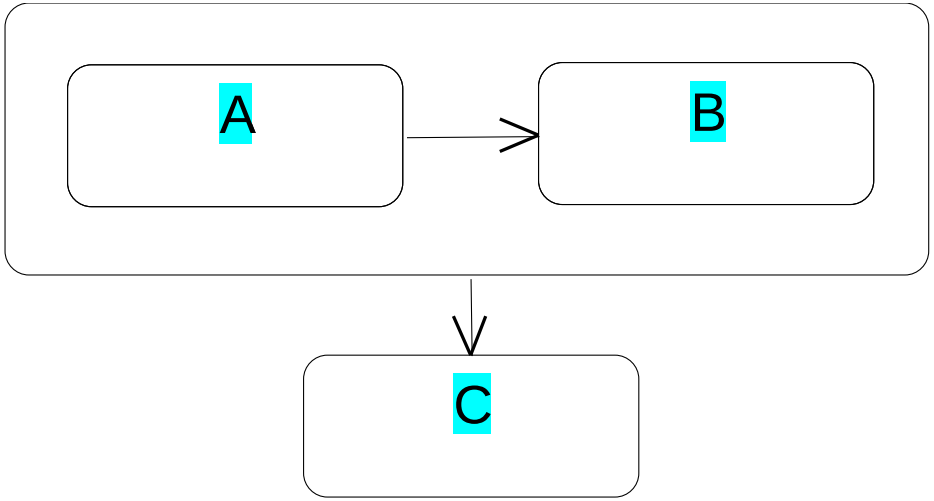




Managementul complexitatii
Abstractizare

- diagramele de stari pot deveni foarte complicate
- problema poate fi rezolvata prin abstractizare:
  - mai multe stari pot fi abstractizate intr-o singura stare
  - o stare poate fi descompusa in sub-stari disjuncte

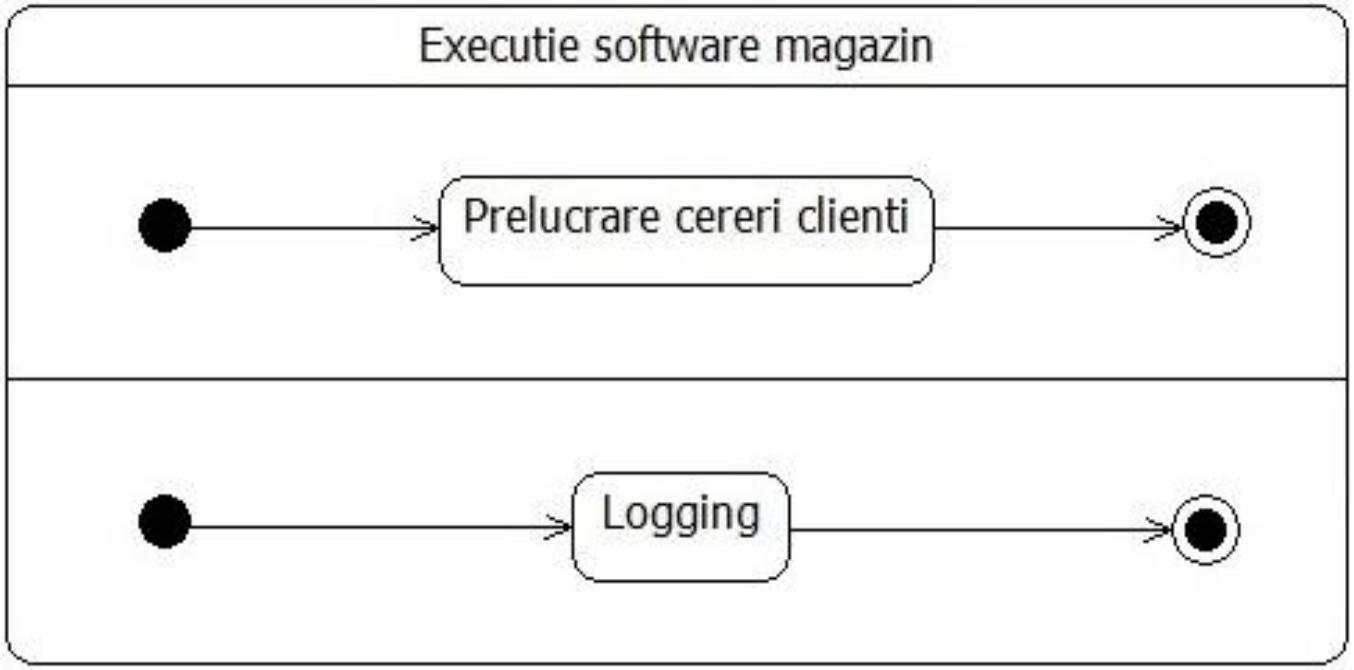


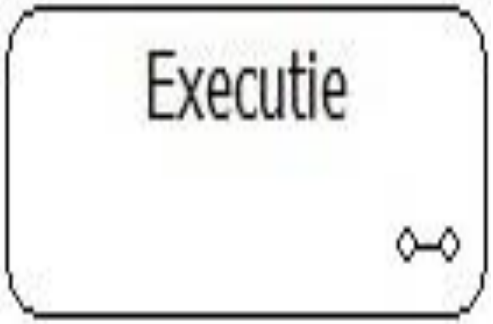


Automatele acceptate de UML sunt **deterministe**

- Pentru fiecare nivel de abstractizare exista o singura stare initiala
- Este posibil sa existe mai multe stari finale, fiecare corespunzand unei conditii de sfarsit diferite.
- De asemenea, este posibil sa nu existe nici o stare finala. Este cazul unui sistem care nu se opreste niciodata.

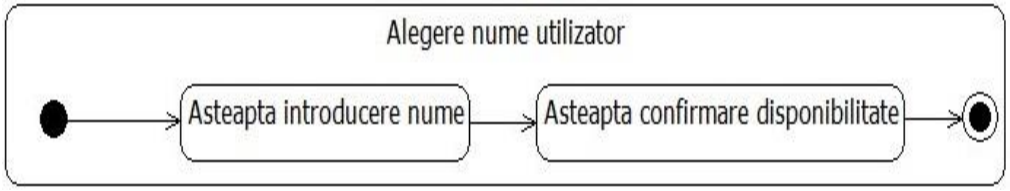
Substari: secventiale si paralele



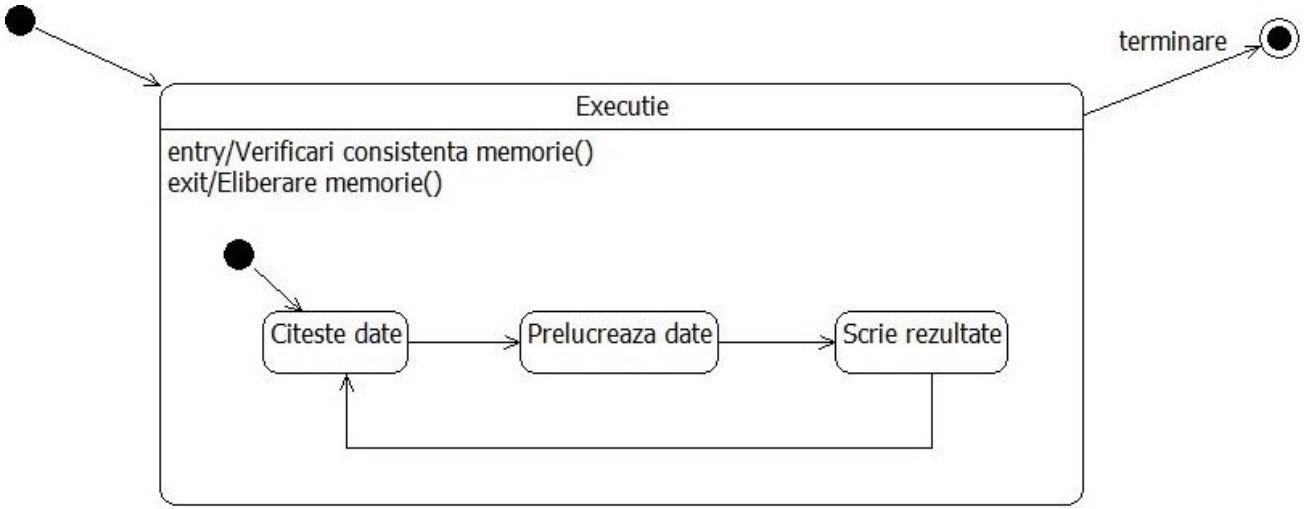


Stari compuse (1)

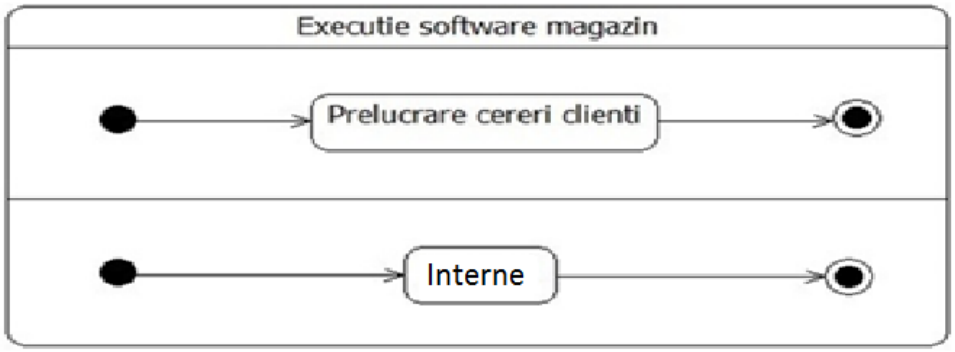
- O stare ce conține substări se numește *stare compusă*. O stare ce nu conține substări se numește *stare simplă*.
- Substările unei stări compuse pot fi secvențiale (disjuncte) sau concurente (ortogonale).



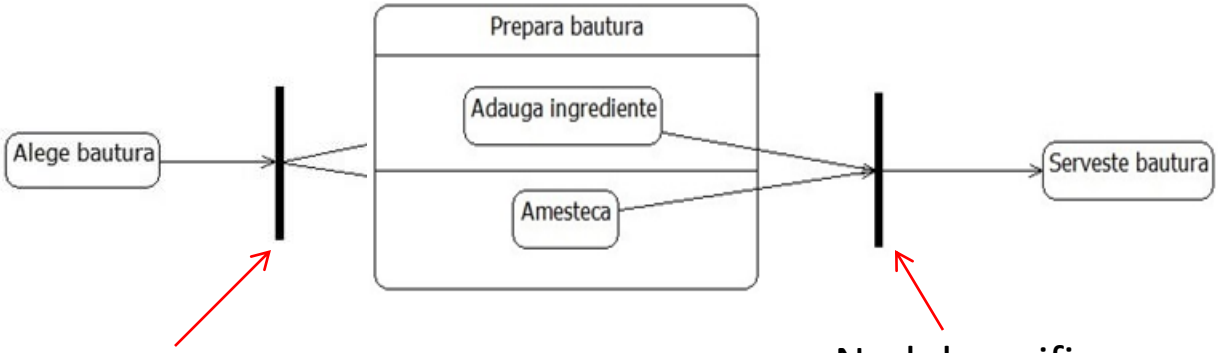
Stare compusa cu 2 substari secventiale



Stari compuse (2)



Stare compusa cu 2 substari concurente.



Nod de ramificare in cele 2 stari ortogonale.

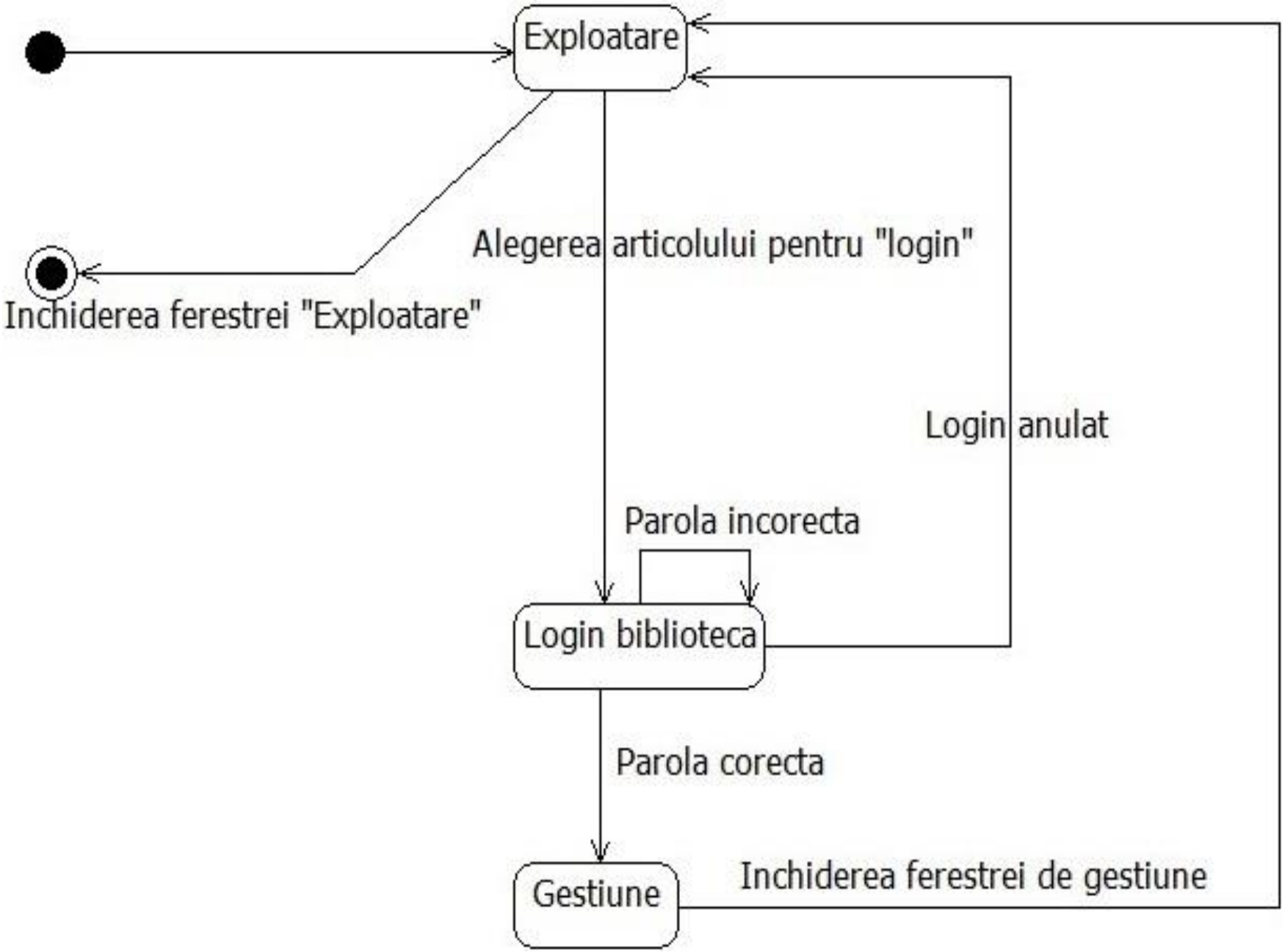
Nod de unificare

Diagrame de stari
pentru modelarea scenariilor

- Diagramele de stari pot fi folosite si pentru modelarea scenariilor dintr-un caz de utilizare
- O diagrama de stari modeleaza un caz de utilizare sau mai multe scenarii

Intreg sistemul este obiectul

ale carui stari sunt modelate


Exploatare
Alegerea articolului pentru "login"
Inchiderea ferestrei "Exploatare"
Login anulat
Parola incorecta
Login biblioteca
Parola corecta
Gestiune
Inchiderea ferestrei de gestiune

Pseudostari

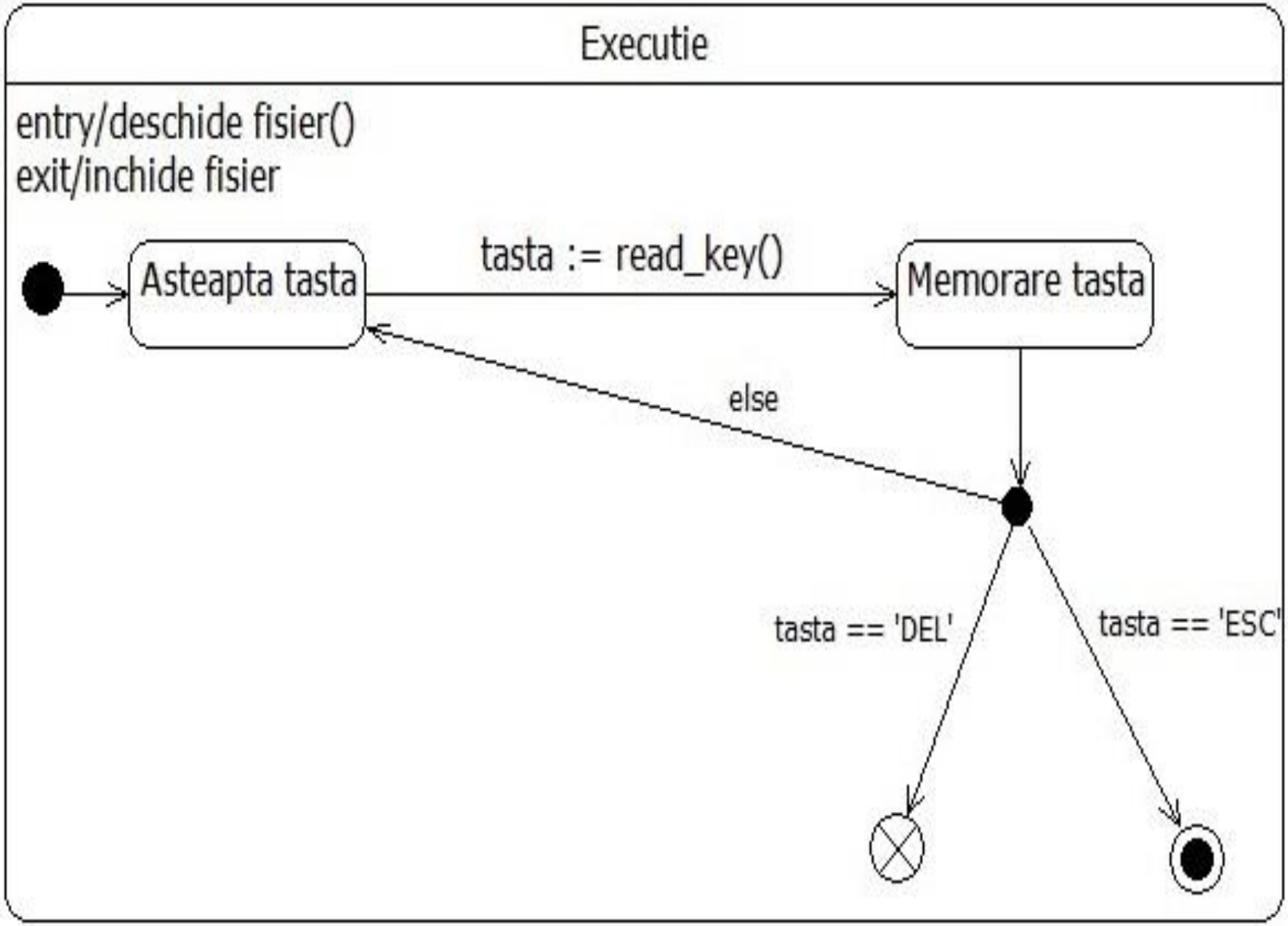
- Ramificatie conditionata
- Ramificare
- Unificare
- Puncte de convergenta si divergenta
- Referirea activarilor anterioare
- Terminare imediata

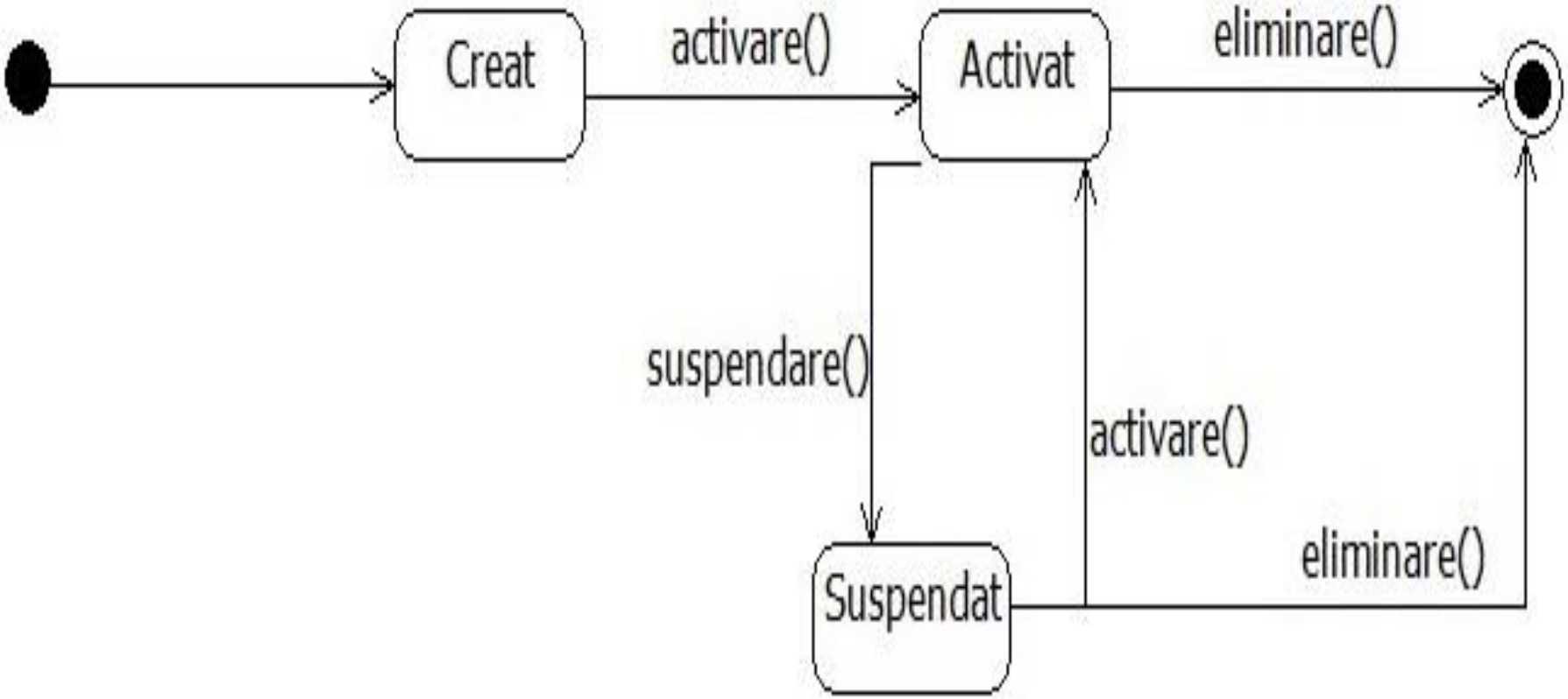
nu le detaliem in curs, nu sunt materie de examen
sunt prezentate pe larg in carte

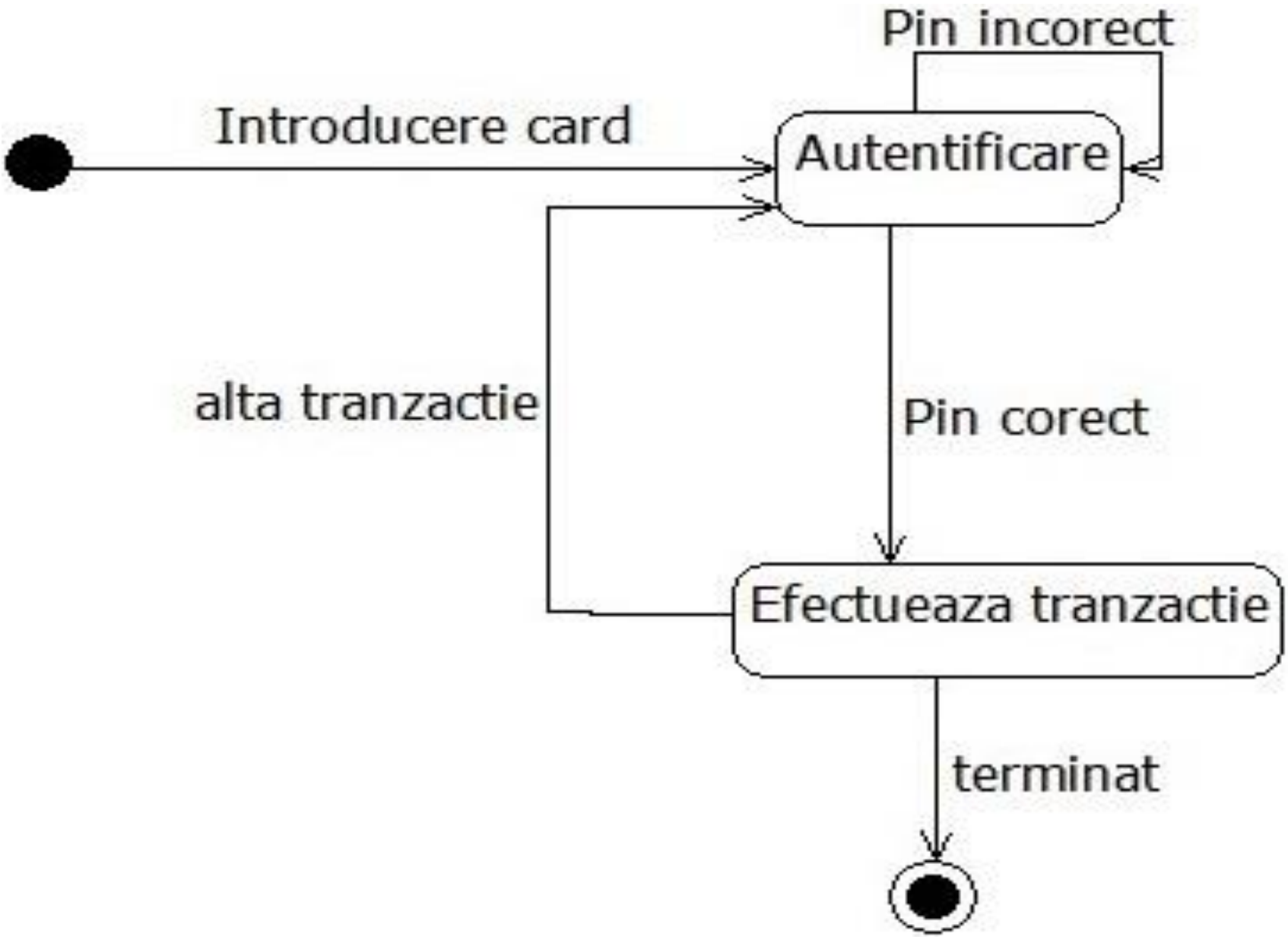
NOTA: alte notatii folosite in diagramele de stari →

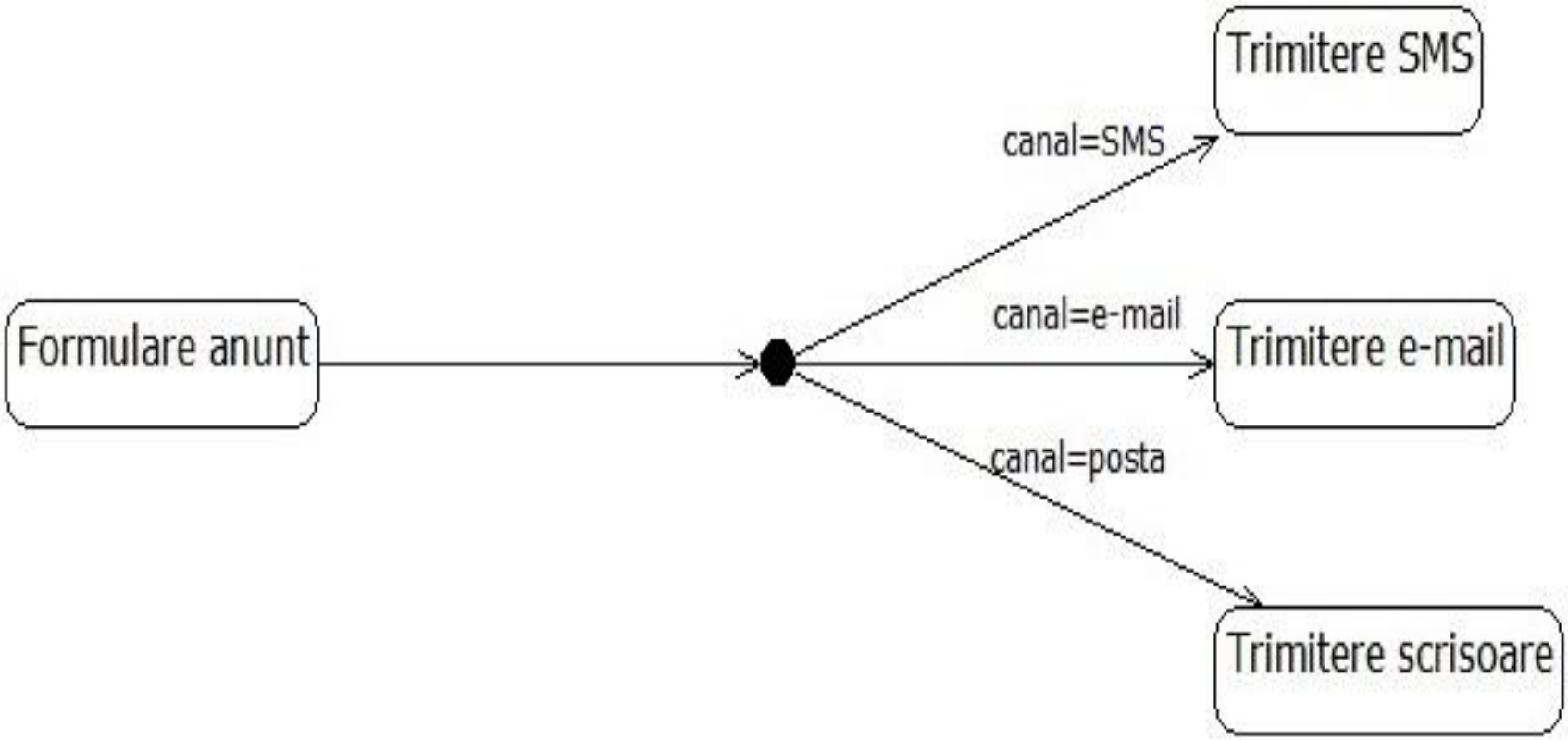
UML practic, Ed. MatrixRom, 2014.

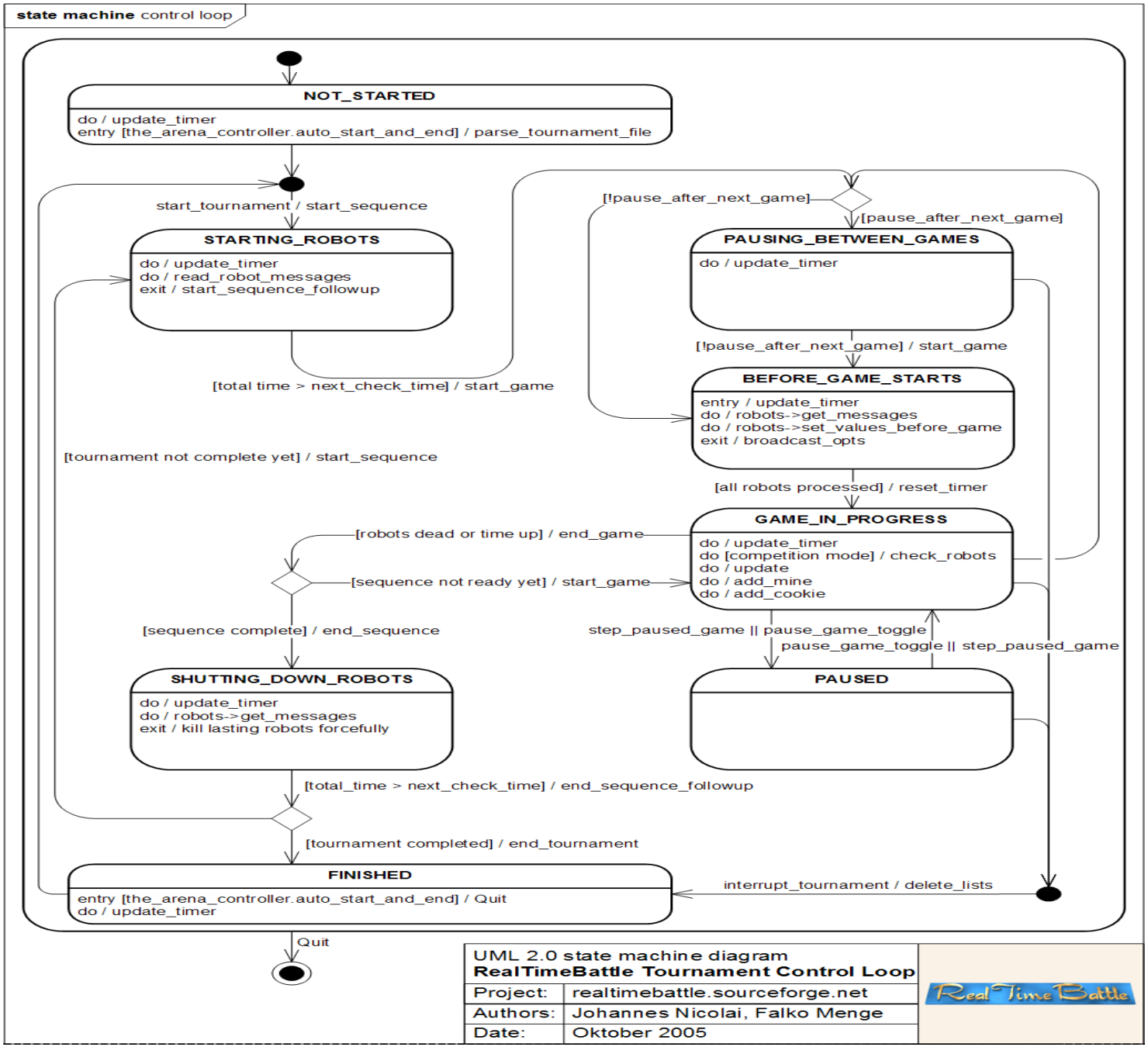
EXEMPLE DIAGRAME DE STARI


Executie
entry/deschide fisier()
exit/inchide fisier
Asteapta tasta
tasta := read_key()
Memorare tasta
else
tasta == 'DEL'
tasta == 'ESC'

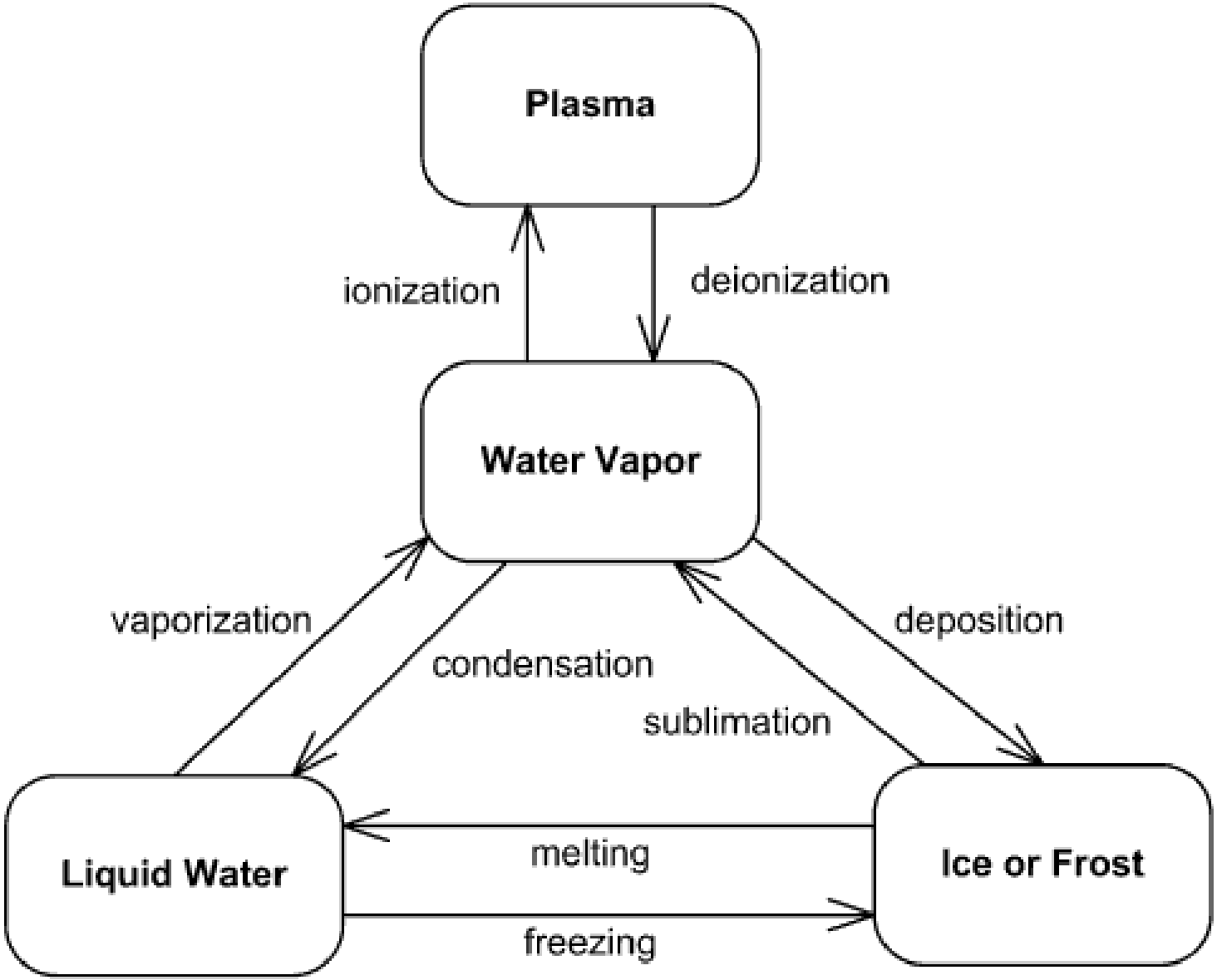

Creat
activare()
Activat
eliminare()
suspendare()
activare()
eliminare()
Suspendat

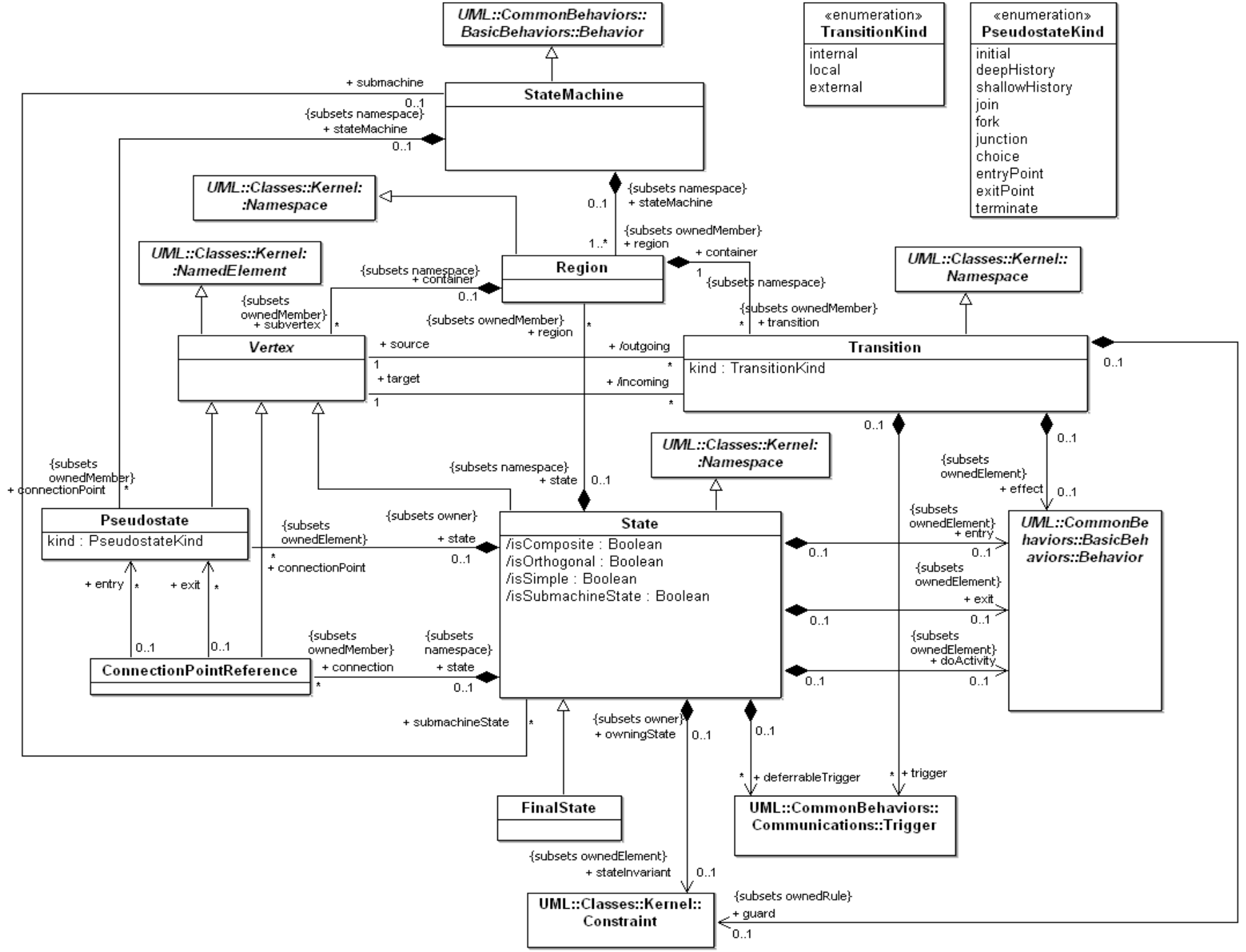

Pin incorect
Introducere card
Autentificare
alta tranzactie
Pin corect
Efectueaza tranzactie
terminat


Trimitere SMS
canal=SMS
Formulare anunt
canal=e-mail
Trimitere e-mail
canal=posta
Trimitere scrisoare



UML 2.0 state machine diagram **RealTimeBattle Tournament Control Loop**	
Project:	realtimebattle.sourceforge.net
Authors:	Johannes Nicolai, Falko Menge
Date:	Oktober 2005

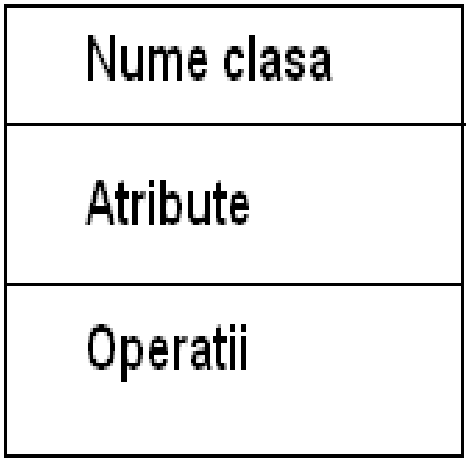

Plasma
ionization
deionization
Water Vapor
vaporization
condensation
deposition
sublimation
Liquid Water
melting
freezing
Ice or Frost


UML::CommonBehaviors::
BasicBehaviors::Behavior
«enumeration»
TransitionKind
internal
local
external
«enumeration»
PseudostateKind
initial
deepHistory
shallowHistory
join
fork
junction
choice
entryPoint
exitPoint
terminate
StateMachine
+ submachine
0..1
{subsets namespace}
+ stateMachine
0..1
UML::Classes::Kernel:
:Namespace
{subsets namespace}
+ stateMachine
0..1
1..*
{subsets ownedMember}
+ region
+ container
1
{subsets namespace}
UML::Classes::Kernel:
:NamedElement
Region
{subsets namespace}
+ container
0..1
UML::Classes::Kernel::
Namespace
{subsets
ownedMember}
+ subvertex
*
{subsets ownedMember}
+ region
*
{subsets ownedMember}
* + transition
Vertex
+ source
1
+ target
1
+ /outgoing
*
+ /incoming
*
Transition
kind : TransitionKind
0..1
0..1
0..1
UML::Classes::Kernel:
:Namespace
{subsets
ownedMember}
+ connectionPoint
*
{subsets namespace}
+ state
0..1
{subsets
ownedElement}
+ effect
0..1
Pseudostate
kind : PseudostateKind
{subsets
ownedElement}
+ connectionPoint
*
{subsets owner}
+ state
0..1
State
/isComposite : Boolean
/isOrthogonal : Boolean
/isSimple : Boolean
/isSubmachineState : Boolean
{subsets
ownedElement}
+ entry
0..1
0..1
UML::CommonBe
haviors::BasicBeh
aviors::Behavior
{subsets
ownedElement}
+ exit
0..1
0..1
{subsets
ownedElement}
+ doActivity
0..1
0..1
+ entry
*
+ exit
*
0..1
0..1
ConnectionPointReference
{subsets
ownedMember}
+ connection
*
{subsets
namespace}
+ state
0..1
+ submachineState
*
{subsets owner}
+ owningState
0..1
0..1
* + deferrableTrigger
* + trigger
FinalState
UML::CommonBehaviors::
Communications::Trigger
{subsets ownedElement}
+ stateInvariant
0..1
UML::Classes::Kernel::
Constraint
{subsets ownedRule}
+ guard
0..1

UML - Clase si diagrame de clase

CLASA

- Reprezinta un grup de obiecte care au:
  - proprietati similare (atribute)
  - un comportament comun (operatii)
  - relatii comune cu alte obiecte
  - o aceeasi semantica.

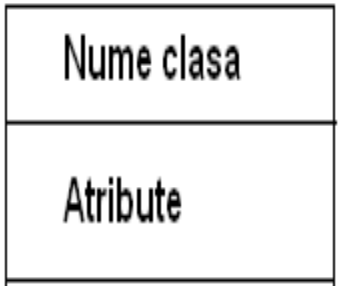


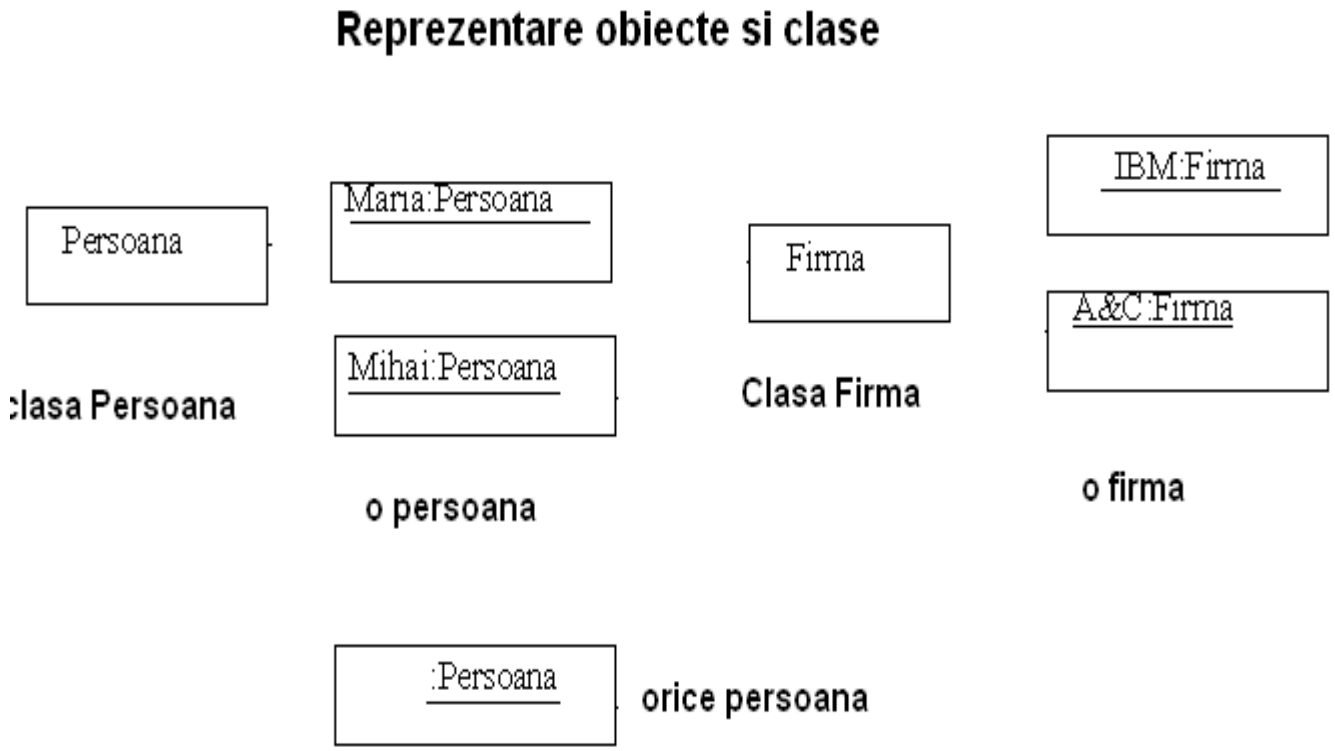
Compartimentul atributelor si cel al operatiilor pt fi omise

Clase conceptuale

- **Clasele definite in etapa de „Analiza a cerintelor”**

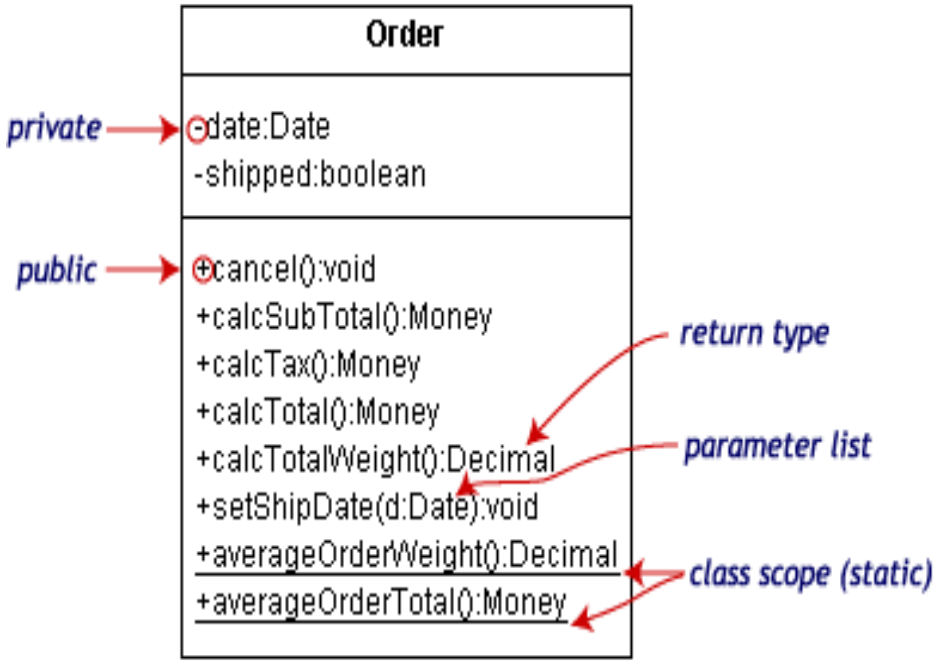
contin, de regula, numai numele clasei si atributele (de regula fara specificarea tipului fiecarui atribut).





Reprezentarea detaliata a unei clase

- **Este construita in etapa de proiectare de detaliu**
- **Precizeaza:**
  - vizibilitatea informatiilor din clasa,
  - tipul atributelor
  - lista de parametri a fiecarei operatii si tipul parametrilor.



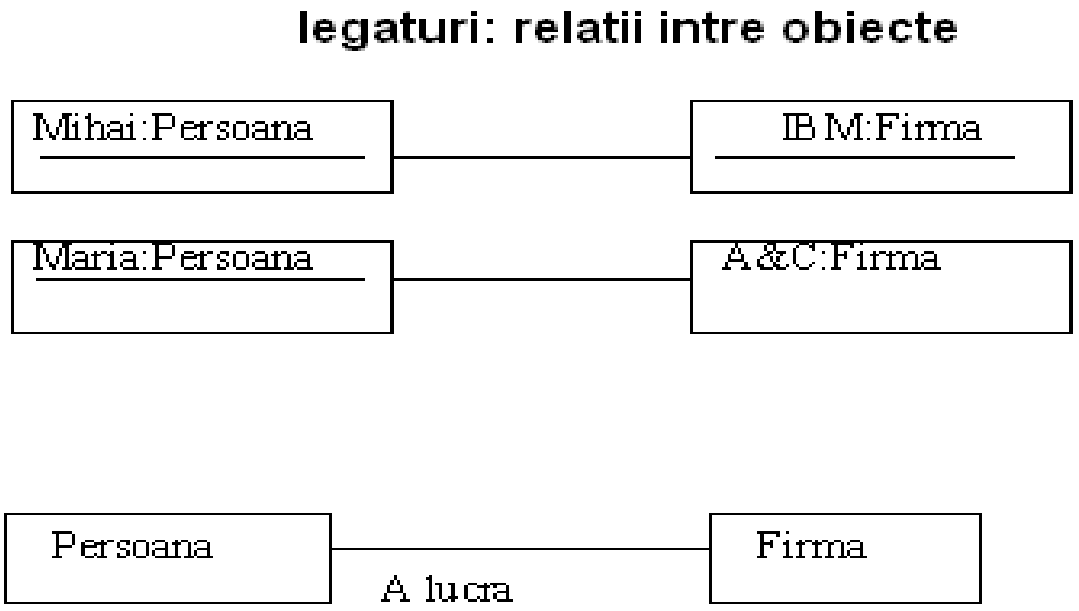
DIAGRAME DE CLASE

Redau structura statica obiectuala a unui software.

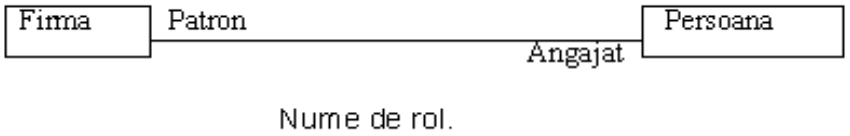
Relatii intre clase:

- asociere
- generalizare

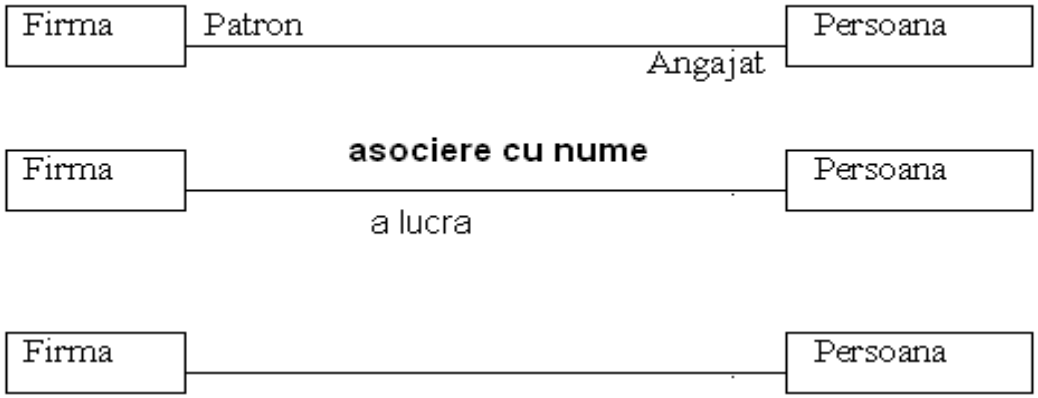
Asocierea: o abstractie a unui set de legaturi intre obiecte



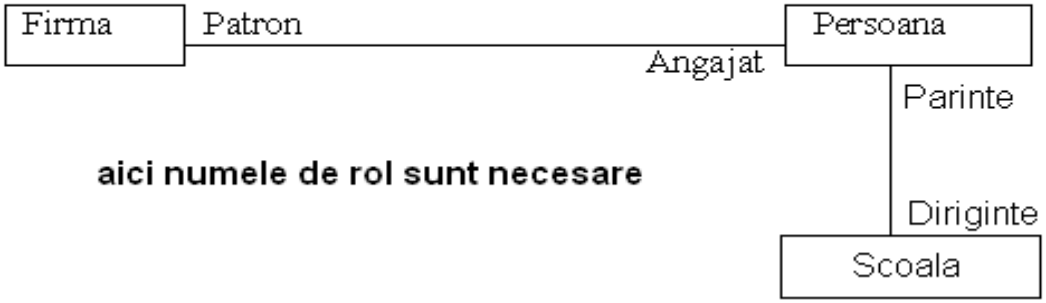
Extremitatea unei asocieri este numita ROL



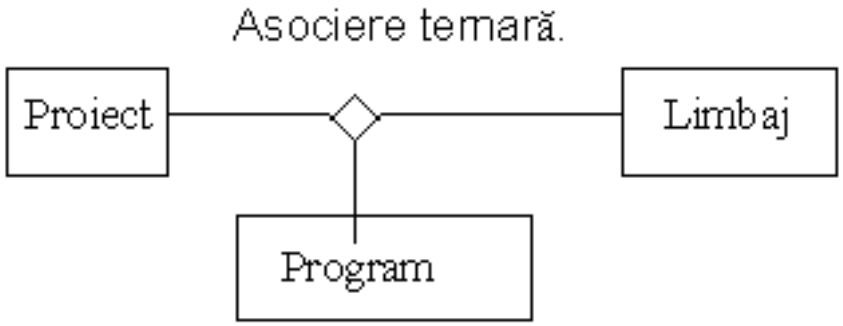
Rol: felul in care o clasa “vede” o alta clasa intr-o relatie de asociere



Nume de asociere
si
nume de roluri

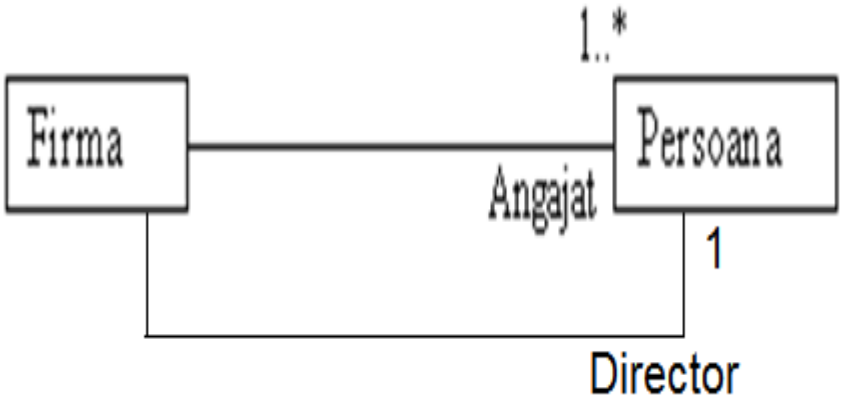


Aritatea asocierilor



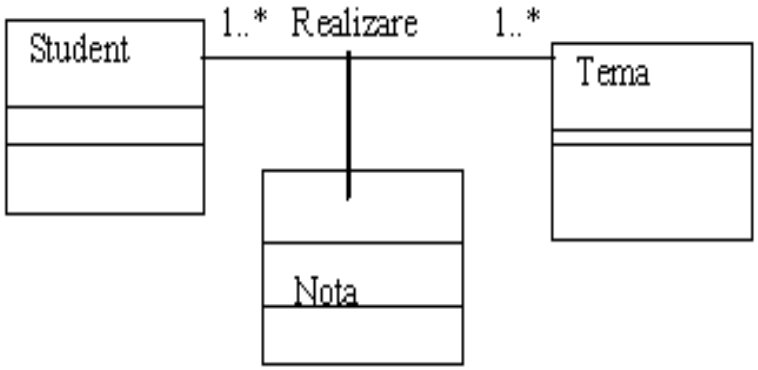
"Proiectele sunt implementate prin Programe scrise în Limbaje de programare".

Multiplicitatea asocierilor: cate obiecte ale unei clase pot fi legate unui obiect al celeilalte clase



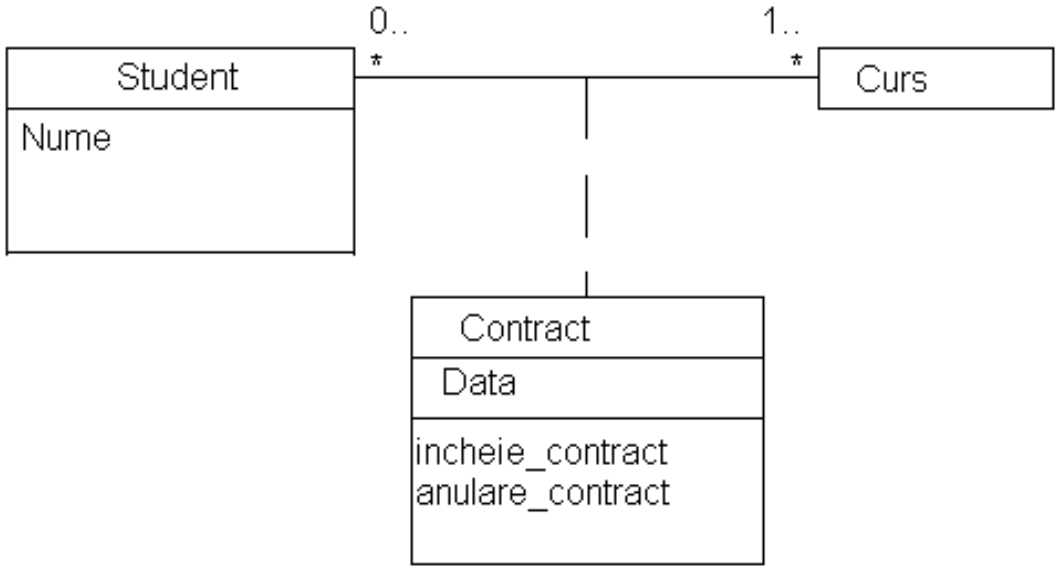
1
4
0..* sau *
n1..n2

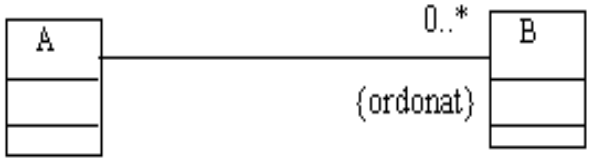
Multiplicitatea unei asocieri exprima o constrangere valabila pe toata durata de existenta a obiectelor claselor asociate



Asocieri cu atribute.

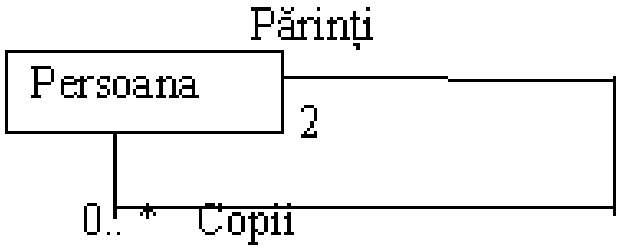
Clasa asociere





Asociere constrânsă.

Instantele clasei B asociate unei instante a clasei A trebuie sa fie irdonate

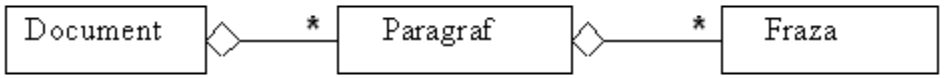


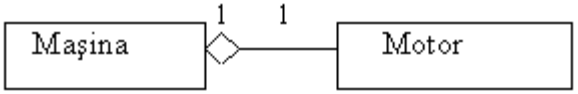
Asociere reflexivă

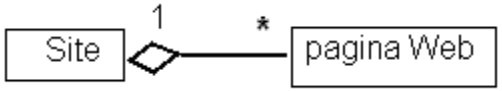
Numirea rolurilor este în acest caz esentiala pentru claritatea diagramei

Agregarea

- Forma particulara de asociere care exprima un cuplaj de tipul “compus-componenti”:



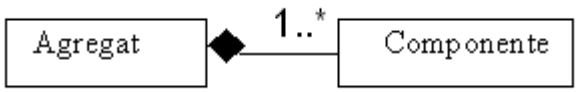


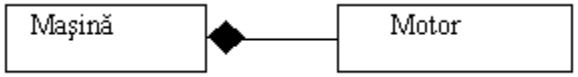


- Agregatul nu poate exista fara una dintre componente
- Distrugerea agregatului nu conduce la distrugerea componentelor

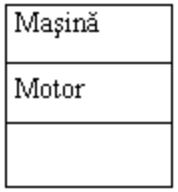
Compunerea

- **O agregare prin continere fizica**





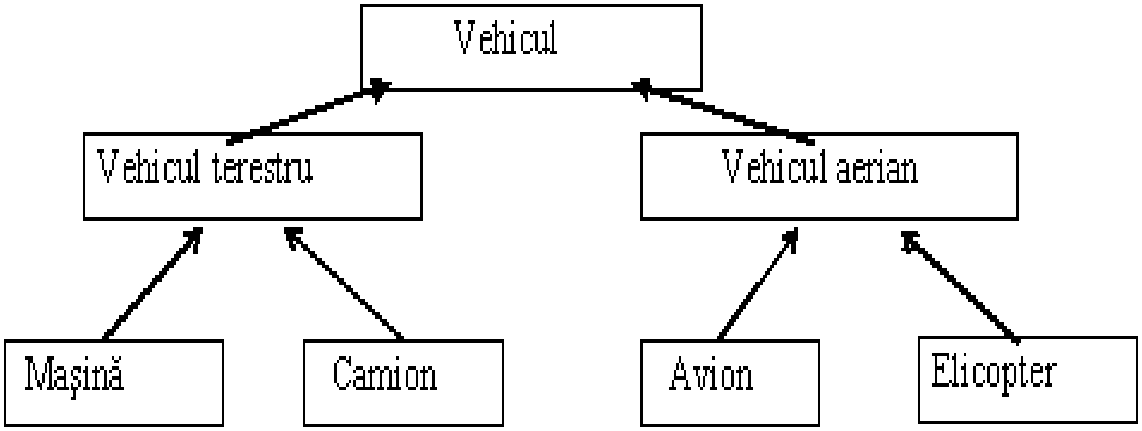
Echivalent cu:



- **Distrugerea agregatului conduce la distrugerea componentelor**

Generalizarea

clasificare, generalizare, specializare



Generalizarea:

factorizarea elementelor comune (atribute, operatii si constrangeri)

ale unui ansamblu de clase într-o clasa mai generala, numit superclasa.

• Relatia de generalizare din UML este mai abstracta decat relatia de mostenire din limbajele POO

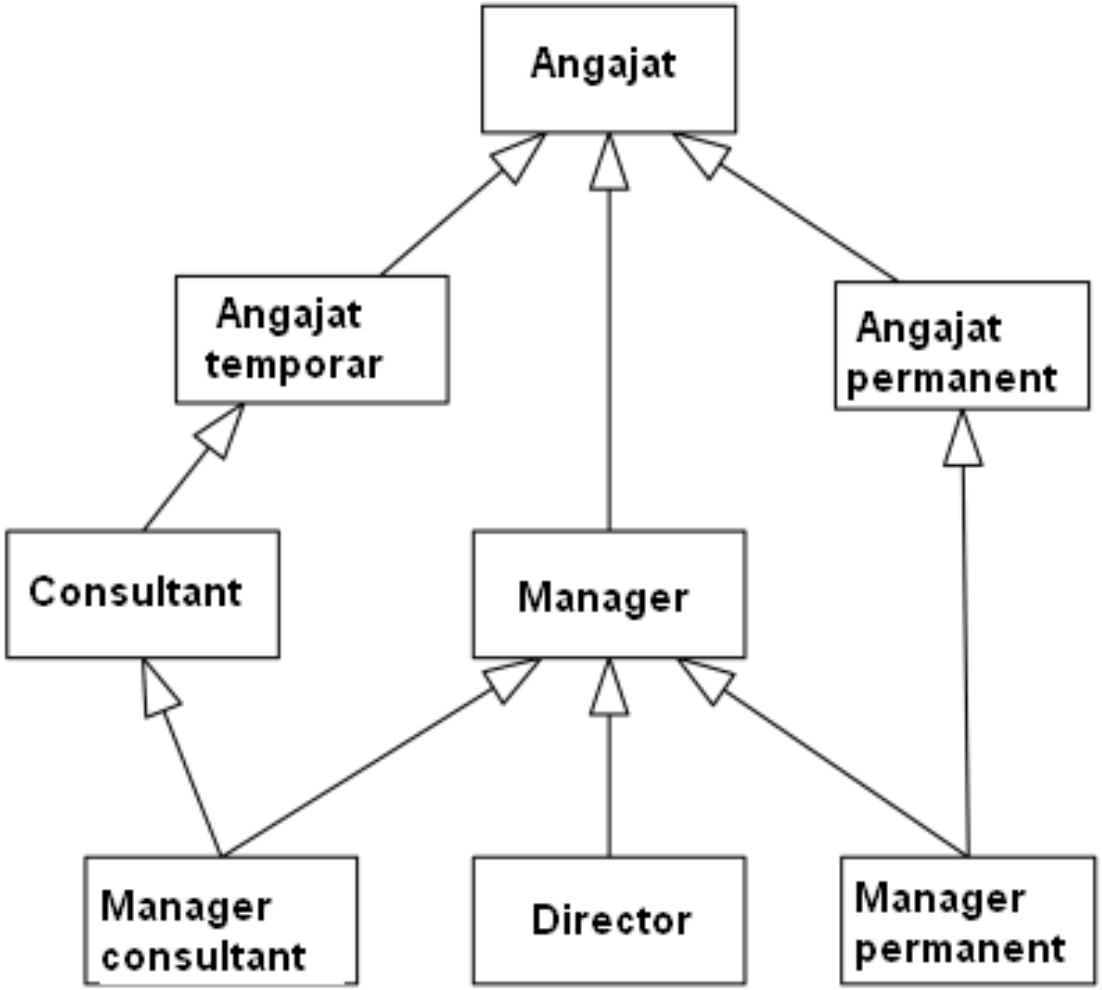
Ea este mai adecvata etapei de analiza (exista si intre cazuri de utilizare!).

Specializarea: capturarea particularitatilor unui ansamblu de obiecte,

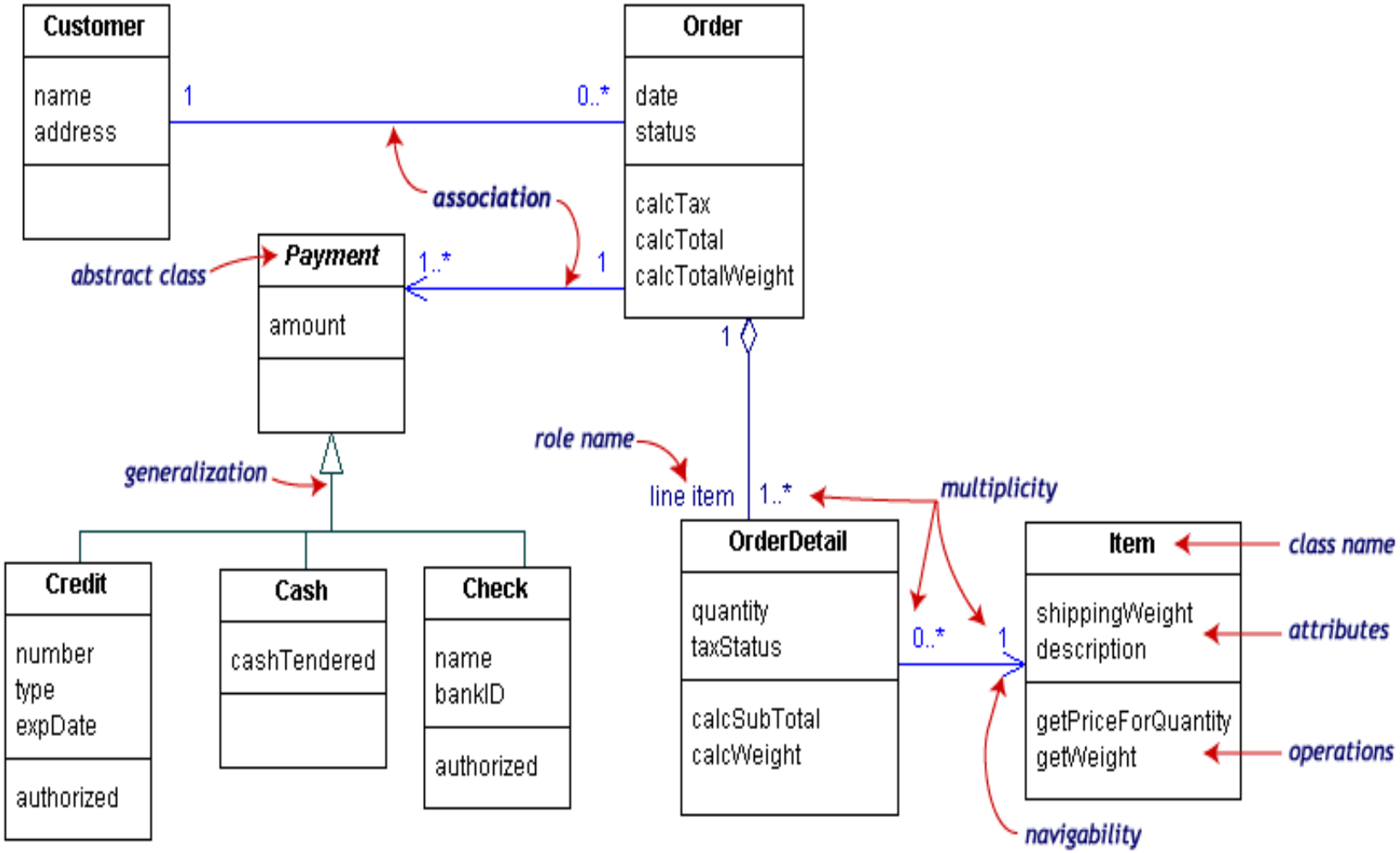
nereprezentate prin clasele existente.

Graful de mostenire:

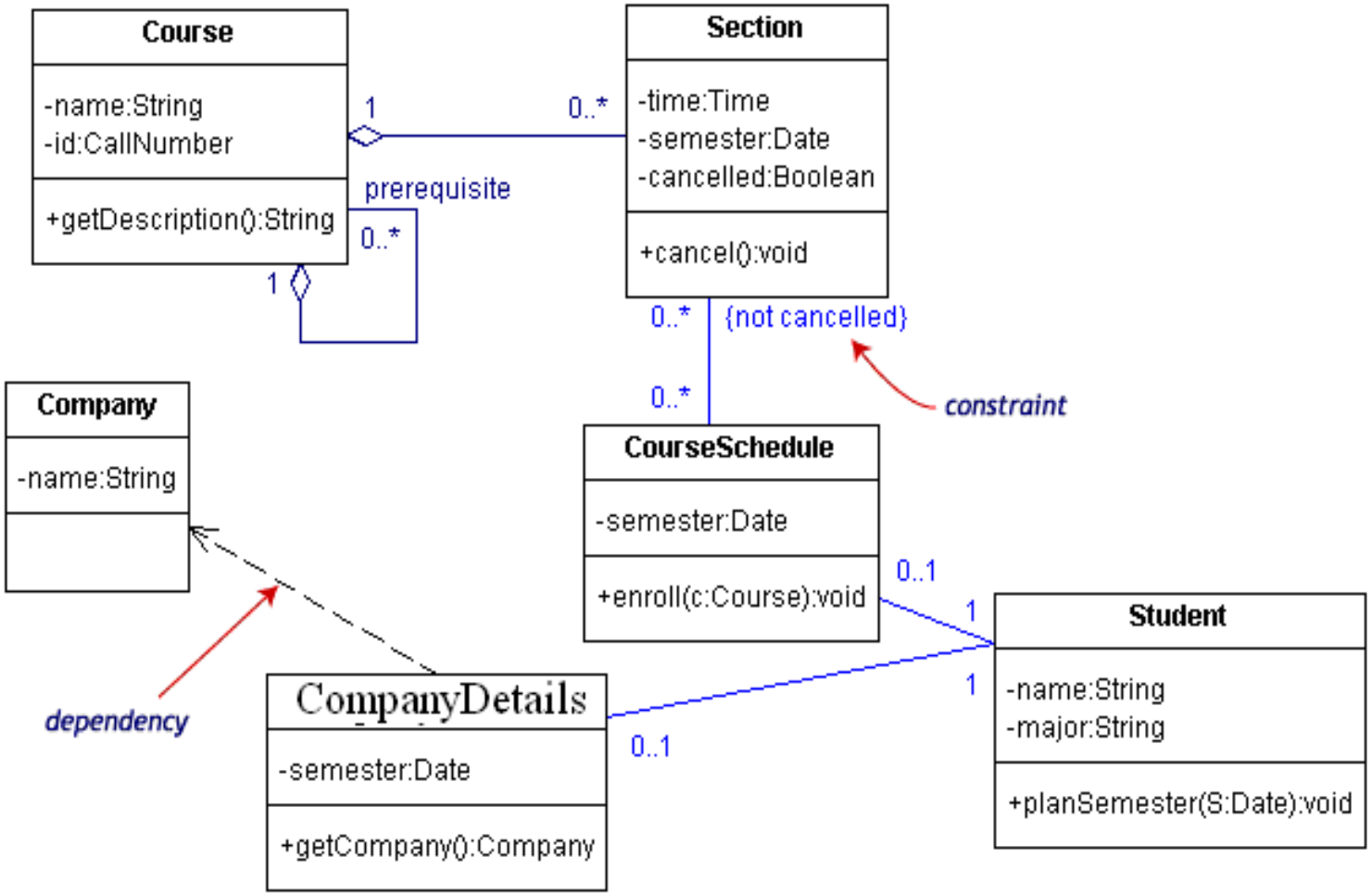
clasele pot avea mai multe super-clase



Navigabilitate



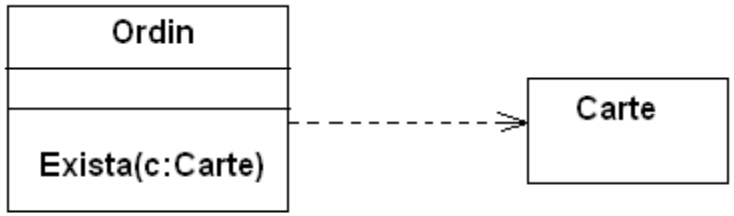
Dependente si constrangeri



Dependenta: o relatie slaba intre clase

Relatia de dependenţă (1)

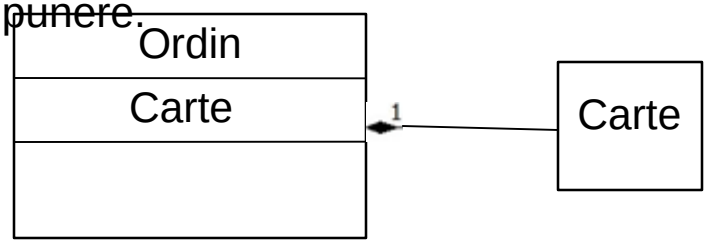
- Este cea mai slaba relatie între doua clase.
- Exprima de obicei o relatie temporara între obiecte: obiectele clasei dependente interactioneaza pentru scurt timp cu obiectele clasei tinta.
- De ex., o clasa A depinde de o clasa B, daca o metoda a clasei A are un parametru de tip



Client: elementul dependent Furnizor: elementul independent

Daca definitia clasei Carte se modifica, este posibil sa fie necesara modificarea functiei Exista().

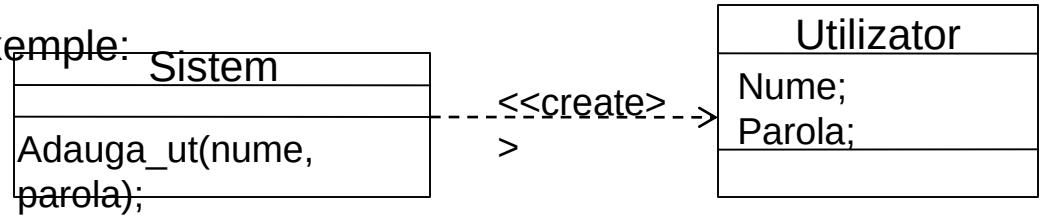
- Se deosebeste de cazul asocierii dintre A si B, cand un atribut al clasei A este o instanta a clasei B, caz in care intre A si B exista o relatie de agregare prin compunere.



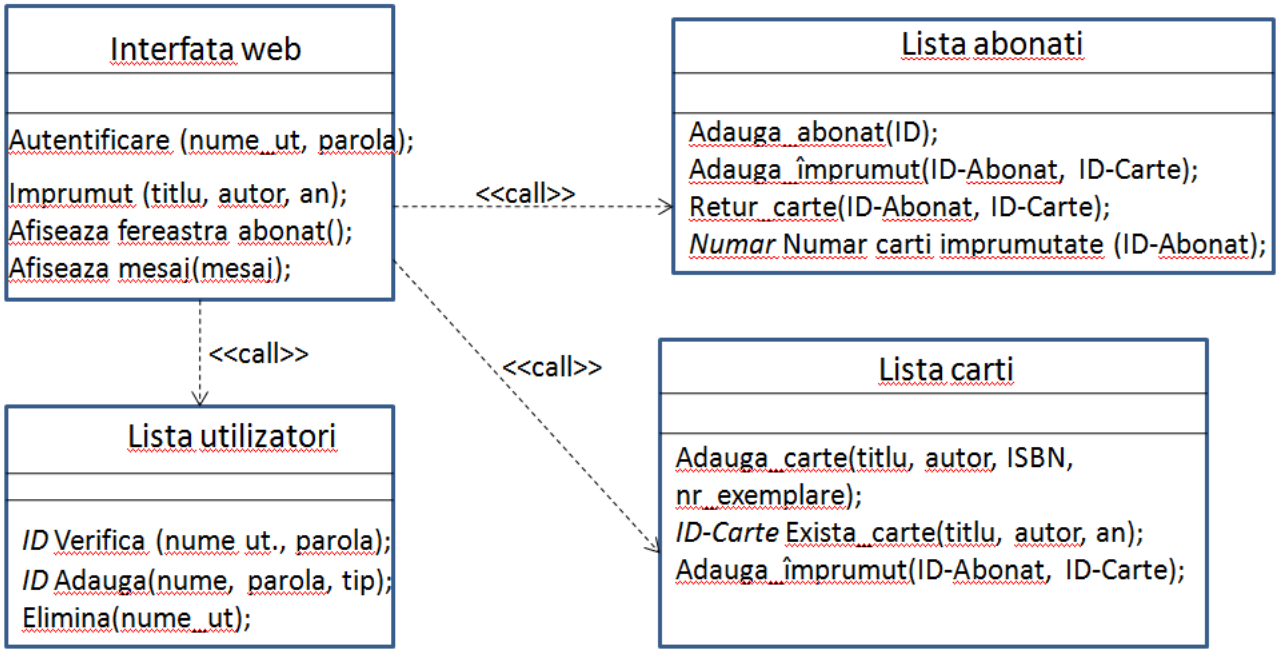
Relatia de dependenţă (2)

Relația de dependență poate fi adnotată cu un stereotip care oferă informații despre natura relației.

Exemple:



Obiectele clasei Utilizator sunt create de un obiect al clasei Sistem.



Interfata web apeleaza operatii din Lista utilizatori, Lista abonati si Lista carti.

Folosirea diagramelor de clase:

1) In modelarea conceptuala (analiza orientata obiect) si specificarea software

- Clasele corespund conceptelor / obiectelor (entitatilor) din domeniul aplicatiei
- Nu exista neaparat o legatura directa cu clasele de obiecte utilizate in implementare

  si deci diagrama de clase nu face parte din modelul structural al sistemului
- De regula, nu sunt definite operatiile din clase prin tipurile parametrilor si nici tipul atributelor.

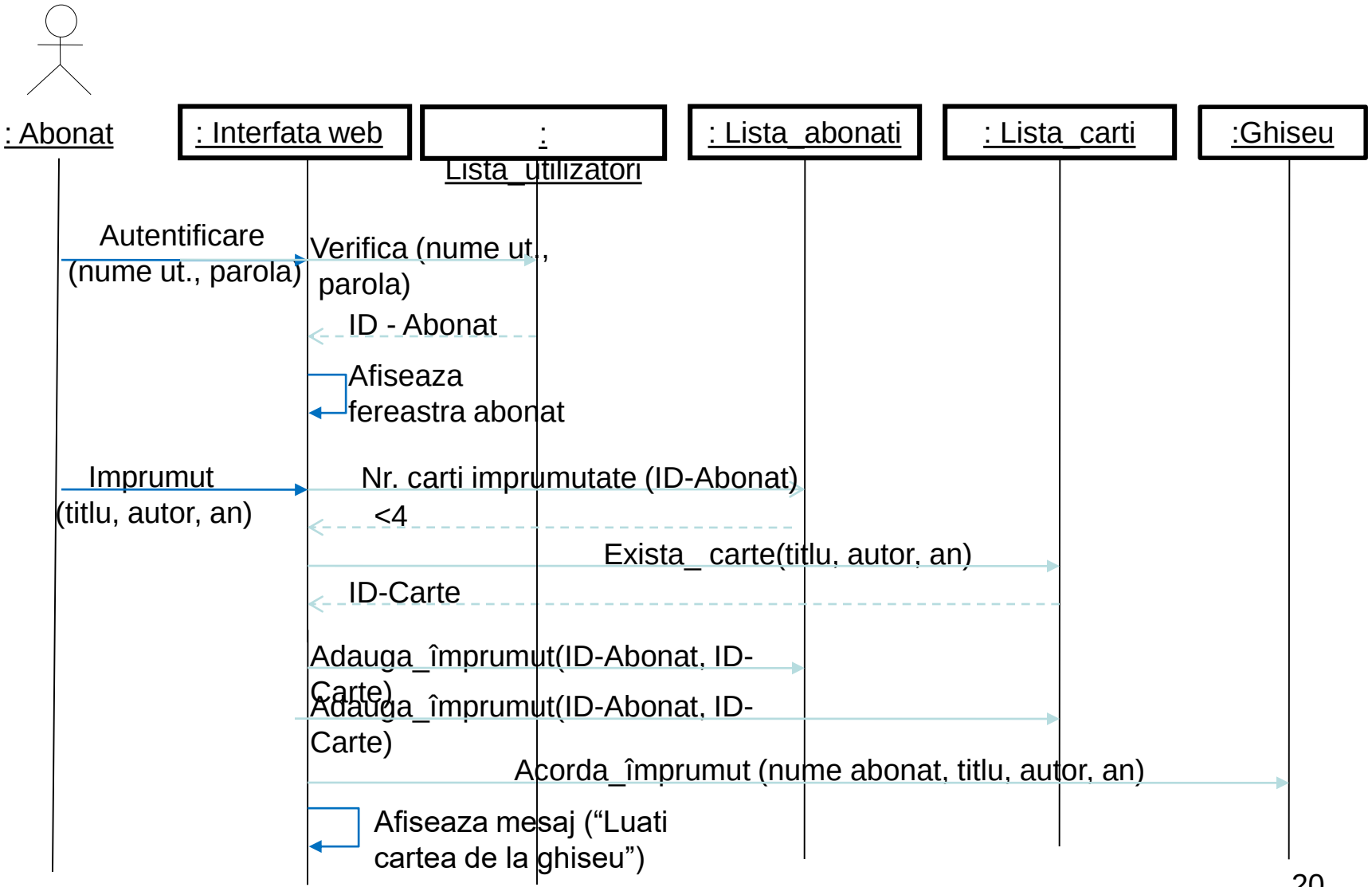
2) In proiectarea de detaliu si implementare

- Diagramele contin clase de obiecte intr-un anumit limbaj de programare
- Diagramele fac parte din modelul structural al sistemului

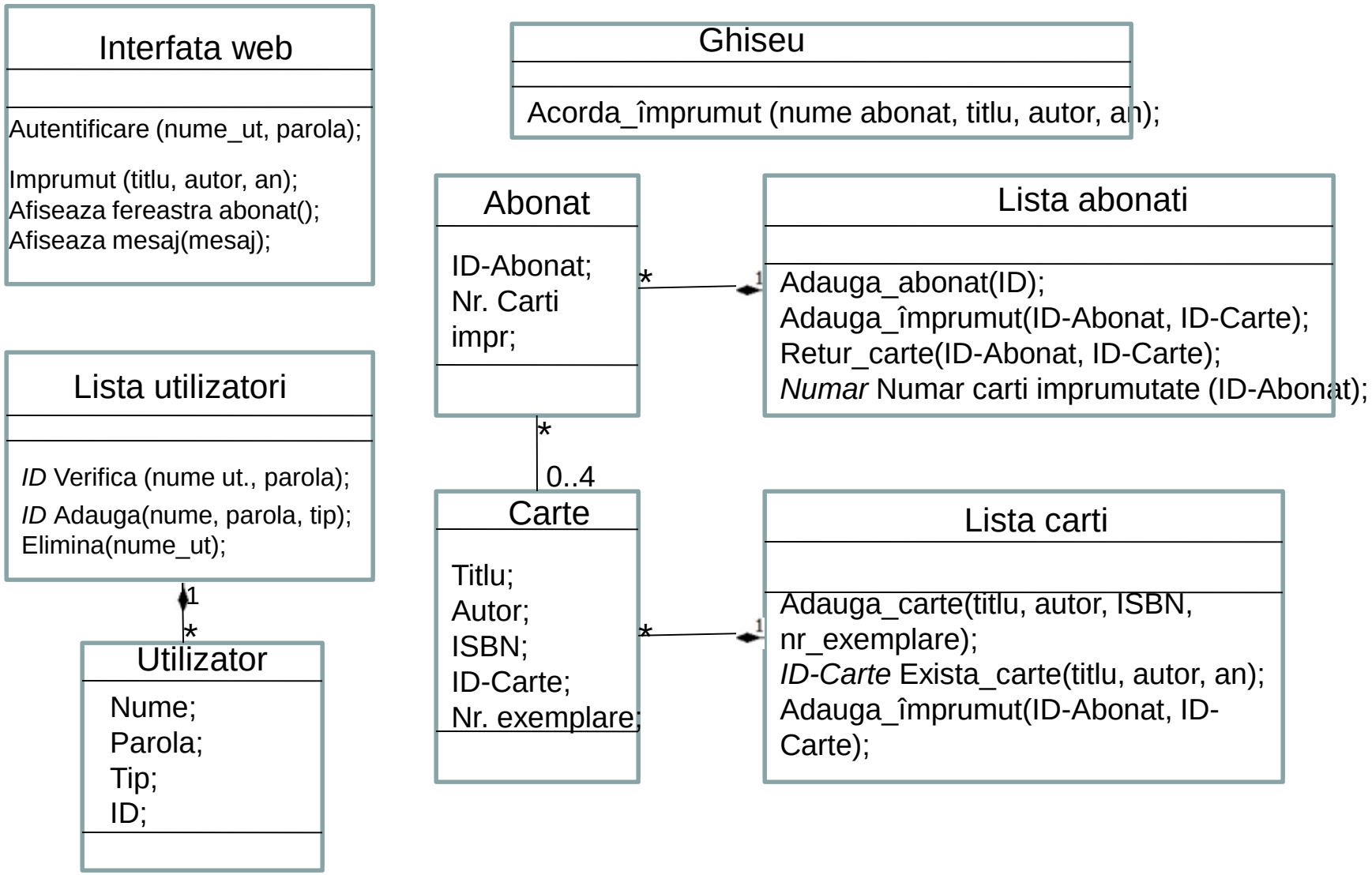
NOTA: alte notatii folosite in diagramele de clase →

UML practic, Ed. MatrixRom, 2014.

Exemplu:
de la diagrama de secventa la diagrama de clase (1)

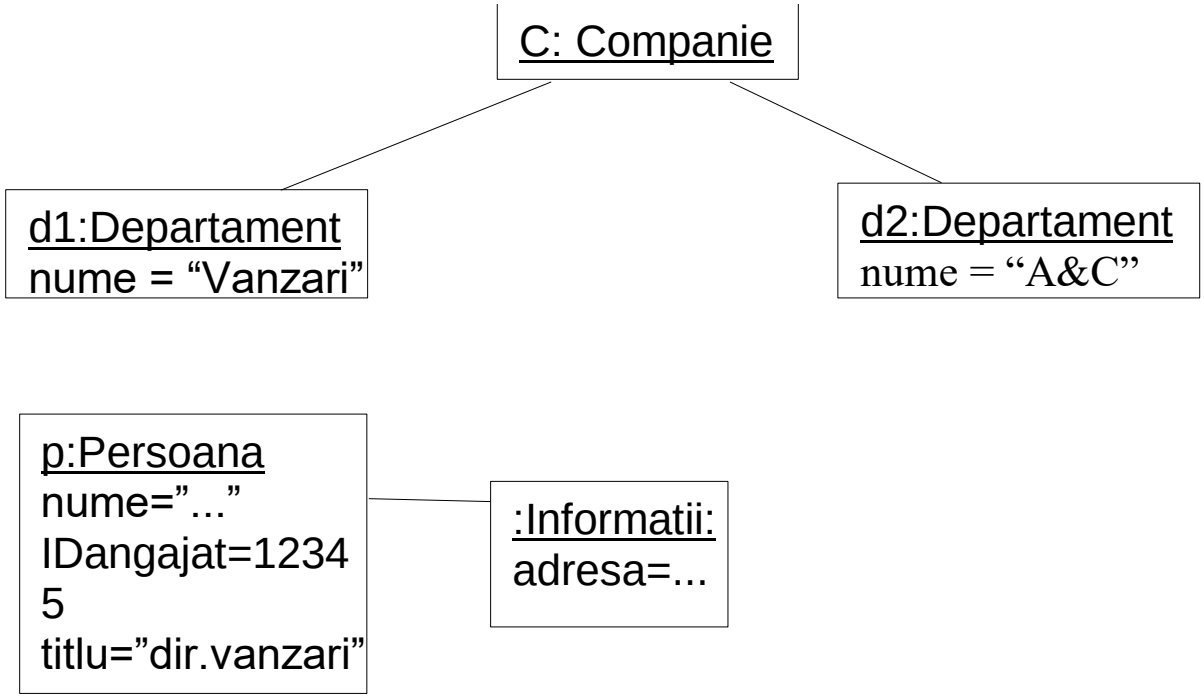


Exemplu:
de la diagrama de secventa la diagrama de clase (2)



DIAGRAME DE OBIECTE

➢ O diagrama de obiecte este o instanta a unei diagrame de clase.



O diagrama de interactiune contine in plus mesajele schimbate intre obiecte.

Modelarea interfetelor în UML

Interfeţe (1)

In Java, **definitia unei clase** poate fi separata in doua parti:

- **Interfata**: metodele publice ale clasei, exceptand constructorul si distructorul
- **Implementarea:** implementarile metodelor din interfata, constructorul, distructorul si atributele.

Avantaj: implementarea clasei este ascunsa (“information hiding”)

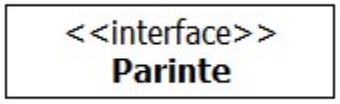
→ clientii clasei sunt fortati sa foloseasca numai interfata clasei

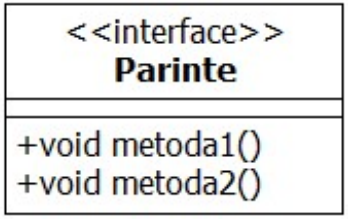
→ modificarea implementarii interfetei nu afecteaza clientii

Reprezentarea interfețelor

❖ O interfață poate fi reprezentată într-o diagramă de clase sau într-o diagramă de componente în două moduri, în funcție de nivelul de detaliu pe care îl dorim:

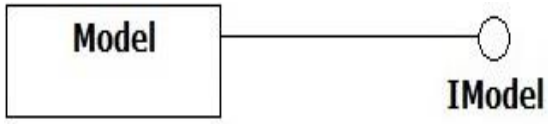
1. Reprezentare asemănătoare cu a unei clase:





Interfeţe (2)

2. Reprezentare printr-un cerc adnotat cu numele interfeței, notatie numita *lollipop* (acadea):



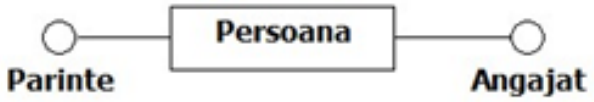
Clasa *Model* implementează interfața *IModel*.

O clasa poate sa implementeze mai multe interfeţe:

→ **o interfaţă reprezinta un rol** pe care obiectele unei clase il joaca in raport cu obiectele altei clase

→ **obiectele unei clase pot juca mai multe roluri**

Exemplu: clasa Persoana poate implementa interfeţele “Angajat” si “Parinte”.

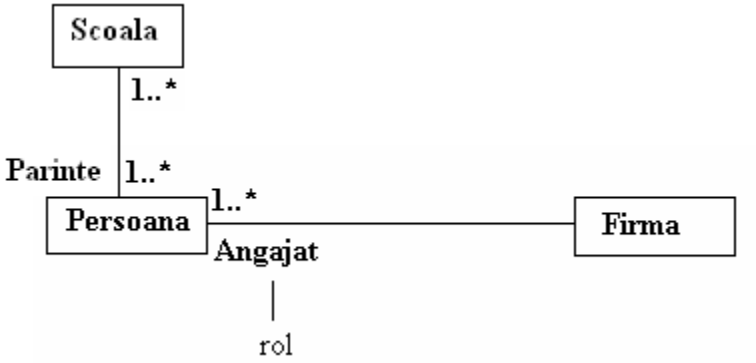


Clasa **Persoana *furnizeaza*** interfeţele **Parinte** si **Angajat**.

Interfete in Java

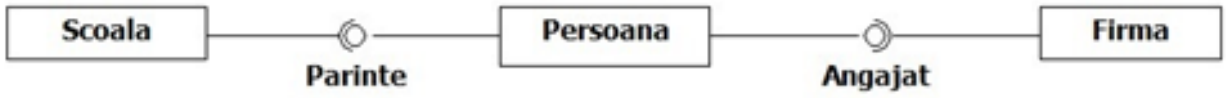
In Java:

```
interface Angajat
{  float Salariu();
   int    oreLucrate();
   String Name();
 }

interface Parinte
{ public void metoda1();
  public void metoda2();
}

class Persoana implements Angajat, Parinte
{.....................................}
```



Interfeţele definesc roluri in asocierea dintre 2 clase.

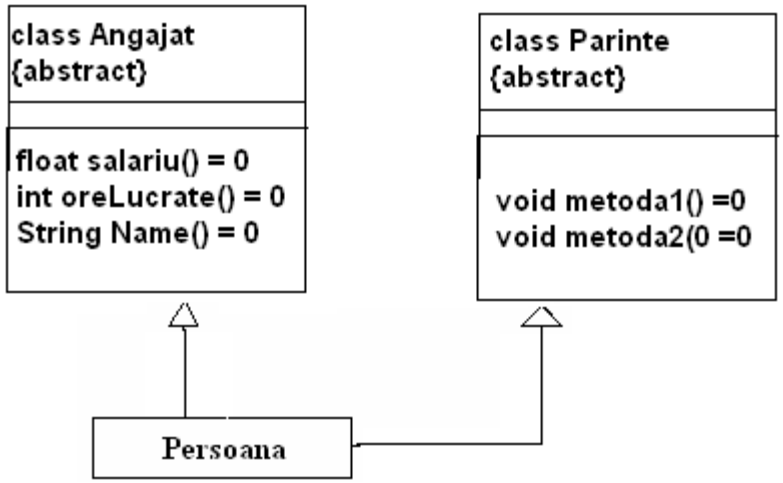
Clasa **Scoala *utilizeaza*** interfata **Parinte** iar clasa **Firma** *utilizeaza* interfata **Angajat.**



Interfete in C++

In C++ o interfata poate fi reprezentata printr-o clasa abstracta pura:

```
class Angajat // clasa abstracta pura
{ // contine numai functii virtuale pure
  // nu contine atribute
   public: virtual float Salariu()=0;
           virtual int oreLucrate()=0;
           virtual String Name()=0;
};

class Parinte // clasa abstracta pura
{ public: virtual void metoda1()=0;
           virtual void metoda2()=0;
};

class Persoana: public Angajat, Parinte
{ public: Persoana(...); // constructorul
           virtual  void ~ Persoana(); // distructorul
           virtual float Salariu();
           virtual int oreLucrate();
           virtual String Name();
           virtual void metoda1();
           virtual void metoda2();
}
```



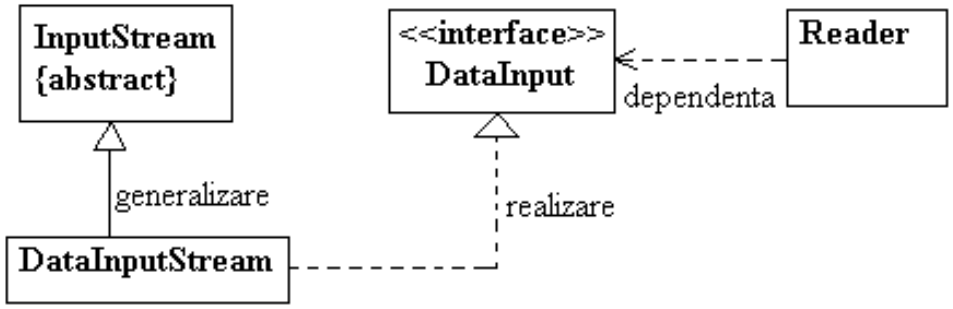
Interfeţe şi clase

In concluzie:

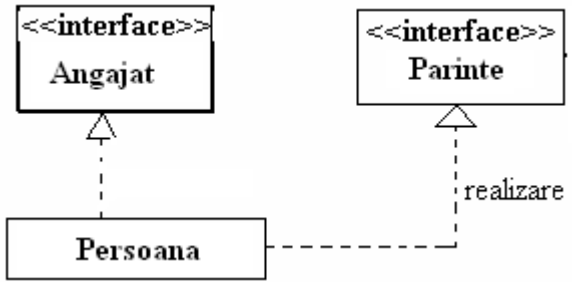
- O interfaţă este un set de metode corelate care definesc o anumita comportare.
- Toate metodele sunt publice si nu se specifica nici un fel de implementare pentru ele.
- O interfata nu are stare (nu contine variabile).

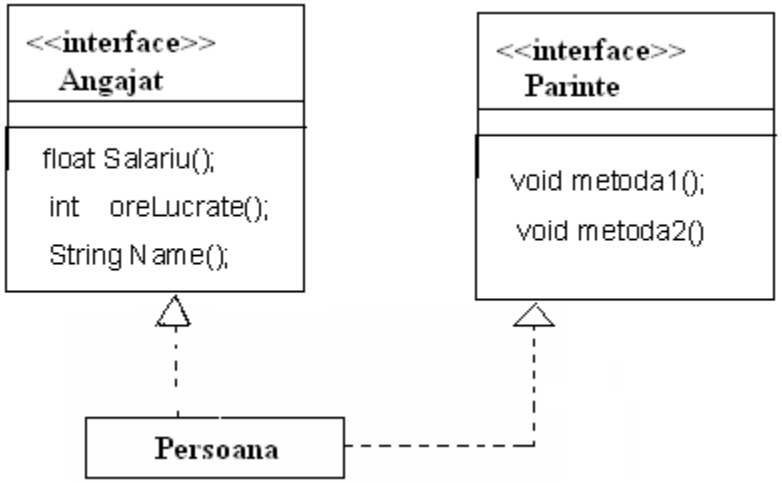
Intre interfete si clase pot fi stabilite relatii de realizare şi dependenţă:

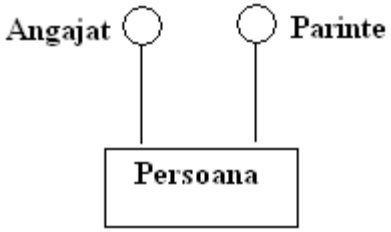
- clasa **InputStream** este abstracta.
- clasa **DataInputStream *implementeaza*** atat clasa abstracta **InputStream** cat si interfata **DataInput.**
- clasa **Reader *utilizeaza*** functiile oferite de interfata **DataInput**.

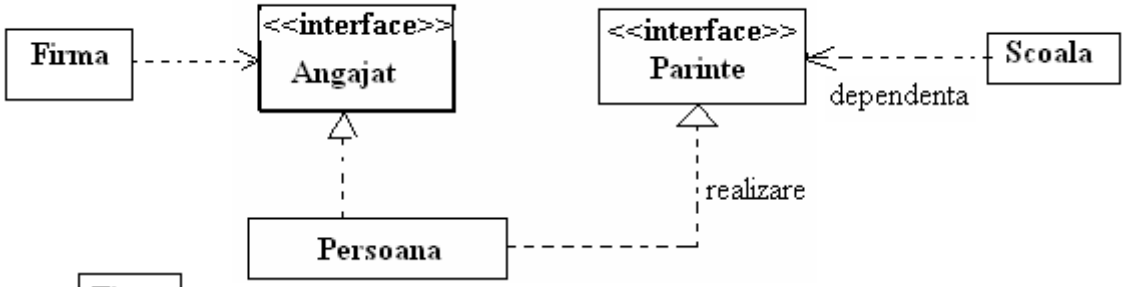


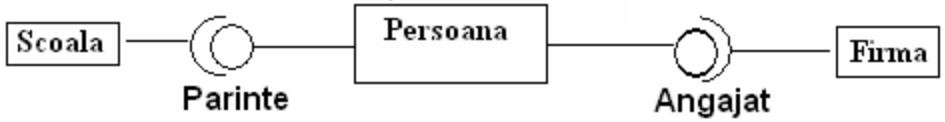
Reprezentarea interfetelor si a relatiilor dintre clase si interfate











Intelegerea unei interfeţe

Pentru a se uşura întelegerea unei interfete, se pot ataşa intefeţei:

- pre si post conditii pentru fiecare operatie
- invariant la nivelul unei clase (un predicat care trebuie sa fie adevarat pentru toate obiectele clasei)
- specificarea formala a semanticii, folosind OCL (Object Constraint Language, inclus in UML)
- se poate atasa un automat (diagrama de stari) pentru a specifica ordonarea operatiilor interfetei
- se pot atasa diagrame de colaborare pentru a specifica comportarea prevazuta pentru interfata.

Diagrame de componente

Componente(1)

In UML 1.x

O componenta este un element software fizic din componenţa unui sistem: fisier cod binar, document, fisier continand cod sursa sau date, tabela a unei baze de date.

➢ **O componenta binara este o parte fizica substituibila a unui sistem, care realizeaza si este in conformitate cu un set de interfete.**

▪ Componentele binare sunt independente de limbajul de programare in care au fost codificate iar utilizarea lor se bazeaza exclusiv pe interfete.

▪ Tehnologii folosite pentru crearea de componente binare: COM+, DCOM, CORBA, Java Beans, .NET.

Reprezentarea grafica a unei componente in UML 1.x

Componente(2)

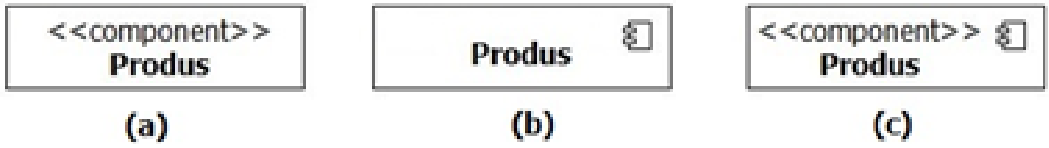
In UML 2.x

O componenta este o construcţie logica definita la proiectarea sistemului.

O implementare a unei componente poate fi uşor reutilizată sau înlocuită cu o altă implementare, deoarece componenta "încapsulează" un comportament expus prin interfeţele sale.

Componentele pot reprezenta:

- Construcţii logice care nu au un corespondent explicit la executie (de exemplu, un subsistem, o funcţionalitate la nivelul domeniului aplicaţiei, etc.)
- Construcţii logice care au un corespondent explicit la executie, de exemplu, un server de baze de date (sistemul de management al bazei de date).



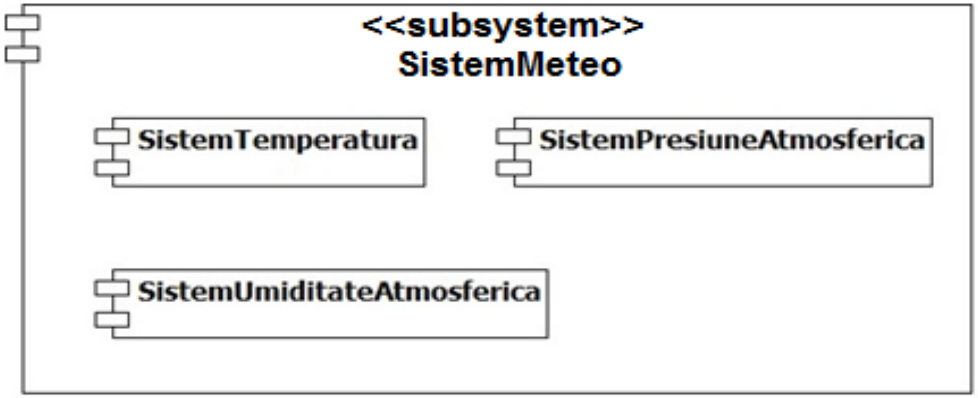
Reprezentarea grafica a unei componente in UML 2.x.

Componente(3)

Tipul unei componente poate fi precizat printr-un stereotip:

«subsystem»

- O componentă ”subsistem” reprezintă o unitate de decompoziţie a unui sistem software de dimensiune mare. Un subsistem poate grupa mai multe componente şi este parte aproape independentă a unui sistem.



«process»

- Componentă bazată pe tranzacţii.

«service»

- Componentă funcţională, fără stare.

Componente(4)

«specification»

- Se foloseşte pentru a adnota o componentă care reprezintă numai o specificaţie a unui comportament, prin interfeţele furnizate şi cele necesare. Definiţia sa nu precizează entităţile care realizează specificaţia. Acestea pot fi ataşate componentei de tip specificaţie printr-o *relaţie de realizare*.

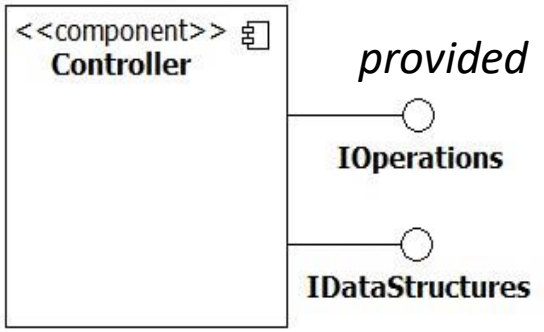
«realization»

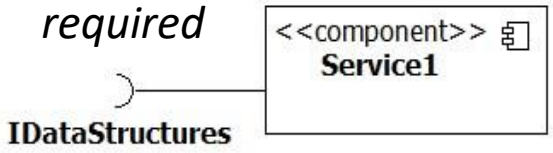
- Se foloseşte pentru a adnota o componentă care realizează (implementează) comportarea specificată printr-o altă componentă.
- Stereotipurile «specification» şi «realization» sunt utilizate pentru a modela componente cu definiţii separate pentru specificaţie şi realizare, unde o specificaţie poate avea mai multe realizări.

Interfeţe furnizate si interfeţe necesare

O interfaţă **furnizată** (*provided*) de o componentă poate fi:

- realizată (implementată) de componenta însăşi, sau
- realizată de alte componente, care *realizează* componenta.





Componenta Service1 utilizeaza (necesita) serviciile oferite de interfata IDataStructures

Componenta ***Controller*** furnizeaza interfetele ***Ioperations*** si ***IDataStructures***.

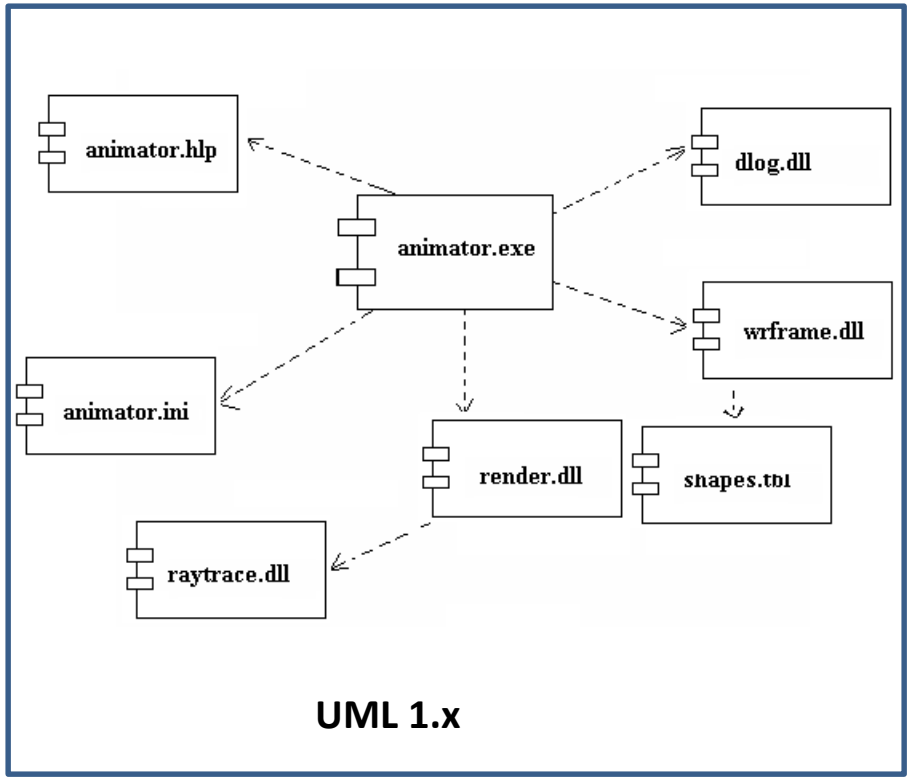
O interfaţă **necesară** (*required*) unei componente este una care conţine funcţii:

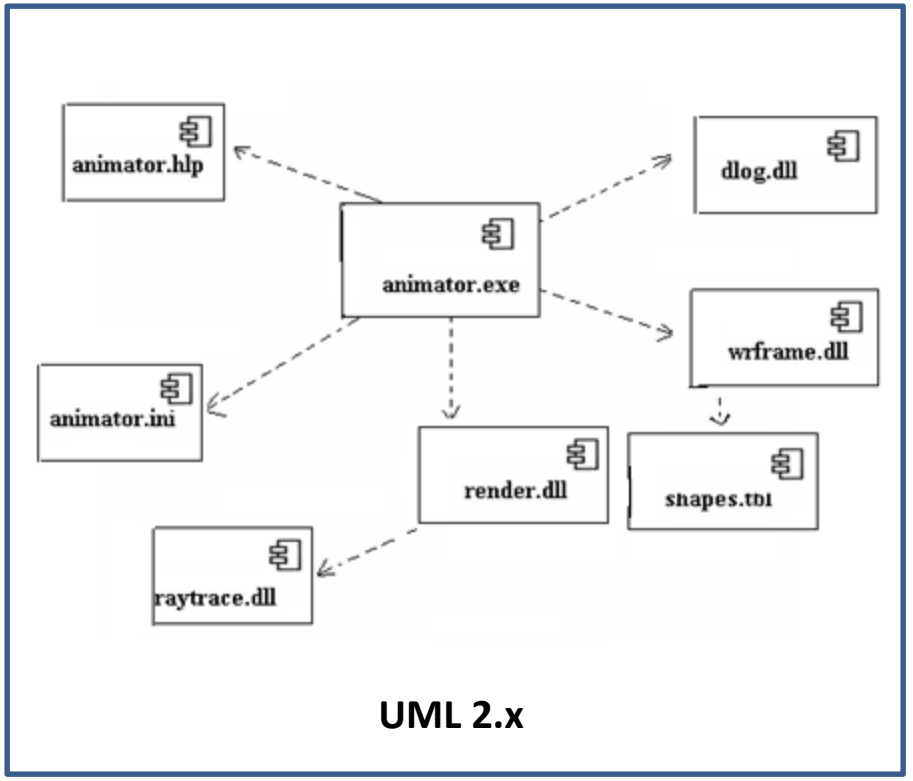
- necesare (implementării) componentei, sau

- necesare entităţilor care *realizează* componenta.

Diagrame de componente

- Redau relatiile structurale dintre componentele software ale unui sistem.



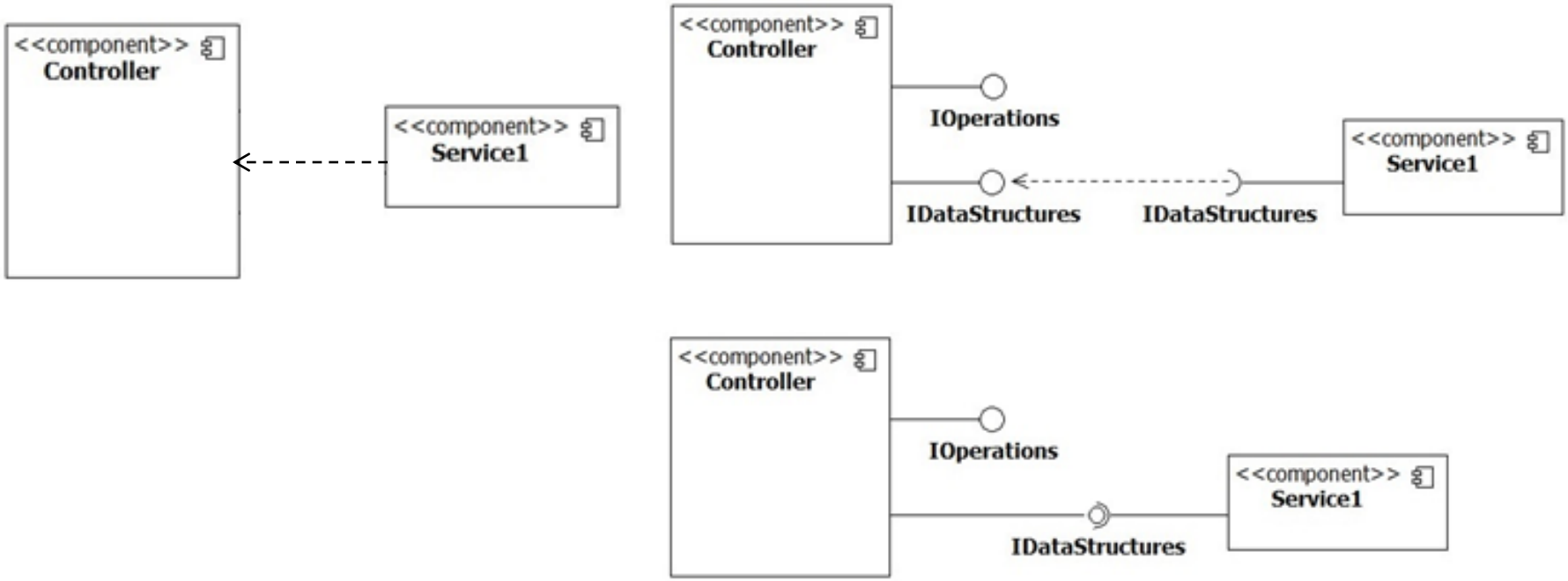


Relaţii între componente (1)

Dependența

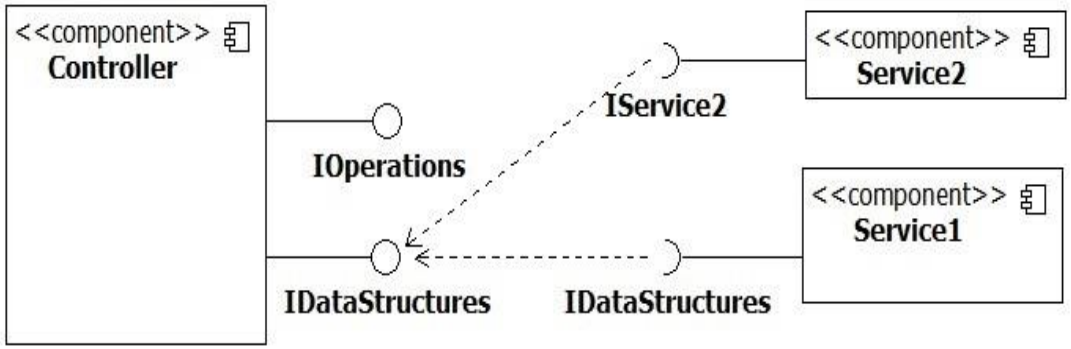
- O componentă care necesită o interfaţă *depinde* de componenta care implementează interfaţa respectivă.

Reprezentari pentru relatia de dependenţă



Relaţii între componente (2)

Daca interfaţa necesară unei componente conţine un subset din funcţiile furnizate de o interfaţă, numele interfeţei necesare poate fi diferit de cel al interfeţei care furnizează funcţiile.

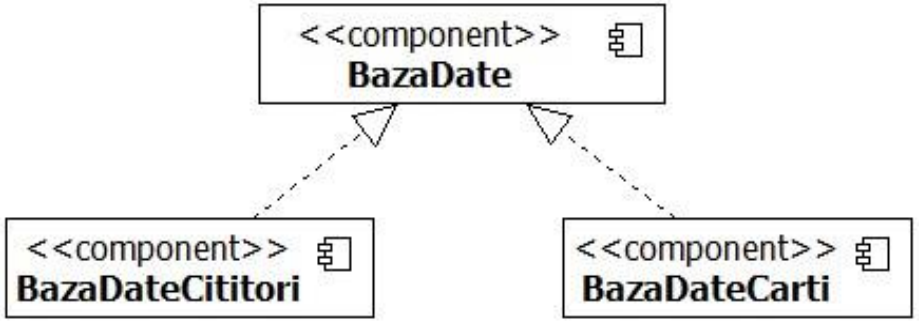


Realizarea

- O componentă poate fi doar o specificaţie a unui comportament, prin intermediul interfeţelor furnizate şi necesare.
- Comportamentul poate fi realizat (implementat) de alte componente, caz în care între componentă şi componentele care o realizează există o relaţie de realizare.

Relaţii între componente (3)

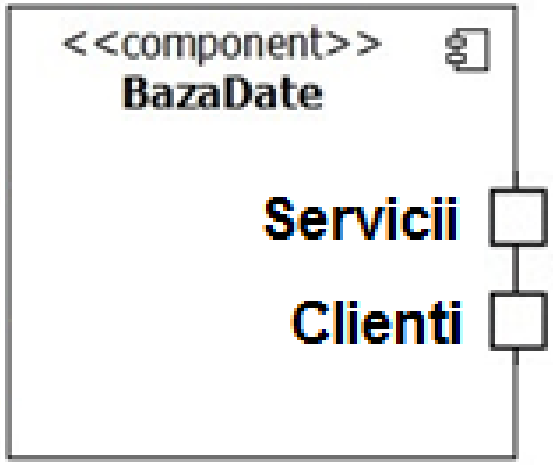
Relatia de ***realizare***



Porturi

Un port desemnează un punct de interacțiune între o componentă și mediul extern sau între părți ale componentei.

Portul are asociat un nume și se reprezintă ca un dreptunghi plasat pe una dintre laturile dreptunghiului care încadrează componenta.



Conectori (1)

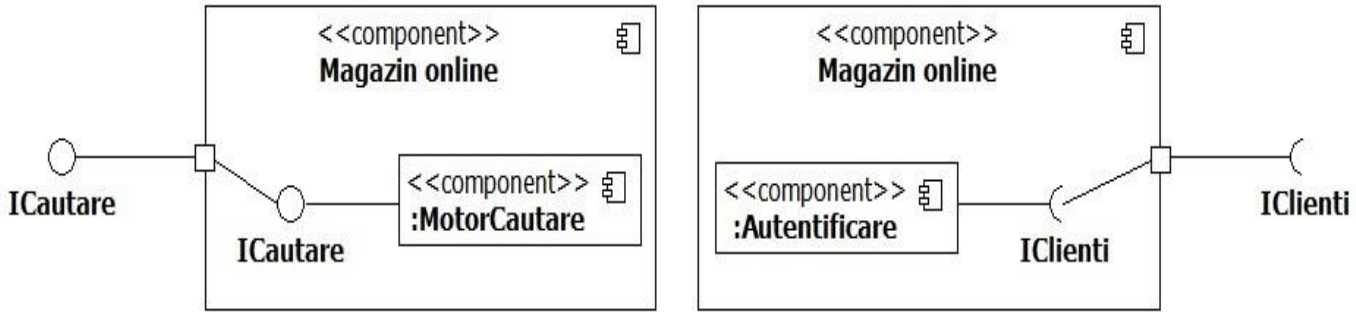
- Un conector este ***o legătură care reprezintă comunicarea dintre două sau mai multe instanţe***. Conectorii sunt de două tipuri:

  - Conectori de delegare

  - Conectori de asamblare

- Un **conector de delegare** leagă o interfață a unei componente de o subcomponentă a acesteia, care implementează sau utilizează interfaţa.

- Conectorul de delegare redirecționează semnalele primite, către sau dinspre una sau mai multe părți ale componentei.

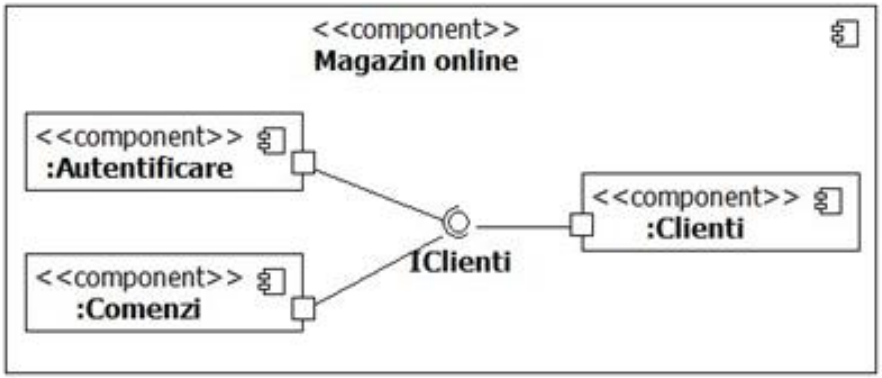


Conectori de delegare

Conectori (2)

Conectori de asamblare

Conectorii de asamblare conectează două sau mai multe subcomponente ale unei componente, indicând faptul că o subcomponentă furnizează servicii pe care alte subcomponente le folosesc.



Utilitatea diagramelor de componente

- In modelarea arhitecturală a unui sistem software.
- Furnizează o vedere arhitecturală de nivel înalt a sistemului, facilitând luarea deciziilor în privinţa asignării tascurilor.
- Permit modelarea componentelor software de nivel înalt şi a interfeţelor acestora.
- Odata definite interfeţele, se poate repartiza mult mai bine efortul de dezvoltare între subechipele care dezvolta componentele.
- Pe parcursul dezvoltării, interfeţele pot fi modificate pentru a reflecta noi cerinţe, schimbări ale cerinţelor sau ale arhitecturii produsului software.
- Sunt mijloace utile de comunicare între participanţii cheie în proiect şi echipa de implementare.

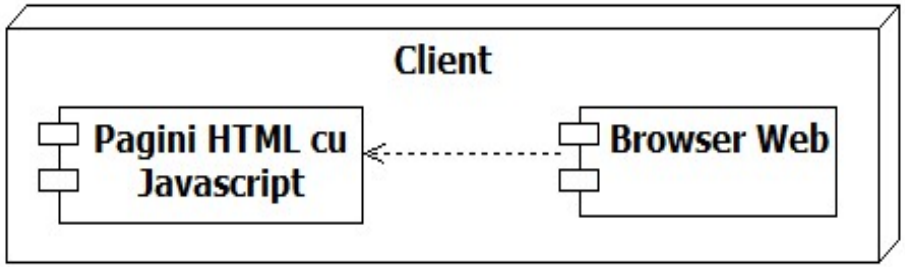
Diagrame de distributie

(Deployment diagrams)

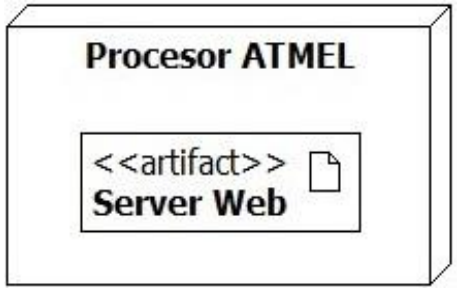
Componente, artefacte, noduri (1)

O diagramă de distribuţie reprezintă arhitectura unui sistem prin *distribuţia artefactelor software* pe echipamentele mediului de implementare, reprezentate ca *noduri (cuburi in vedere perspectiva)* în diagramele de distribuţie.

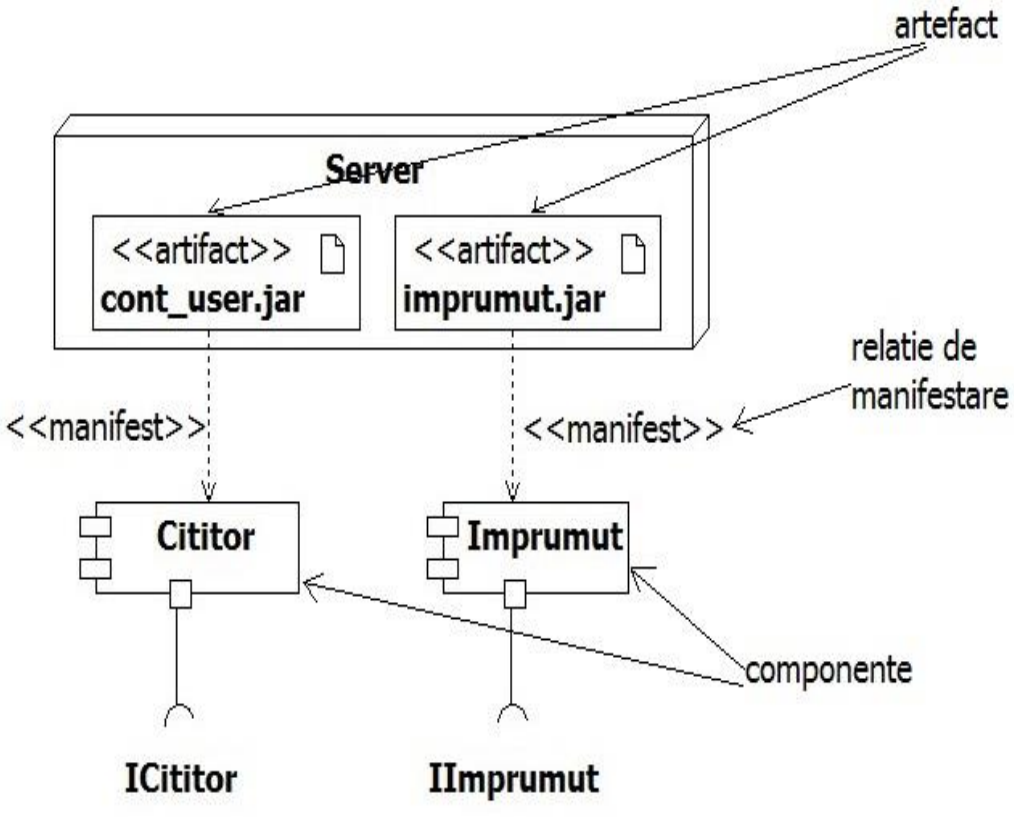
În UML 1.x, componentele sunt distribuite direct pe noduri, ele fiind entităţi fizice din componenţa unui sistem.



În UML 2.x, componentele sunt distribuite indirect pe noduri, prin implementări (*manifestari*) ale lor, numite *artefacte*.



Componente, artefacte, noduri (2)



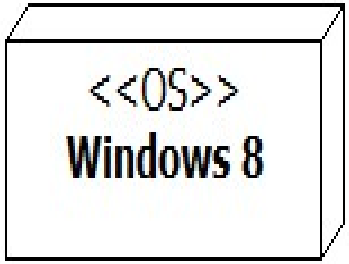
Noduri

- Nodurile din diagramele de distribuție reprezintă resurse computaționale. Acestea pot fi:
  - **Nod de tip dispozitiv**, reprezentând un echipament hardware
  - **Nod de tip mediu de execuție**, care reprezintă o resursă software ce rulează pe un echipament hardware și este un mediu pentru execuția altor elemente software.
- Cele doua tipuri de noduri pot fi marcate prin stereotipurile standard

  <<device>>

  <<executionEnvironment>>

sau alte stereotipuri nestandard: *<<computer>>, <<cd-rom>>, <<server>>, <<pc>>, <<client>>*, <<application server>>, <<OS>>, <<database system>>, <<web server>>.



Noduri si artefacte

Nodurile <<device>> pot fi reprezentate şi prin simboluri grafice specifice:

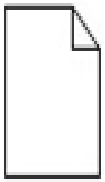
<<Server aplicatie>>
IBM system z

<<mobile device>>
smartphone

Artefactele sunt entităţi fizice (efective) obținute printr-un proces de dezvoltare software: fişiere executabile, biblioteci, fişiere arhivă, documente,baze de date, etc.

Ele **se execută sau sunt stocate pe noduri.**

Reprezentarea artefactelor

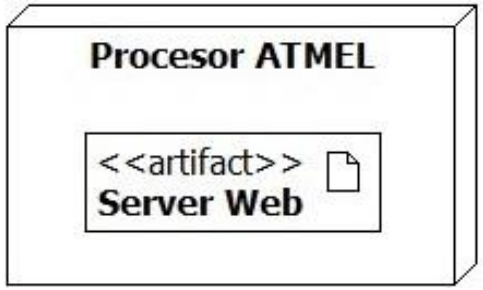


BDComenzi

Distributia artefactelor pe noduri

Sunt 2 metode de a reprezenta distributia unui artefact pe un nod:

- amplasarea artefactului pe suprafața nodului

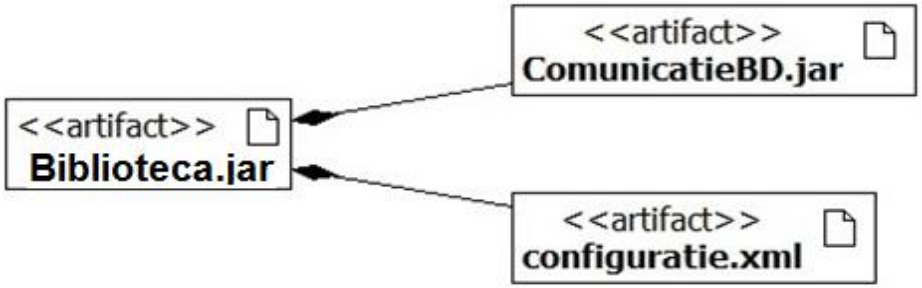


- reprezentarea unei relatii de distributie de la artefact catre nod

- folosind un specificator de distributie (a se vedea “UML practic”, Ed MatrixRom, 2014)

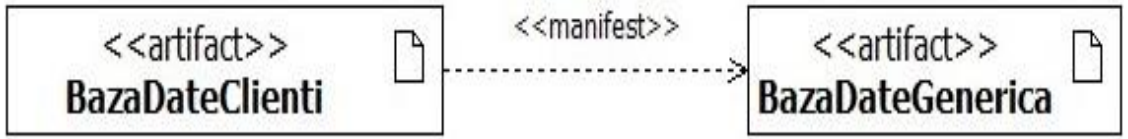
Relatii intre artefacte

- Între artefacte se pot stabili relații de asociere, dependență şi manifestare.
- O ***asociere*** între două artefacte poate fi de tip ***compunere sau agregare*** – cu aceeasi semnificatie ca relatiile corespunzatoare dintre clase.



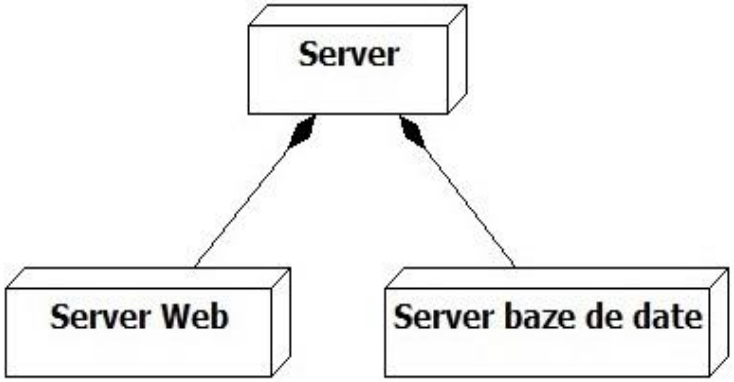
- ***Dependenta***

- ***Manifestarea:*** BazaDateClienti implementeaza BazaDateGenerica



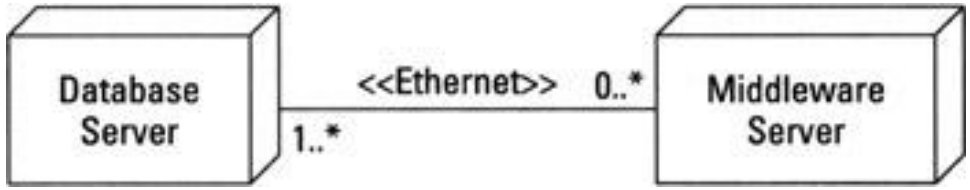
Relatii între noduri (1)

Compunerea



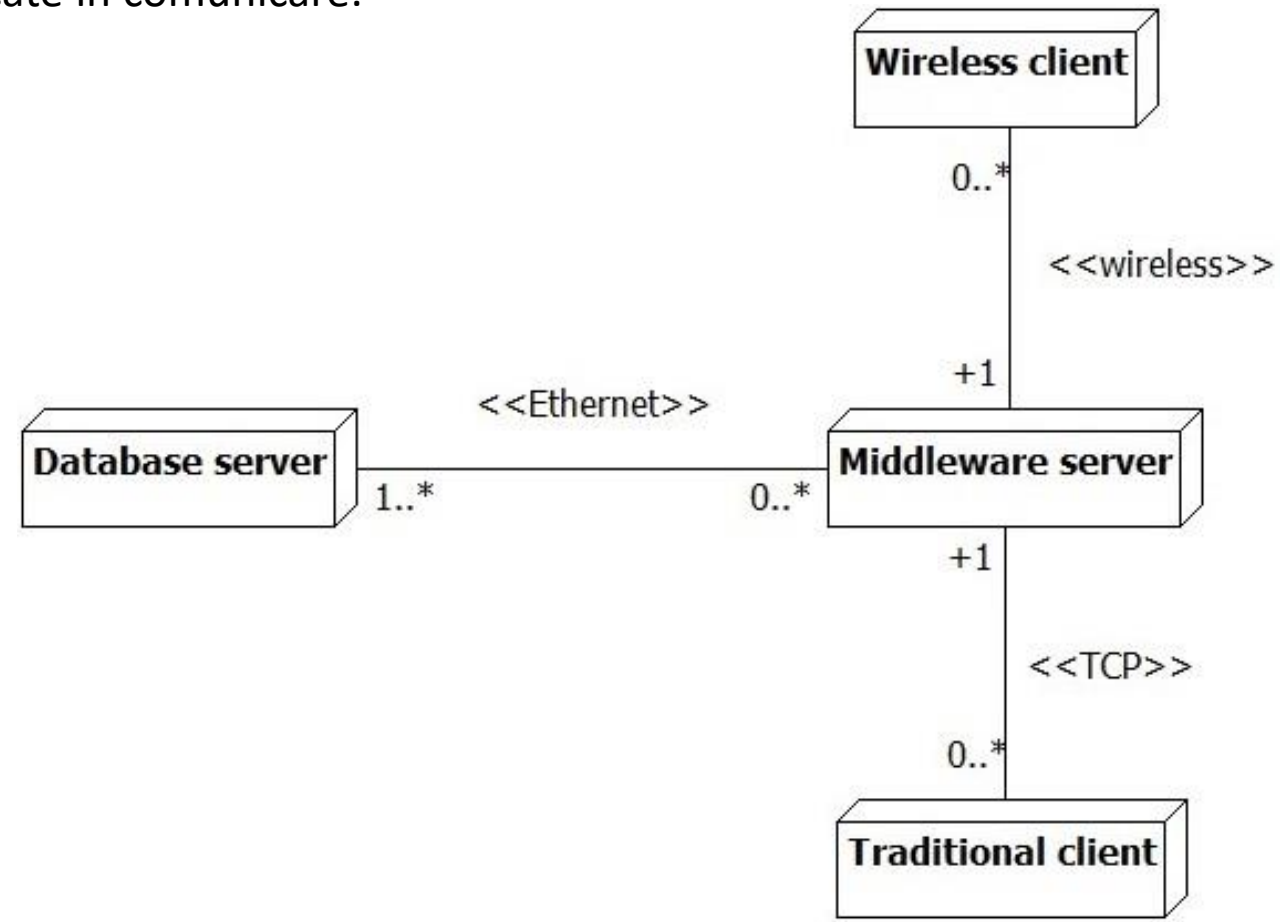
Asocierea

- Reprezintă o cale de comunicare între două noduri și se notează ca şi relaţia de asociere dintre clase.
- Pentru a reprezenta modul în care comunică nodurile se folosesc, ca nume de asociere, stereotipuri care indică modul de comunicare: <<TCP>>, <<UDP>>, <<Ethernet>>, etc.

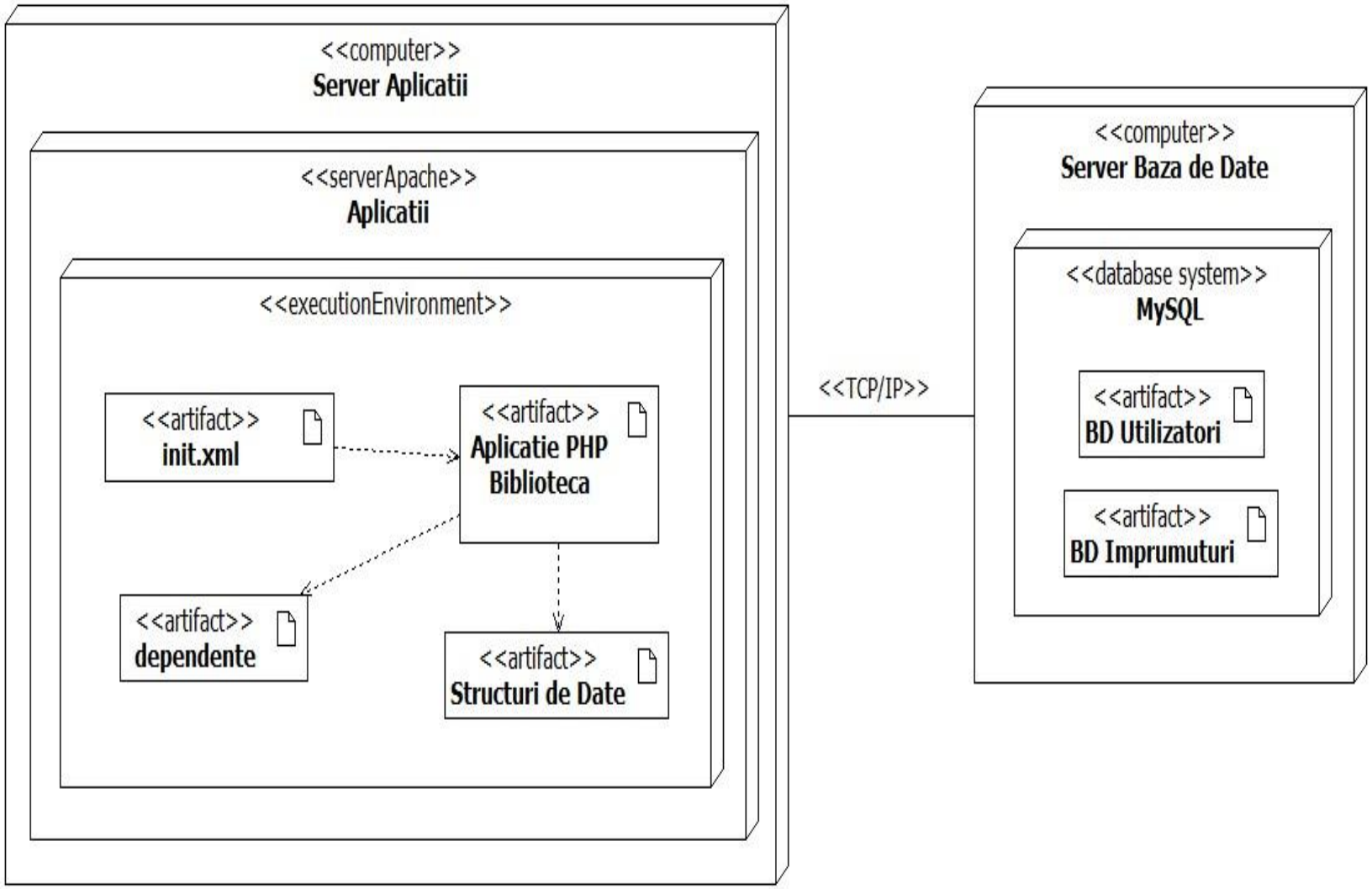


Relatii între noduri (2)

- Capetelor asocierii le pot fi atribuite multiplicităţi, reprezentând numărul de instanțe ce pot fi implicate în comunicare.



Diagramă de distributie in UML 2.x



Diagrame pentru managementul modelelor (Packages)

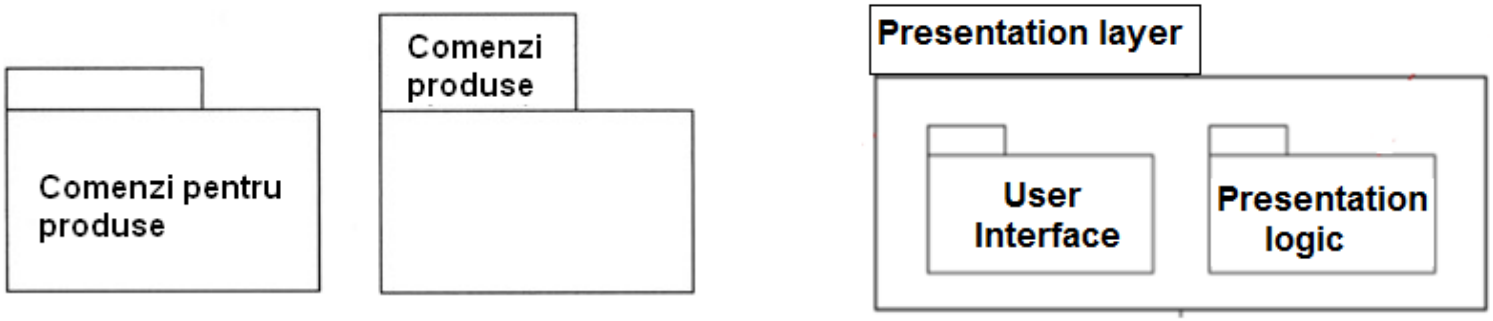
Prof. univ. dr. ing. Florica Moldoveanu

Pachete (1)

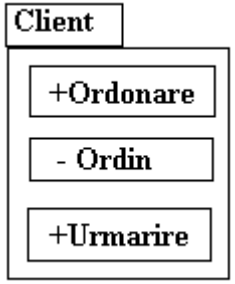
- Se folosesc pentru organizarea artefactelor rezultate in procesul de dezvoltare al unui sistem software.
- Un pachet are exact aceeasi functionalitate ca si un director pe disc (folder).
- Un pachet contine elemente de modelare corelate logic: cazuri de utilizare, diagrame de interactiune asociate cazurilor de utilizare, diagrame de clase si diagrame de componente care implementeaza cazurile de utilizare, diagrame de activitate, clasele care formeaza un subsistem, etc.
- Un pachet poate contine alte pachete, deci se poate crea o ierarhie de pachete.
  - Instrumentele de modelare UML afiseaza grafic gruparea elementelor de modelare in pachete printr-un arbore similar cu cel afisat de Windows Explorer.
- Pachetele sunt containere. Daca se sterge sau muta un pachet, toate elementele din interiorul său sunt sterse/mutate.

Pachete (2)

- Un pachet este reprezentat printr-un simbol de “folder”. Numele pachetului poate fi amplasat in centrul simbolului sau in coltul stanga sus:



- Un pachet are asociat un „spatiu de nume” (namespace): elementele amplasate in acelasi pachet trebuie sa aiba nume unice; elemente din pachete diferite pot avea acelasi nume.
- Elementele unui pachet pot fi marcate cu atributele de vizibilitate **public (+)** sau **private (-):**



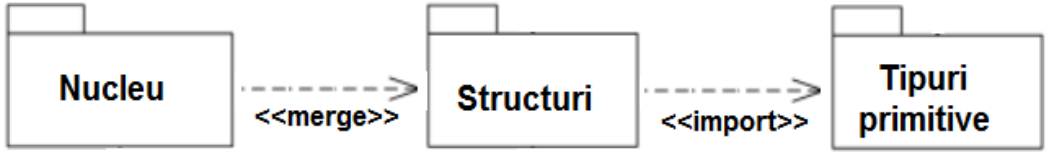
➢ Elementele publice ale unui pachet sunt întotdeauna accesibile din afara pachetului folosind ***nume calificate***, de forma:

Nume_pachet::nume_element
Nume_model::nume_pachet::nume_clasa::nume_operatie

Relatii intre pachete (1)

Extindere şi import

- Un pachet poate importa fie membrii individuali ai altor pachete fie alte pachete în întregime.
- Un pachet poate fi extins cu un alt pachet.
- Relaţiile de import şi extindere (merge) sunt relaţii de dependenţă între pachete.



- Pachetul „Nucleu” este extins cu pachetul „Structuri”.
- Pachetul “Tipuri primitive” este importat de pachetul “Structuri”, ceea ce permite referirea la elementele pachetului “Tipuri primitive” din pachetul “Structuri” utilizând nume necalificate: spaţiul de nume al pachetului “Tipuri primitive” este adăugat la spaţiul de nume al pachetului “Structuri”.

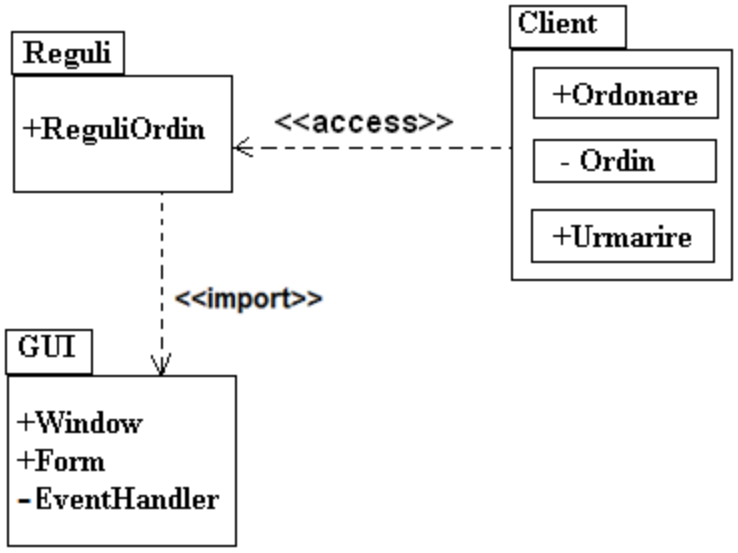
Relatii intre pachete (2)

- Importul elementelor unui pachet poate fi public sau privat:

  - public - elementele din pachetul importat isi pastreaza atributul de vizibilitate în cadrul spaţiului de nume al pachetului care importă;

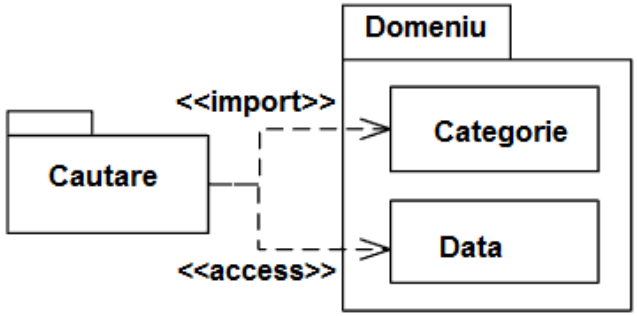
  - privat - elementele pachetului importat sunt accesibile numai în pachetul care-l importă.

- Vizibilitatea importului este definită prin stereotipul ataşat relaţiei de import: **<<import>>** pentru public, respectiv **<<access>>** pentru privat.



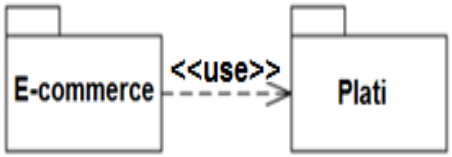
Relatii intre pachete (3)

- **Importul unor membri individuali** ai unui pachet se reprezintă printr-o săgeată care punctează către membrii importaţi.
- Membrii importaţi sunt adăugaţi la spaţiul de nume al pachetului care importă.



Vizibilitatea importului unui membru al unui pachet poate fi de tip **public** sau **privat**, cu aceeasi semnificaţie ca în cazul importului întregului pachet.

Relaţia de dependenţă intre pachete



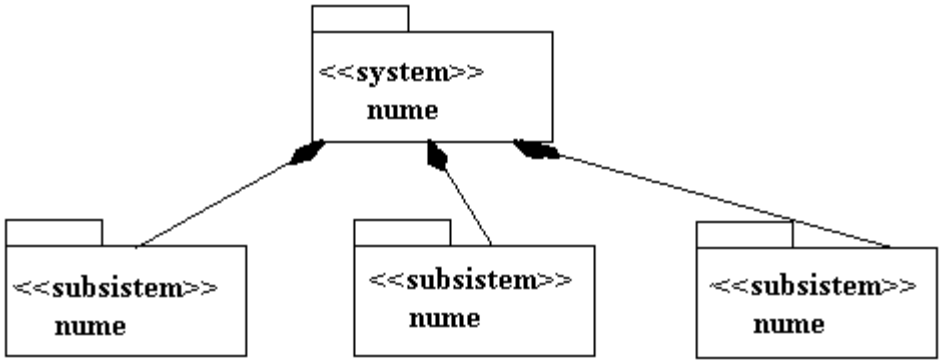
Relaţia <<use>> nu precizează cum elementul client (E-commerce) utilizează furnizorul (Plati). Astfel, pachetul „Plăţi” ar putea fi utilizat în definiţia sau în implementarea pachetului „E-commerce”.

Sisteme si Modele(1)

- Un sistem este o grupare de elemente organizata pentru realizarea unui scop.
- Sistemele pot fi descompuse in subsisteme separate, fiecare subsistem putand fi vazut la randul sau ca un sistem, la un nivel de detaliu mai coborat.
- Subsistemele sunt grupari de elemente care pot fi dezvoltate relativ independent. Descompunerea are loc in etapa de proiectare arhitecturala, cand se decide cum vor fi realizate cerintele.
- Relatiile sistem-subsisteme pot fi reprezentate:
  - In UML 1.x, printr-o diagrama de pachete
  - In UML 2.x, printr-o diagrama de componente

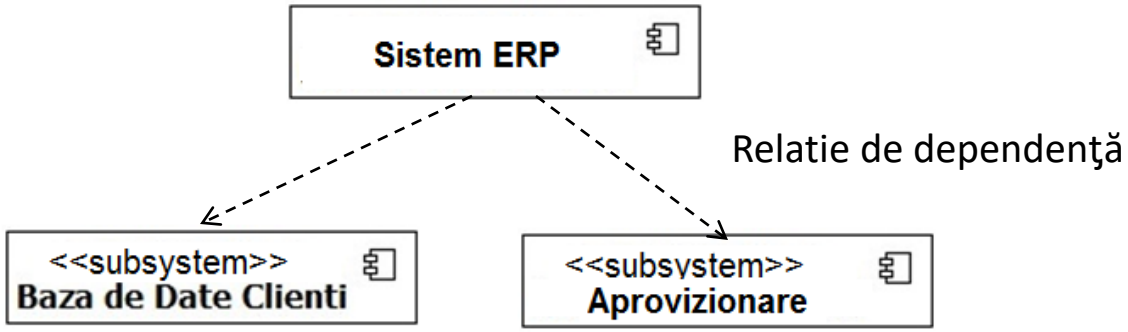
Sisteme si Modele (2)

- In UML 1.x, un subsistem este un tip particular de pachet. Se reprezinta ca un pachet cu stereotipul <<subsistem>>.



“ un sistem este compus din mai multe subsisteme”

- In UML 2.x, un subsistem este un tip particular de componenta



Sisteme si Modele (3)

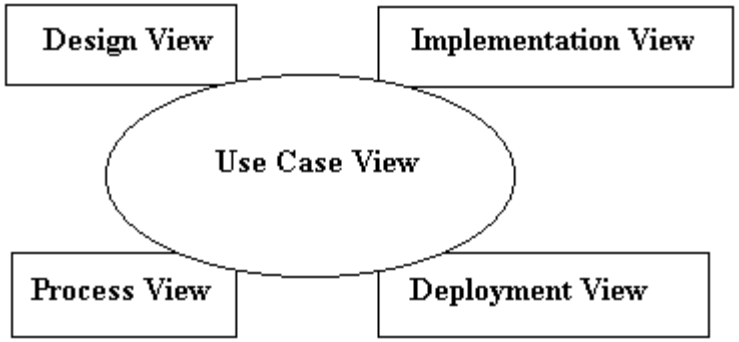
➢ Modelul unui sistem este o simplificare a realitatii, o abstractie a sistemului, creata pentru a intelege mai bine sistemul si a-l specifica.

➢ O **vedere** este o proiectie a unui model dintr-un punct de vedere in care sunt omise aspectele nerelevante din punctul de vedere respectiv.

➢ Exemplu: cele 5 vederi ale unei arhitecturi software

Use Case View cuprinde:

- Cazurile de utilizare care descriu comportarea sistemului vazuta de utilizatori, analisti si testeri,
- Diagrame de cazuri de utilizare,
- Diagrame de interactiune, de stari si de activitati.



Sisteme si Modele (4)

Design View

Cuprinde:

- clasele, interfetele - pentru modelarea aspectelor statice
- diagrame de interactiune, de stari si de activitate - pentru modelarea aspectelor dinamice

Process View

Descrie procesele si firele de executie care formeaza mecanismele de concurenta si de sincronizare ale sistemului.

- cuprinde aceleasi tipuri de diagrame dar cu accent pe clasele si obiectele care reprezinta fire si procese de executie.

Implementation View

Descrie componentele care alcatuiesc sistemul fizic final. Se folosesc:

- diagrame de componente pentru a modela aspectele statice
- diagrame de interactiune, de stari si activitate, pentru modelarea aspectelor dinamice.

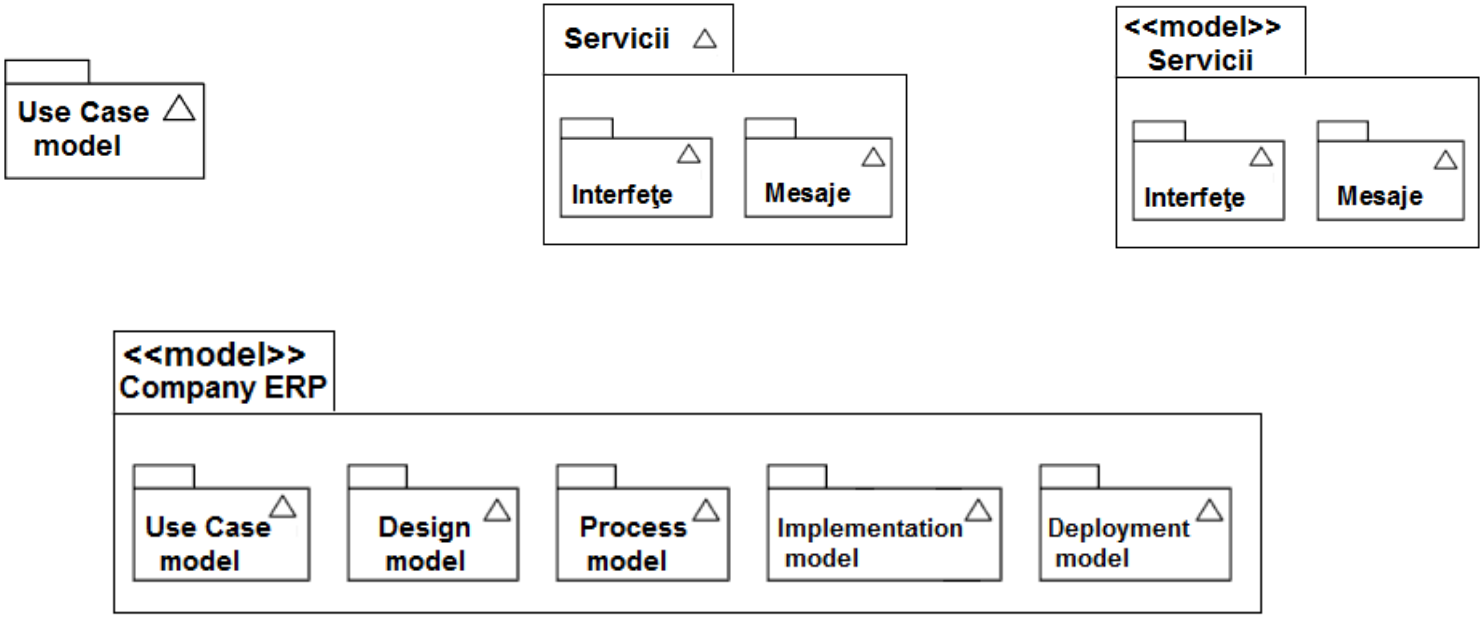
Deployment View

Cuprinde nodurile care formeaza topologia hardware a sistemului. Se folosesc:

- diagrame de distributie, pentru modelarea aspectelor statice
- diagrame de interactiune, de stari si de activitate, pentru modelarea aspectelor dinamice.

Sisteme si Modele (5)

- **Elementele de modelare care constituie o vedere asupra unui sistem pot fi grupate intr-un *model*.**
- Un model se reprezinta printr-un pachet marcat cu stereotipul <<model>> sau unul care include simbolul triunghi:



PROIECTAREA

ARHITECTURALĂ

PROIECTAREA ARHITECTURALĂ

- Modelul de analiză = vedere externă asupra sistemului (problema de rezolvat)
- Modelul de proiectare = vedere internă (soluția)

PROIECTAREA = CONCEPȚIA SI DESCRIEREA SOLUȚIEI OPTIME

Proiectarea arhitecturală = obținerea (, plecând de la modelul de analiză, a) unui model de nivel înalt de realizare efectiva a sistemului**.**

Cerințele din URD/SRD se transformă în (sunt acoperite de) componente conceptuale de nivel înalt ale viitorului sistem.

Principalele activități:

- Stabilirea criteriilor de proiectare
- Descompunea in subsisteme
- Alegerea strategiilor/șabloanelor arhitecturale și materializarea lor în arhitectură
- Tratarea condițiilor limita.

De la modelul de analiză la modelul de proiectare

Cerințe nefuncționale → Criteriile proiectării

Modelul funcțional → Descompunerea în subsisteme

Modelul obiect → Distribuția subsistemelor pe hardware, managementul datelor

Modelul dinamic → Fluxul controlului, activități concurente

Modelul de proiectare arhitecturală cuprinde:

- **Criteriile proiectării:** - calități ale sistemului ce trebuie urmărite în dezvoltare
  - restricții manageriale.

  Vor fi avute permanent în vedere în dezvoltare (proiectare, implementare), ghidând deciziile când sunt necesare alegeri, compromisuri.

- **Arhitectura software: <u>prezintă</u> următoarele aspecte; <u>justifică</u> alegerile, deciziile**
  - ✓ descompunerea sistemului in subsisteme
  - ✓ responsabilitățile subsistemelor
  - ✓ dependentele intre subsisteme
  - ✓ alocarea subsistemelor pe hardware
  - ✓ fluxul controlului
  - ✓ controlul accesului la sistem
  - ✓ stocarea datelor persistente
  - ✓ Condițiile limită: configurarea, pornirea, inițializarea, oprirea, tratarea excepțiilor

Documentul de proiectare arhitecturala
Arhitectural Design Document (ADD)

Introducere

- Scopul sistemului
- Criterii de proiectare, prioritizare, compromisuri
- Definitii, acronime, abrevieri
- Referinte la alte documente (e.g. URD, SRD, alte sisteme)

[Arhitecturi de referinta]

- a sistemului existent *(daca noul sistem inlocuieste unul existent)*
- ale unor sisteme similare

Arhitectura (unde au fost alternative/luate decizii, se vor documenta si justifica)

- Prezentare generala a arhitecturii *(identificarea sumara a subsistemelor si functiilor lor)*
- Decompozitia in subsisteme; responsabilitatile fiecarui subsistem; colaborari
- Distributia subsistemelor pe echipamente hardware
- Managementul datelor persistente *(datele persistente, SGBD, schema conceptuala a BD)*
- Controlul accesului utilizatorilor la sistem
- Fluxul global al controlului
- Conditiile limita

Glosar de termeni

STABILIREA CRITERIILOR PROIECTĂRII

Criteriile proiectării sunt derivate din cerințele nefuncționale.

e.g. timp de răspuns scurt, capacitatea sistemului de a executa un număr mare de taskuri, ușurința în utilizare, cost scăzut de mentenanță etc.

Criteriile ghidează stilul general de proiectare, precum **și alegerea** între variante arhitecturale.

Ulterior, trebuie permanent urmărite în dezvoltare (pr. detaliată, implementare).

Categorii de criterii de proiectare :

- performanta
- încrederea în sistem
- costul
- mentenanța
- criteriile utilizatorului final

CRITERII DE PROIECTARE (1)

- **PERFORMANȚĂ**
  - **Timp de răspuns**: cât de repede răspunde sistemul la cererile utilizatorului
  - **Throughput**: câte taskuri poate realiza într-o perioadă de timp fixă
  - **Resurse necesare:** **memorie, procesor, transfer de date** etc.
- **ÎNCREDERE** (Dependability): Ghidează efortul alocat pentru a minimiza căderile sistemului și consecințele lor:

Fiabilitatea	Reliability	Gradul în care realizează constant comportamentul specificat (calculat pentru un interval de timp dat / o anumită cantitate de date etc.)
Robustețea	Robustness	Capacitatea de a rezista la intrari nevalide
Disponibilitatea	Availability	Procentul de timp in care sistemul poate fi folosit
Toleranța la defecte	Fault tolerance	Capacitatea de a functiona in conditii de eroare
Securitatea	Security	Capacitatea de a rezista la atacuri
Siguranța	Safety	Capacitatea de a nu pune in pericol vieti omenesti chiar si in prezenta erorilor de operare sau a caderilor

CRITERII DE PROIECTARE (2)

- **COSTURI:**
  - Dezvoltarea sistemului inițial
  - Tranziția la utilizatori (Deployment: instalarea sistemului și instruirea utilizatorilor)
  - Avansul de la un eventual sistem existent:
    - costul modificării sistemului existent pentru a obține funcționalitatea dorită

    vs.
    - costul asigurării compatibilității cu sistemul înlocuit (backward compatibility)
  - Mentenanța (adăugări de funcții, reparare defecte – în timp)
  - Operare (inclusiv administrare)

COSTUL DEZVOLTĂRII ESTE IN OPOZIȚIE CU TOATE CELELALTE COSTURI

CRITERII DE PROIECTARE (3)

- **MENTENABILITATE:** ușurința de a modifica sistemul dupa livrare.
  - Extensibilitatea: ușurința de a adăuga noi funcționalități
  - Modificabilitatea: ușurința de modificare a unor funcții
  - Adaptabilitatea: ușurința de a adapta sistemul la alte domenii
  - Portabilitatea: ușurința de a porta sistemul pe alte platforme
  - Claritatea codului: ușurința de a înțelege sistemul prin citirea codului sursă
  - Trasabilitatea cerințelor: ușurința de a face corespondența între codul sursa și cerințe specifice.
- **CRITERII ALE UTILIZATORULUI FINAL**
  - Utilitatea: cât de mult îl va ajuta sistemul pe utilizator în munca lui
  - Ușurința de utilizare: cât de ușor este pentru utilizator sa folosească sistemul

PRIORITIZAREA CRITERIILOR, COMPROMISURI

Toate criteriile sunt dezirabile, însă nu toate pot fi realizate simultan.

e.g. fiabilitatea si siguranța vs. cost de dezvoltare redus.

Se prioritizează criteriile. Se decid cele esențiale și cele la care se renunță.

Se decid compromisuri prin care se ating parțial mai multe criterii. 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.
- **Costuri, 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.
- **Costuri, Timp de livrare <–> Calitate**: daca sistemul nu poate fi realizat la calitatea cerută în limitele de timp și cost, poate fi livrat la timp dar cu defecte acceptate, sau poate fi livrat mai târziu dar cu mai puține defecte.
- **Timp de livrare <–> Costuri**: calitatea și cantitatea resursei umane necesare.

PROIECTAREA ARHITECTURII

- Arhitectura rezulta printr-un **proces iterativ** de **descompunere a cerințelor funcționale si alocarea lor unor subsisteme ale viitorului sistem**.
- Scopul descompunerii: reducerea complexității (“divide et impera”).

- Un **subsistem:**
  - este o parte substituibila a unui sistem, cu interfețe bine definite, ce încapsulează stare și/sau comportament;
  - rezolva sau participă la rezolvarea unui subset de cerințe

Subsisteme relativ independente **===> echipe independente le pot dezvolta in paralel**

- In cazul sistemelor complexe, descompunerea se aplica în mod recursiv, descompunând subsistemele în subsisteme mai simple, pana la un grad de complexitate potrivit

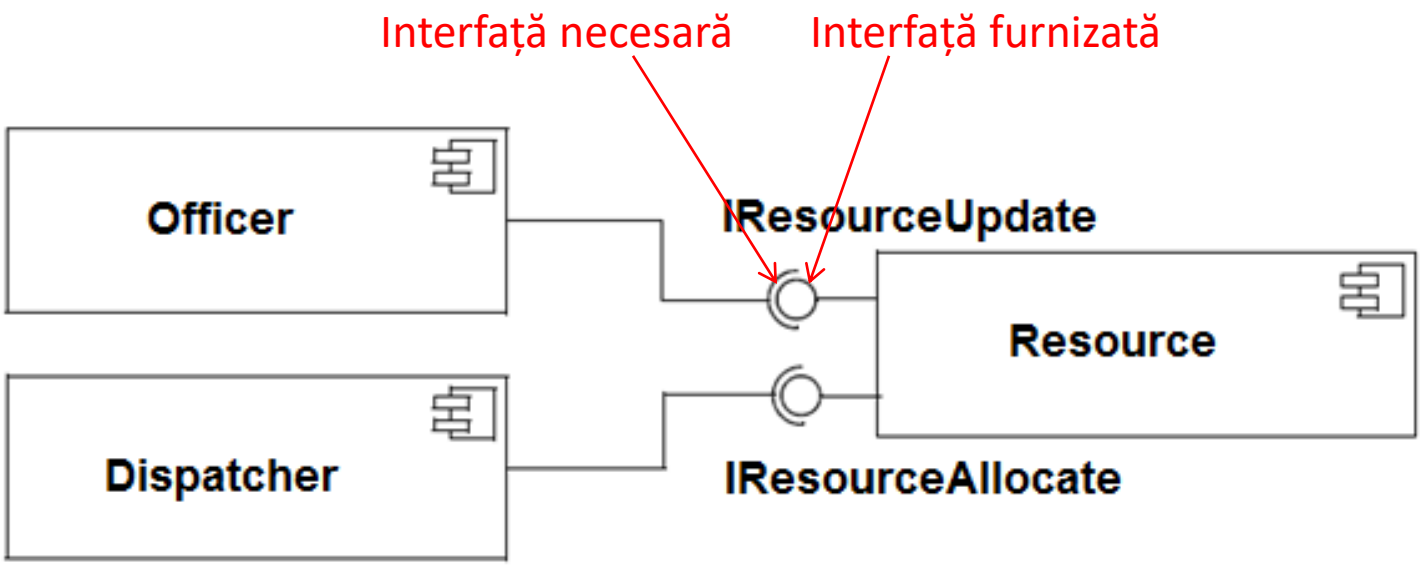
SERVICII ŞI INTERFEȚE DE SUBSISTEME

- Un subsistem se caracterizează prin **serviciile oferite** altor subsisteme.

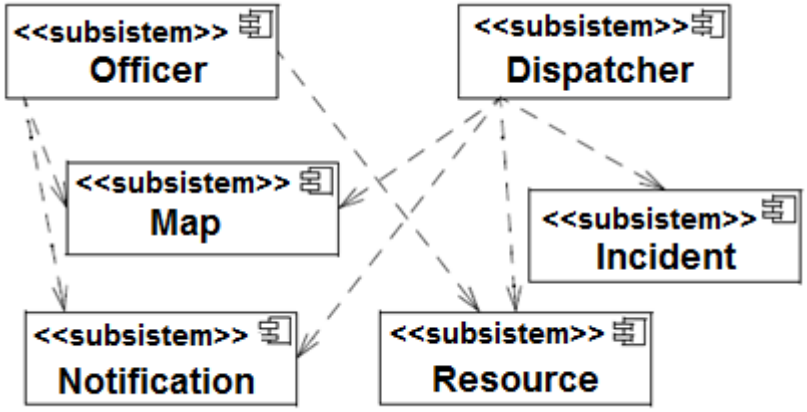
  serviciu = set de operații corelate, cu scop comun

- **Setul de operații oferite de un subsistem formează interfața/interfețele subsistemului**, confinând: numele operațiilor, parametrii și tipul parametrilor, valorile întoarse.

- Proiectarea include concepția și definirea serviciilor pe care le oferă si utilizează fiecare subsistem, cvasi-independent



DESCOMPUNEREA CERINTELOR FUNCTIONALE PE SUBSISTEME



Intre subsistemele rezultate din descompunere se stabilesc relatii de dependenţă.

Subsistemele trebuie documentate clar deoarece sunt dezvoltate de echipe diferite.

- Unele limbaje de programare, cum ar fi Java, furnizeaza constructii pentru modelarea subsistemelor
- In alte limbaje (C, C++), subsistemele nu sunt explicit modelate, dar ele pot fi grupate intr-un pachet UML

PRINCIPII ȘI STRATEGII DE DESCOMPUNERE

- **Subsistemele trebuie sa fie cat mai independente unul de altul**:
  - Un subsistem trebuie sa poata fi inteles ca o entitate de sine-statatoare
  - 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
  - Efectul unei erori trebuie izolat (dacă e posibil) în subsistemul care a generat-o

    Alternativ/complementar: sistem global, unitar de tratare a erorilor
- **Subsistemele trebuie sa "ascundă" conținutul lor**: se separă interfața de implementare

❖ **Strategii de descompunere:**

- **Descompunere ierarhică** în niveluri de abstractizare sau module subordonate (desc. pe verticală)
- **Partiționare** în module de același nivel de abstractizare (descompunere pe orizontala)

CUPLARE ŞI COEZIUNE

Cuplarea a doua subsisteme este data de dependenţele 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** => minimizează impactul pe care schimbarile într-un subsistem îl au 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 = numarul si natura dependenţelor dintre elementele sale.

- O proprietate dorita a proiectarii de sistem este obtinerea de subsisteme cu coeziune mare.

❖ **In general trebuie facut un compromis între coeziune si cuplare**.

Euristici simple:

- ❖ In general 2-5 niveluri de decompozitie sunt suficiente. Max 7 la sisteme extrem de complexe.
- ❖ La fiecare nivel de abstractizare sa fie maxim 5-7 subsisteme.

DESCOMPUNERE IERARHICĂ

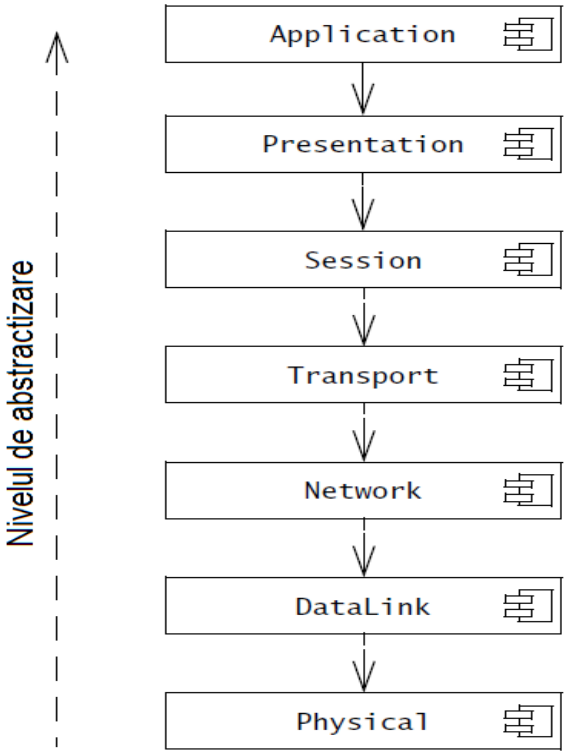
- O descompunere ierarhica a unui sistem produce un set ordonat de niveluri.
  - **Un nivel** este o grupare de subsisteme care furnizeaza servicii corelate, posibil realizate utilizand servicii din alt nivel.
  - Nivelurile sunt ordonate: **fiecare nivel depinde numai de nivelurile de sub el** si nu stie de cele 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.

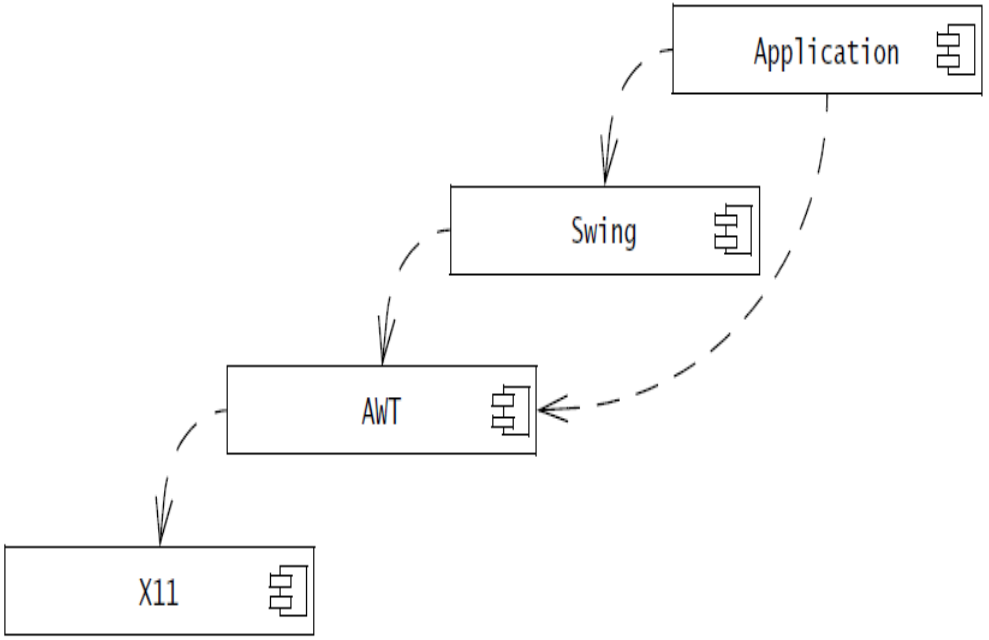
- **Uzual, descompunerile au 3-5 niveluri**

EXEMPLE: ARHITECTURI IERARHICE ÎNCHISE / DESCHISE

Modelul OSI (Open Systems Interconnection)



Java user interfaces



Avantajul unei arhitecturi ierarhice inchise:

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

Dezavantaje:

- fiecare nivel introduce un overhead (viteza↘, memorie↗, uneori cost dezvoltare↗)
- Adaugarea de noi functionalitati este dificila, daca nu a fost prevazuta

PARTIȚIONARE

= **Impartire in subsisteme relativ independente, care apartin aceluiasi nivel de abstractizare**, fiecare fiind responsabil pentru o clasa diferita de servicii.

Ideal: subsistemele sunt cat mai slab cuplate; unele pot opera si independent.

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 partitionare in subsisteme de nivel inalt, care ofera functionalitati specifice sau ruleaza pe noduri hardware specifice.
- Fiecare subsistem este descompus ierarhic, daca se justifica, in niveluri din ce in ce mai joase pana ce se obtin subsisteme suficient de simple.

ALEGEREA STRATEGIILOR / STILURILOR

Stil arhitectural = șablon de nivel foarte înalt pentru descompunerea în subsisteme.

Stilurile pot acoperi aspecte cum ar fi:

- strategii hardware/software
- managementul datelor persistente
- fluxul global al controlului
- politici de control al accesului

Exemple:

- Arhitecturi distribuite
  - Client - Server
  - Peer - to - peer
- Arhitecturi ierarhice
  - 3-tier
  - 4-tier
- Arhitecturi centrate pe date
  - Repository
- Arhitecturi centrate pe fluxul datelor
  - Secvential
  - Pipe and filter
- Arhitecturi dirijate pe evenimente
  - Mediator
  - Broker
- Arhitecturi orientate pe interactiune
  - Model - View - Controller (MVC)

Trebuie intelese bine avantajele si limitarile, problemele posibile ale fiecarui stil.

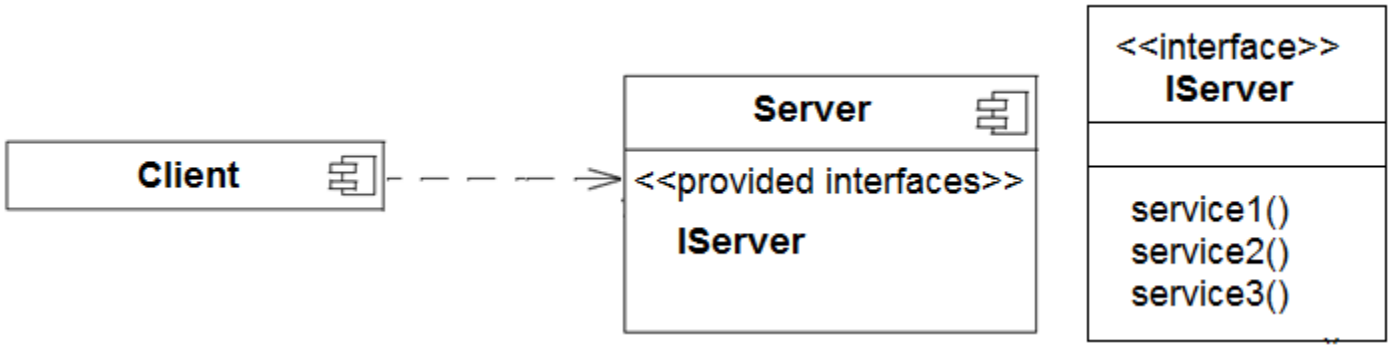
Odată ales un stil, el trebuie materializat (eventual adaptat) în arhitectură.

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 (cel mai adesea) sa cunoasca interfetele clientilor.

De regula, clientii sunt responsabili de functii la nivel aplicatie (e.g. interactiunea 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 de calcul si stocare 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.

- Arhitectura peer-to-peer necesita o proiectare atenta pentru a nu introduce posibilitati de deadlock sau complica fluxul controlului la nivelul fiecarui peer
- Poate pune probleme de fiabilitate / siguranta sau dimpotriva, le poate creste.

ARHITECTURI CENTRATE PE DATE: REPOSITORY

- Mai multi accesori comunica printr-un depozit de date, accesand si modificand datele.
- Accesorii sunt relativ independenti, ele comunicand numai prin depozitul de date.
- Scopul arhitecturii:
  - asigurarea integritatii datelor
  - prelucrari complexe centralizate, etc.

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.

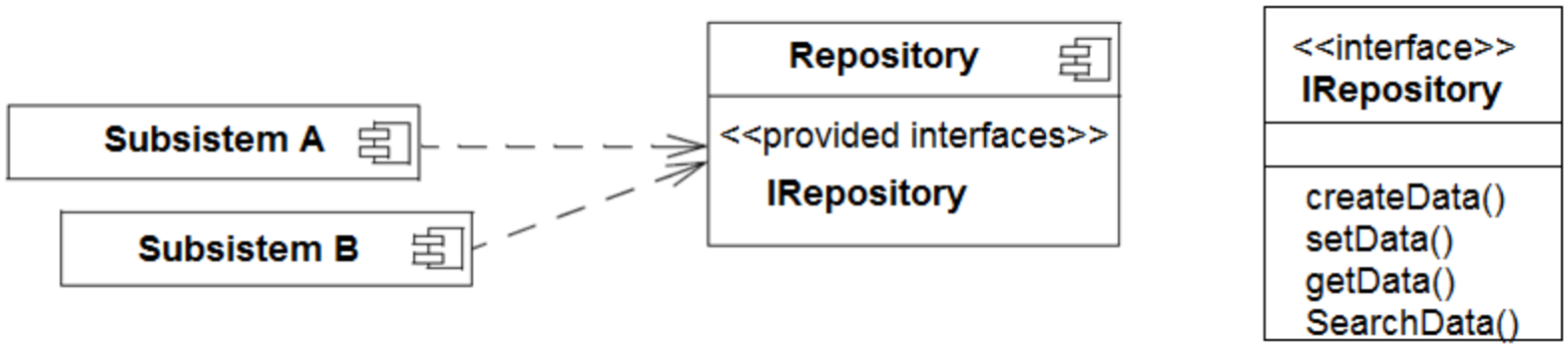
Arhitectura are 2 tipuri de subsisteme:

- Un 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:

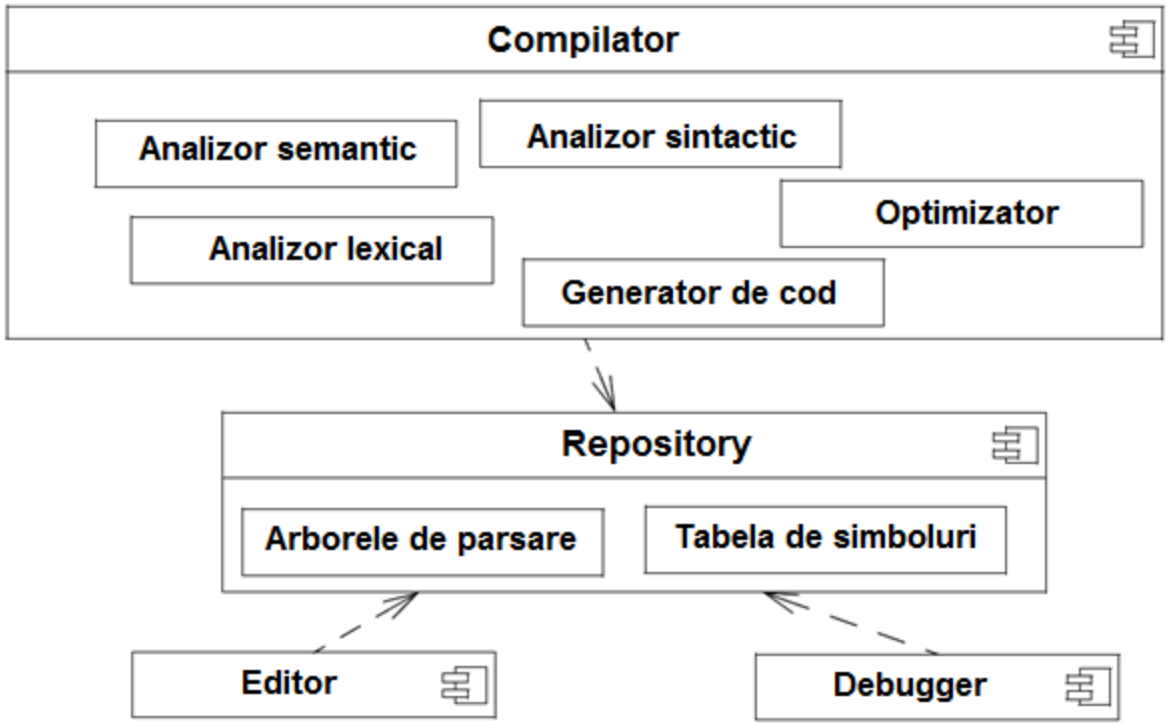
- Arhitecturi de tip “Repository pasiv”
- Arhitecturi de tip “Repository activ”.

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)



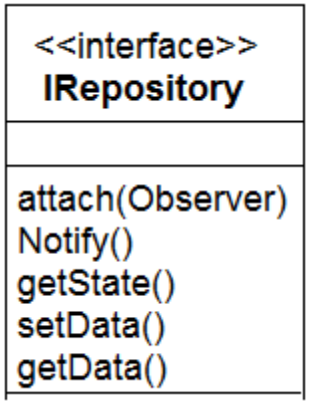
- Componentele Compilator, Debugger si Editor sunt apelate in mod independent de utilizator.

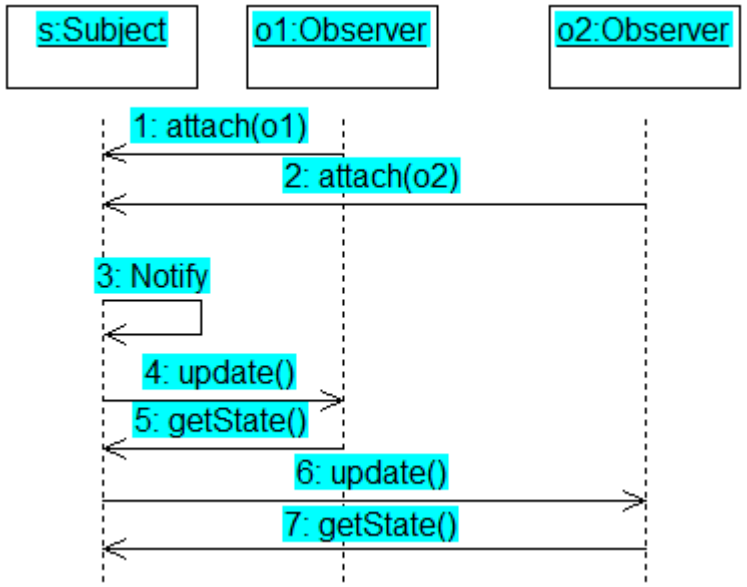
- Componenta Repository trebuie sa asigure serializarea accesurilor concurente.

- **Avantajele arhitecturii**:
  - Asigura integritatea datelor, operatii de backup si restaurare a datelor
  - Asigura scalabilitate – pot exista oricati accesori
  - Reduce overhead-ul produs de transmisia datelor intre 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)

Avantajele arhitecturii:

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

Dezavantaje:

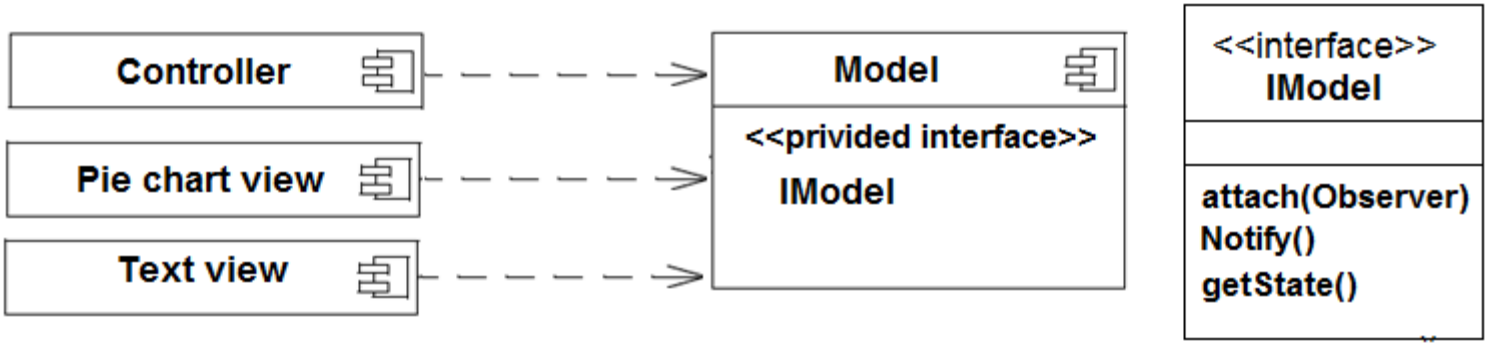
- Dependenta puternica intre componenta Repository si accesori: modificarile structurale ale depozitului pot afecta toti accesorii
- Probleme in sincronizarea accesorilor → dificultati in proiectare si testare

ARHITECTURI ORIENTATE PE INTERACTIUNE: *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

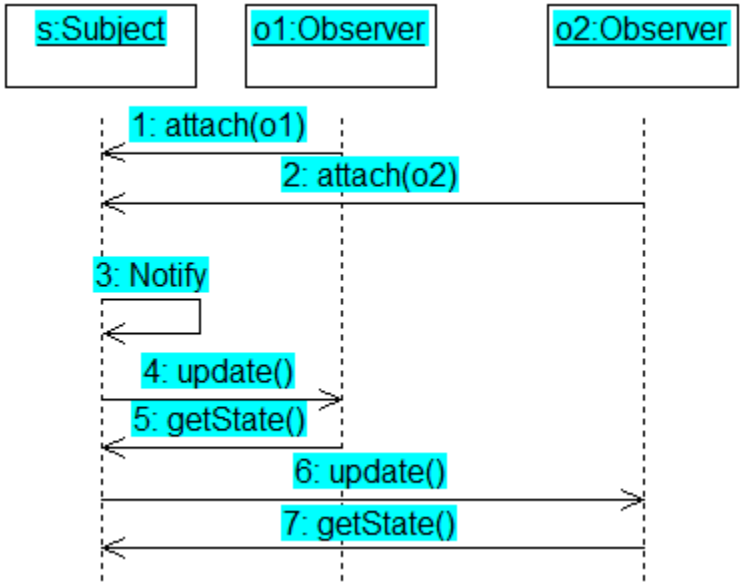
❖ MVC este un caz particular de Repository, unde subsistemele **Model** implementeaza Repository (structura de date centrala) iar subsistemele **Controller** dicteaza fluxul controlului.



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**.



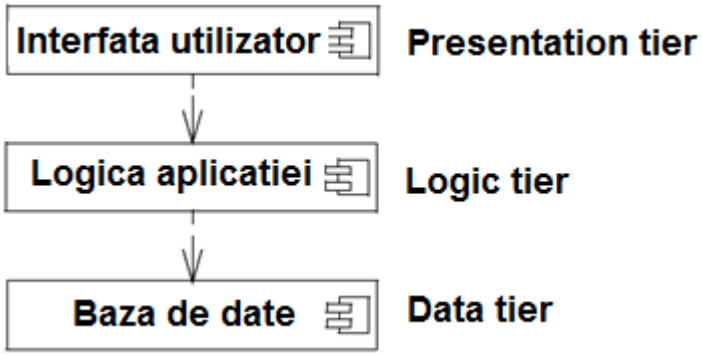
❖ Arhitectura MVC este adecvata sistemelor interactive, mai ales atunci cand sunt necesare mai multe vederi ale modelului.

❖ Motivatia:
- separare stocare si operatii efective pe date de UI
- UI (View si Controller) este mult mai supusa schimbarilor decat modelul

ARHITECTURI IERARHICE: *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.



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; poate contine si o parte din logica aplicatiei (nerecomandat)
  - Web server – trateaza cererile provenite de la web browser: implementeaza nivelul aplicatie (total sau partial)
  - Baza de date – asigura gestiunea datelor persistente

Avantajul arhitecturii:

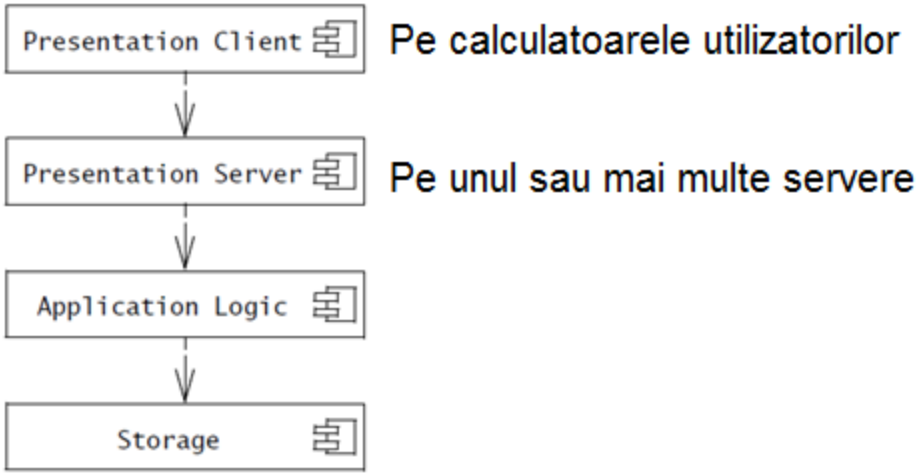
- Fiecare dintre cele 3 niveluri poate fi îmbunatatit sau înlocuit independent, în cazul unei schimbari a cerintelor sau a unei schimbari tehnologice.

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*.

ARHITECTURI CENTRATE PE FLUXUL DATELOR

- Fluxul global al controlului este structurat intr-o secventa de transformari ale datelor de intrare, pentru a produce anumite rezultate. Fiecare etapa de transformare este alocata unui subsistem.
- Conexiunile dintre subsistemele de transformare a datelor pot fi de tip I/O stream, I/O buffers, „pipe”, sau alte tipuri.
- Arhitecturi tipice bazate pe fluxul datelor:
  - Prelucrarea secventiala – pipeline secvential (classic)
  - Pipe and filter – pipeline ne-secvential

Prelucrarea secventiala - modelul de prelucrare clasic

- Executia unei etape de transformare a datelor se poate initia numai după ce s-a terminat executia etapei precedente.
- Comunicarea intre subsistemele de transformare se face prin bufere sau fisiere temporare.

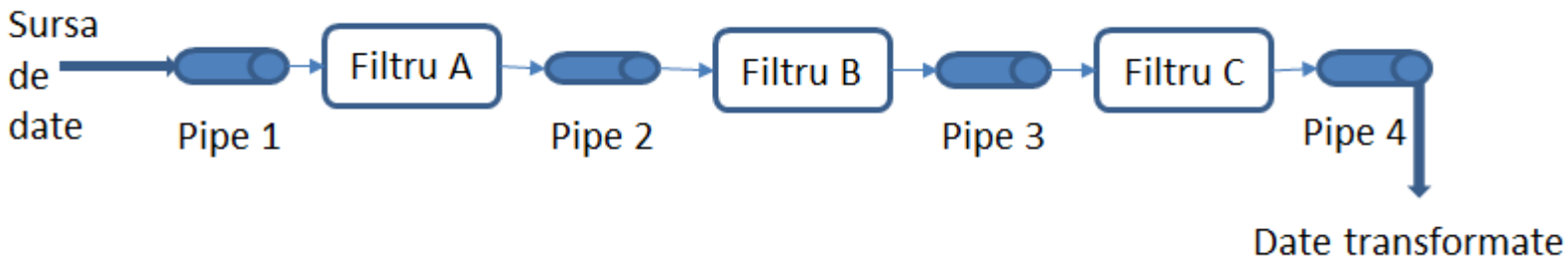
Avantaje:

- Divizarea in subsisteme este simpla: fiecare subsistem poate fi un program independent care primeste date de intrare si produce date de iesire.

Dezavantaje

ARHITECTURA *PIPE AND FILTER* (1)

- Scopul arhitecturii: descompunerea unei prelucrari complexe in prelucrari simple separate, care transforma datele incremental si pot fi executate in paralel.
- Arhitectura este alcatuita din:
  - O sursa de date
  - Mai multe “filtre” (etape de procesare), fiecare fiind capabil sa execute o prelucrare simpla
  - Conectori (pipes) prin care sunt transferate datele intre filtre
    - Filtrele si conectorii formeaza un « pipeline »
  - Un receptor al datelor prelucrate in pipeline

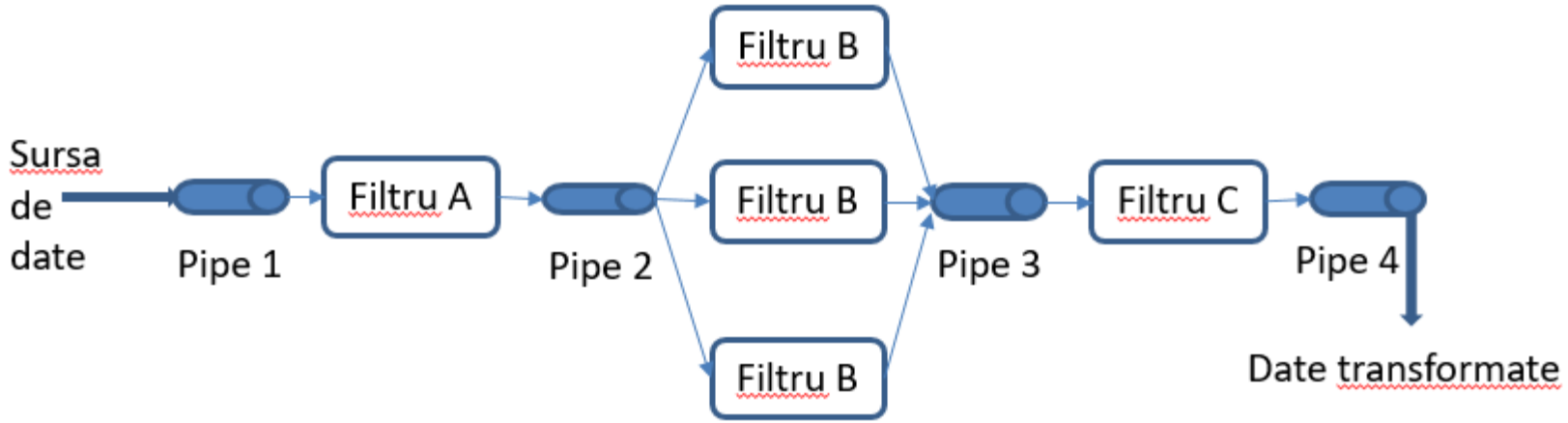


ARHITECTURA *PIPE AND FILTER* (2)

- In general, intrarile si iesirile filtrelor sunt structurate ca fluxuri de date (streams), iar conectorii sunt implementati ca bufere FIFO.
- Avantajele structurarii datelor in fluxuri:
  - Fiecare filtru isi incepe executia imediat ce s-au primit date in conectorul sau de intrare si se executa atata timp cat exista date in conectorul respectiv.
  - Un filtru isi poate incepe executia chiar daca filtrul anterior in secventa de procesare, care produce fluxul său de intrare, nu si-a terminat executia.
  - Filtrele din pipeline pot fi executate in paralel
- Filtrele se pot executa pe resurse de calcul diferite: filtrele intensiv computationale pot rula pe hardware de inalta performanta, altele pe hardware obisnuit.
- Filtrele pot fi executate pe calculatoare aflate in locatii geografice diferite, in apropierea resurselor de care au nevoie.

ARHITECTURA *PIPE AND FILTER* (3)

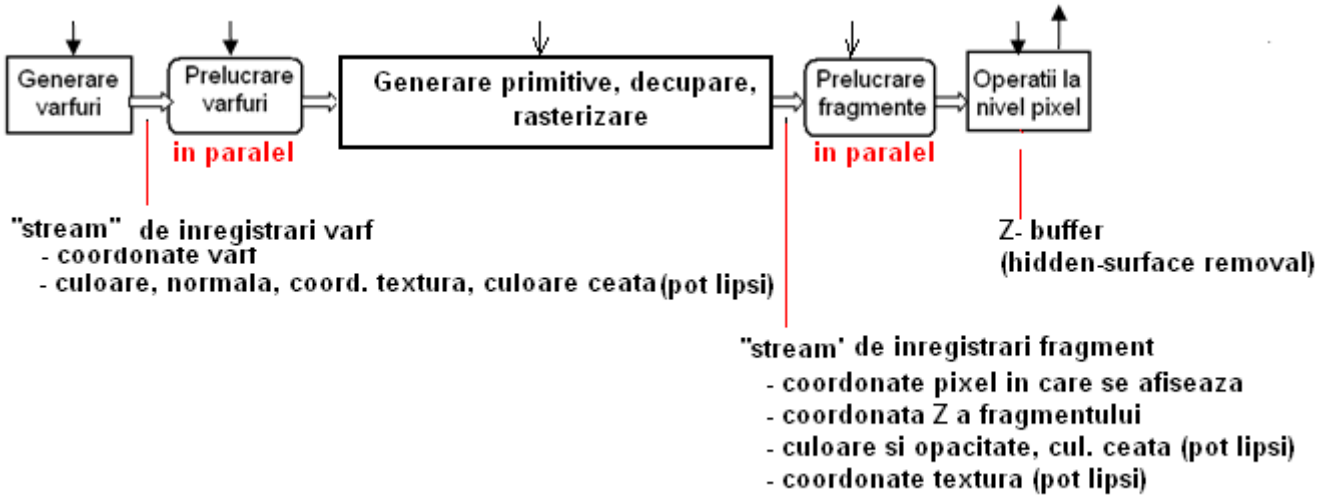
- Mai multe instante ale unui filtru pot fi executate in paralel pe aceeasi masina sau pe masini diferite.



- Cele trei instante ale filtrului B se executa in paralel pentru date diferite provenite din acelasi conector de intrare.
- Datele produse sunt memorate in acelasi conector de iesire.

ARHITECTURA *PIPE AND FILTER* (4)

Exemplu: Graphics pipeline!



Filtrul “Generare varfuri” produce un flux de inregistrari “varf”.

Inregistrarile “varf” sunt prelucrate in paralel de instante ale filtrului “Prelucrare varfuri”(Vertex shader). Instantele sunt executate de procesoare diferite, pentru date (varfuri) diferite.

ARHITECTURA *PIPE AND FILTER* (5)

Avantajele arhitecturii

- Permite paralelismul si asigura o rata inalta de executie a tascurilor de prelucrare a unor volume mari de date.
- Permite reutilizarea: filtrele implementeaza prelucrari simple, care pot fi reutilizate in diferite aplicatii.
- Mentenanta sistemului este mai simpla:
  - pot fi efectuate mai usor modificari
  - filtrele sunt slab cuplate
- Ofera flexibilitate in proiectarea sistemului, fiind posibile atat executia secventiala cat si cea paralela a filtrelor.
- Toleranta la caderi: daca o instanta a unui filtru cade sau masina pe care se executa devine nedisponibila, prelucrarea aflata in curs pe acea instanta poate fi alocata altei instante.

ARHITECTURA *PIPE AND FILTER* (6)

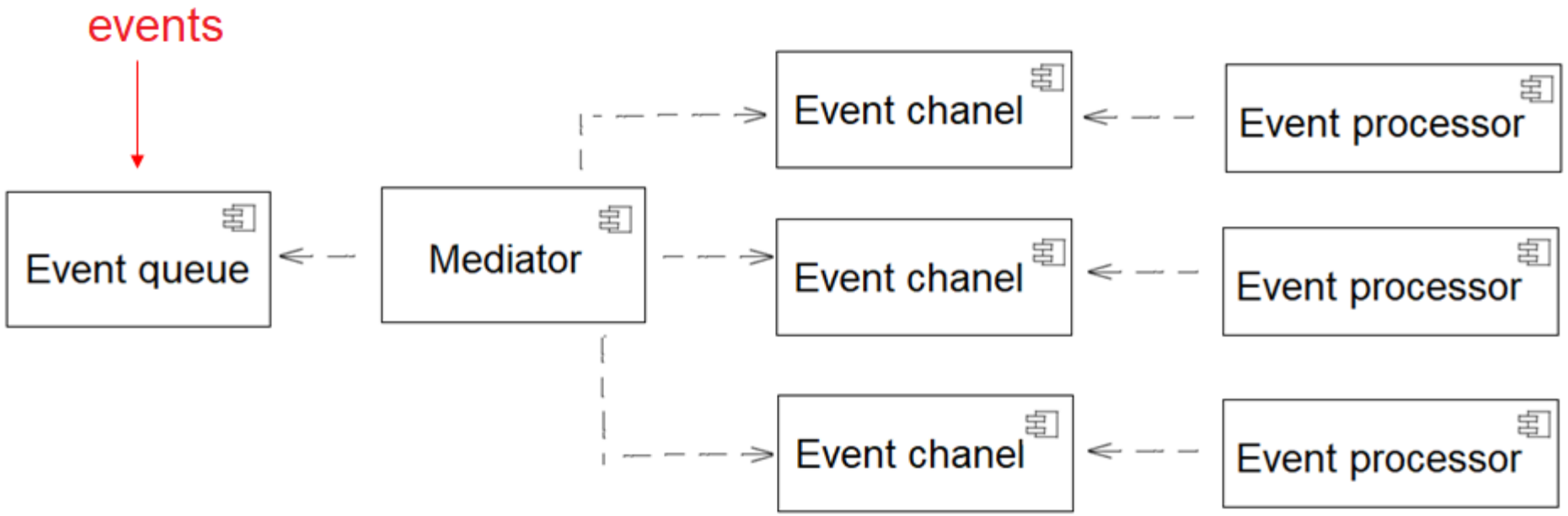
Dezavantajele arhitecturii

- Flexibilitatea crescuta a arhitecturii poate introduce complexitate, mai ales cand filtrele dintr-un pipeline sunt distribuite pe diferite servere.
- Este necesara o infrastructura care sa asigure ca datele transferate intre filtre nu se pierd.
- Nu este adecvata interactiunilor.
- Poate introduce un overhead datorita transferului datelor intre filtre.

ARHITECTURA DIRIJATA DE EVENIMENTE (1)

- Arhitectura distribuita frecvent utilizata
- Adecvata sistemelor care colecteaza date de la senzori (IOT)
- Alcatuita din componente care proceseaza evenimente asincron, in paralel

❖ 2 tipuri de topologii: Mediator si Broker



ARHITECTURA DIRIJATA DE EVENIMENTE (2)

Topologia Mediator

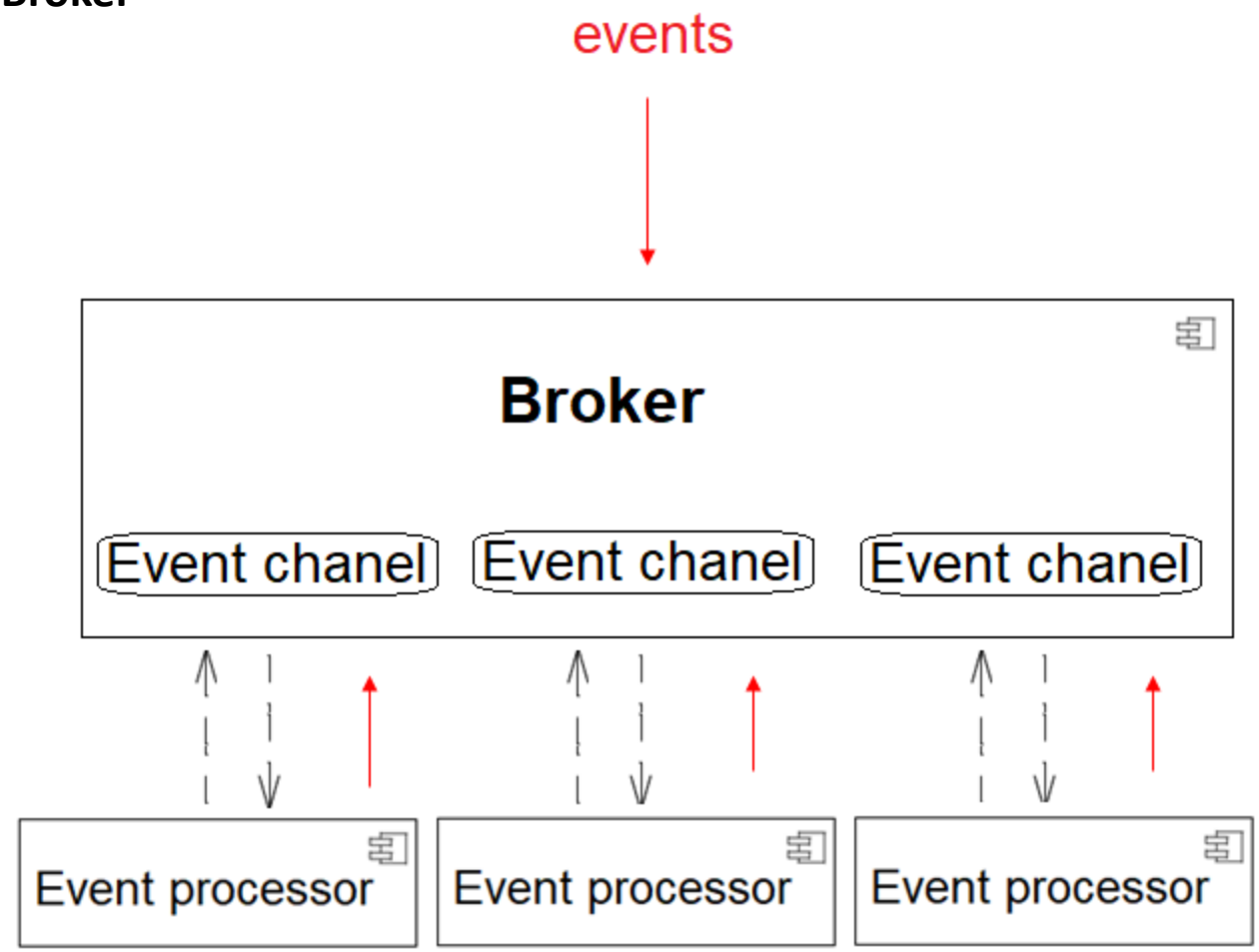
- 4 tipuri de componente: coada de evenimente, mediatorul, canale eveniment si procesoare de evenimente.
- Mediatorul descompune tratarea fiecarui eveniment (**evenimentul initial**) in pasi de procesare care pot fi executati secvential sau in paralel.
- Pentru executia pasilor de procesare a evenimentului initial mediatorul genereaza **evenimente de procesare** care sunt transmise in canale eveniment specifice.
- Fiecare procesor de evenimente preia evenimentele dintr-un canal de evenimente si efectueaza prelucrari specifice, care implementeaza logica aplicatiei.
- Canalele de evenimente pot fi:

  - de tip “coada de mesaje” (fiecare mesaj are un singur consumator) – evenimentele dintr-un canal sunt preluate de un singur procesor

  - de tip “*topics* and *subscriptions”* (un mesaj este transmis la mai multi consumatori inregistrati pentru un anumit tip de mesaj) - evenimentele dintr-un canal sunt preluate si tratate in paralel de mai multe procesoare de evenimente

ARHITECTURA DIRIJATA DE EVENIMENTE (3)

Topologia Broker



ARHITECTURA DIRIJATA DE EVENIMENTE (4)

Topologia Broker

- 2 tipuri de componente: Broker si procesoare de evenimente.
- Broker-ul dirijeaza evenimentele catre procesoarele de evenimente
- Fiecare procesor de evenimente prelucreaza un eveniment si genereaza un nou eveniment la Broker pentru a anunta terminarea prelucrarii. Evenimentul poate fi directionat de Broker catre un alt procesor de evenimente.

Avantaje/dezavantaje ale arhitecturii dirijate de evenimente

- Complexa: asincrona, distribuita
- Lipsa de atomicitate in procesarea unei tranzactii (eveniment): dificil de divizat procesarea in pasi executati in paralel
- Scalabilitate mare

ARHITECTURA DIRIJATA DE EVENIMENTE (5)

- Avantaje/dezavantaje ale arhitecturii dirijate de evenimente
- Adecvata pentru aplicatii mici si foarte mari
- Componente foarte decuplate
  - Componentele Event processor pot fi modificate fara a afecta pe celelalte
  - In cazul Mediator o schimbare la un Event processor poate necesita o schimbare la Mediator
- Dezvoltarea si testarea poate fi complicata: natura asincrona, tratarea conditiilor de cadere, procesoare de evenimente care nu raspund, etc.
- Performanta inalta: prelucrarie asincrone paralele eclipseaza costul transferului de mesaje prin cozi

EXERCITIU

1. Identicati criteriile de proiectare, prioritizati, analizati compromisuri pentru pentru proiectul vostru

2. Alegeti stil/stiluri potrivite pentru proiectul vostru. Justificati

IDENTIFICAREA SUBSISTEMELOR (1)

1. Descompunerea initiala in subsisteme pleaca de la clasele identificate in analiza:

   -se grupeaza in acelasi subsistem clasele corelate functional;

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

2. O alta euristica de identificare a subsistemelor:

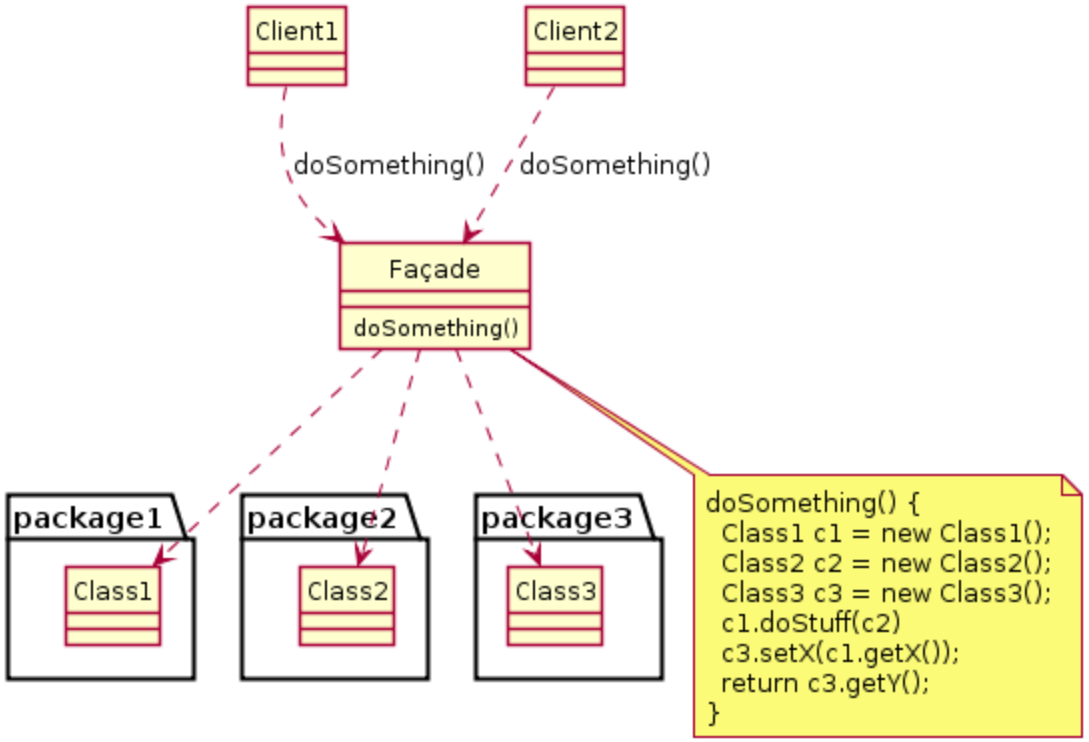
   - se grupeaza într-un subsistem clasele obiectelor care participa într-un caz de utilizare.

❖**Sablonul de proiectare Façade**

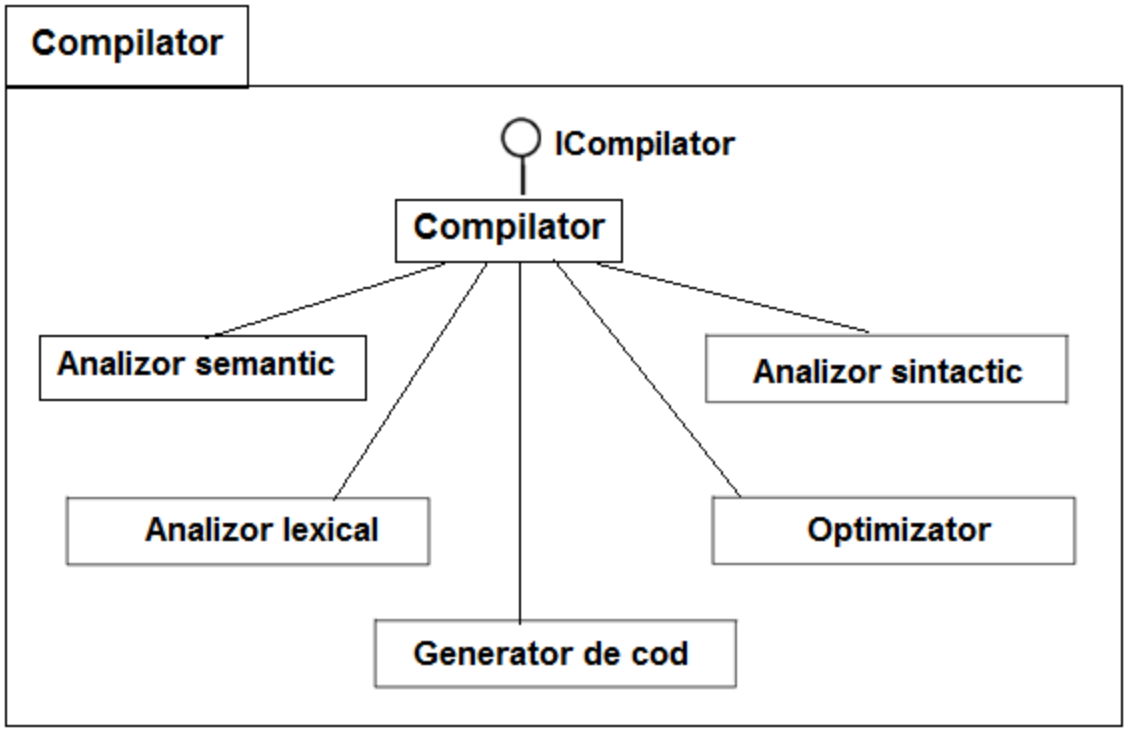
ofera un mecanism adecvat

pentru gruparea in subsisteme şi

reducerea cuplarii



IDENTIFICAREA SUBSISTEMELOR (2)



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

Şablonul Façade

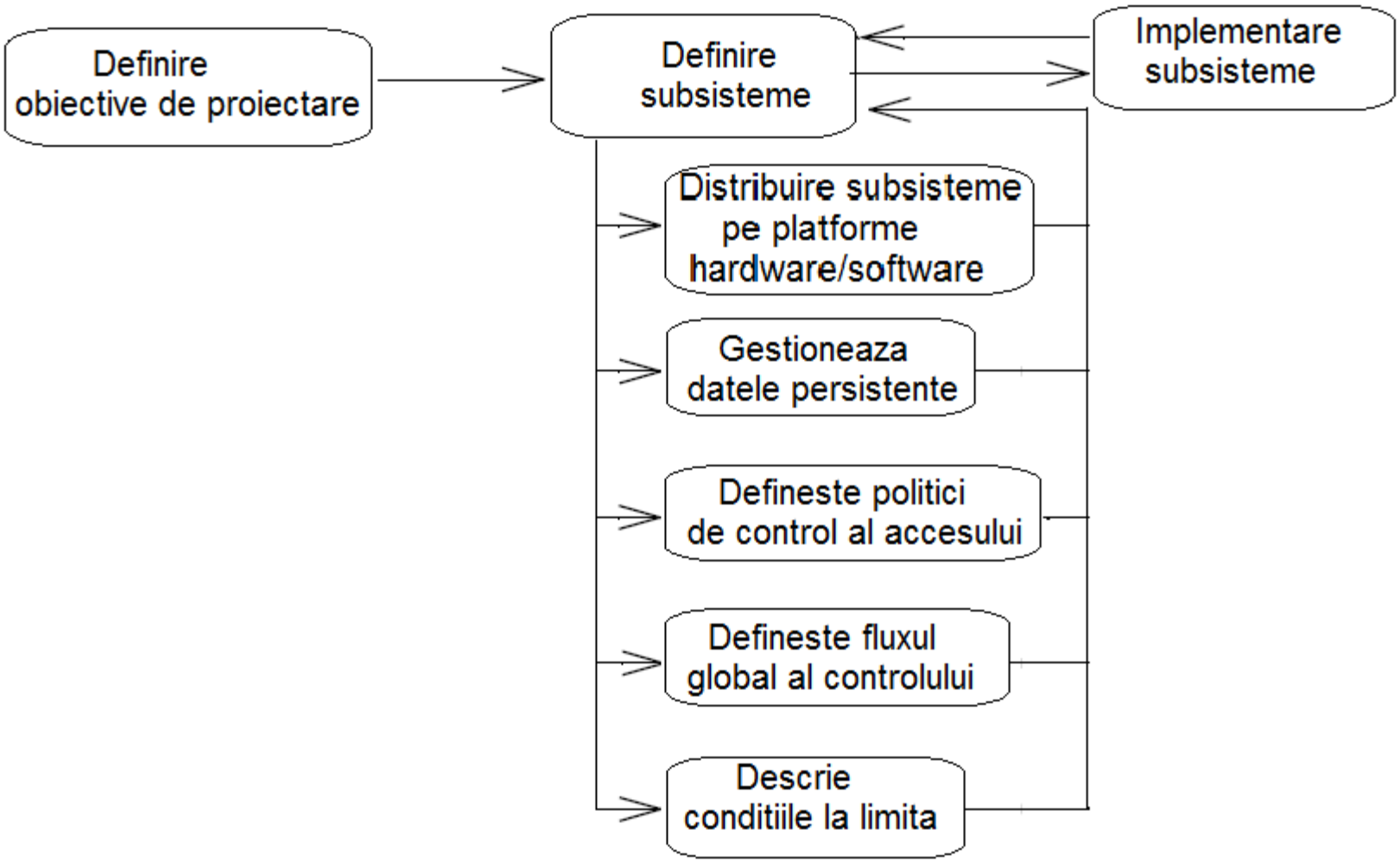
Clasele grupate in subsistemul **Compilator** comunica prin interfata **ICompilator** furnizata de clasa **Compilator.**

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. Ex: pachete de componente UI, sisteme de baze de date, etc. Desigur, prin interfata lor predefinita, fixa, introduc constrangeri de proiectare.
- 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 comunicare, 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 (pornire/oprire, 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 / reteaua locala.
- Alocarea subsistemelor pe noduri hardware influenteaza puternic performanta si complexitatea sistemului: trebuie efectuata la inceputul proiectarii de sistem.
- Selectarea configuratiei hardware include si selectarea masinii virtuale pe care se va executa sistemul: SO, SGBD, networking software
- Selectia configuratiei hardware poate fi restrictionata: hardware existent, cost.
- Alte criterii: anumite componente trebuie sa se execute in locatii specifice (de ex., software pentru un bancomat), 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)

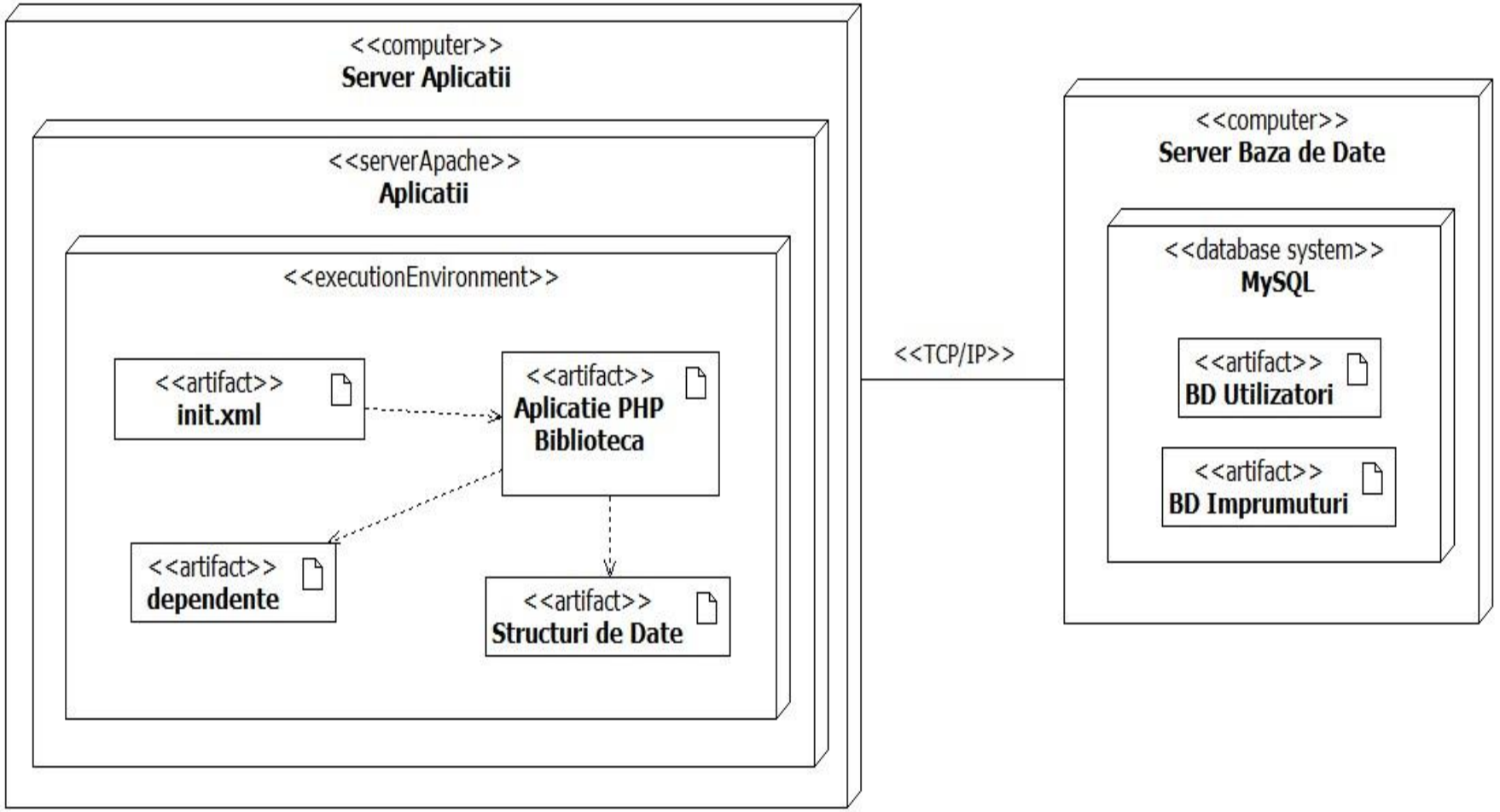


Diagrama de distributie in UML 2.x.

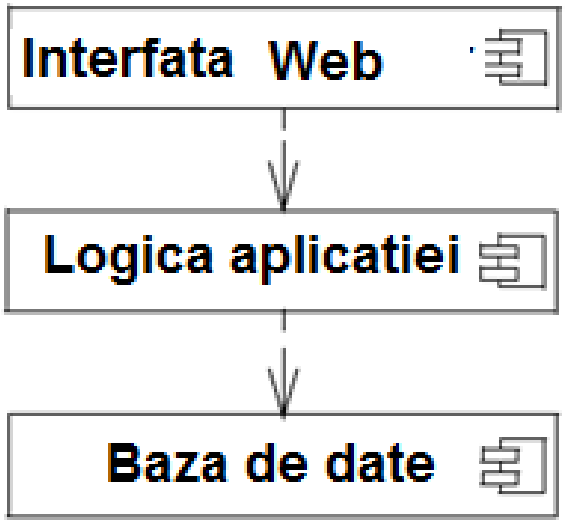
IDENTIFICAREA ŞI GESTIONAREA DATELOR PERSISTENTE

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

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



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

Date utilizatori si carti

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 utilizator (actor) are anumite drepturi de acces la resursele sistemului.

- Se determina obiectele partajate de actori: fișiere, procese, baza de date, etc.
- 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 actorilor la clase/date se poate sintetiza prin **matricea de acces**:

	Obiect A	**Obiect B**	**Obiect C**
Actor 1	Oper1A() Oper2A()		Oper1C() Oper3C()
Actor 2	Oper1A() Oper2A() Oper3A()	Oper1B() Oper2B() Oper3B()	
Actor 3	Oper3A()		Oper1C()

Drepturile de acces

FLUXUL GLOBAL AL CONTROLULUI (1)

Fluxul controlului = secventierea actiunilor la executia sistemului.

Exista 3 metode de nivel inalt 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)

FLUXUL GLOBAL AL CONTROLULUI - CONTROL DIRIJAT PROCEDURAL

Control dirijat procedural

- Operatiile asteapta intrarile de la un actor atunci cand le sunt necesare.
- Folosit in sistemele mai vechi axate pe prelucrari masive, secventiale, de date

Exemplu (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);
```

FLUXUL GLOBAL AL CONTROLULUI - CONTROL DIRIJAT DE EVENIMENTE

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, (prin apelul unei functii de tip event handler) pe baza informatiei asociate evenimentului.

- Este o structura de control simpla, care centralizeaza toate intrarile in bucla principala, fara a impune o secventialitate stricta pentru majoritatea lor.
- Uzuala in sistemele cu GUI complexe.
- Prelucrarile sunt presupuse a fi relativ rapide, altfel interfata/sistemul devine neresponsiva. Prelucrarile complexe trebuie tratate cu grija , eventual mutate pe fire de executie paralele celui in care se gaseste bucla de primire si distributie de evenimente.

FLUXUL GLOBAL AL CONTROLULUI - CONTROL DIRIJAT DE EVENIMENTE

Exemplu (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 - CONTROL BAZAT PE FIRE DE EXECUTIE

Control bazat pe fire de executie

- Pentru numeroase obiecte sau evenimente sistemul creaza cate un thread; Numeroase intrari, calcule si iesiri se pot desfasura in paralel
- Dificultati legate de accesul concurent la date si resurse

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.

Proiectarea detaliata (ultra brief)

Proiectarea de detaliu

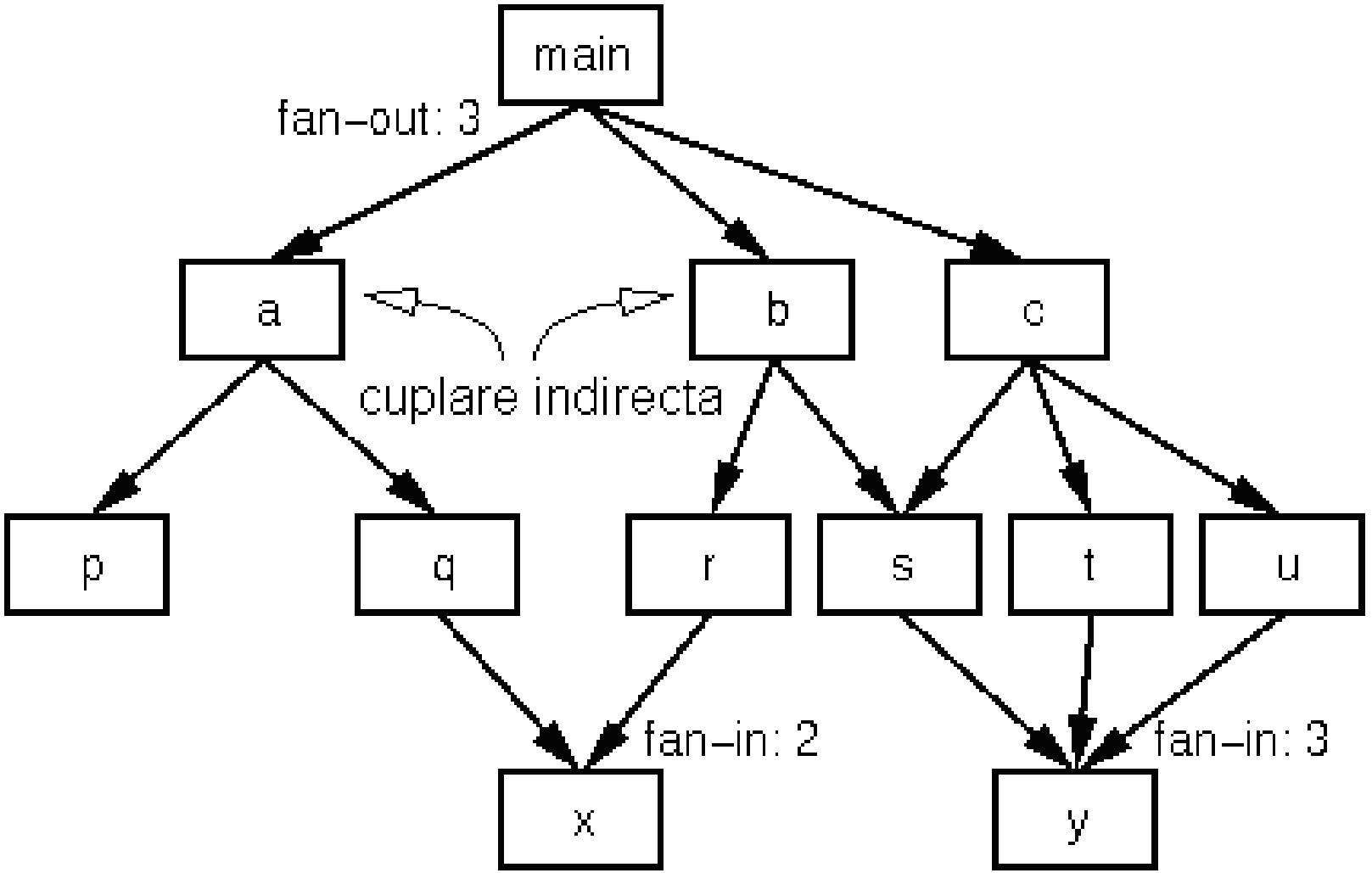
- Se efectueaza la nivelul modulelor definite in proiectarea arhitecturala
- Poate avea loc in paralel, pentru diferite module
- Detaliaza modelul de proiectare arhitecturala:
- pot fi definite module de nivel mai coborat
  - se detaliza componenta claselor: atributele si functiile membre
  - se aleg biblioteci utilizate in implementare
- se incurajeaza reutilizarea
- sunt descrisi algoritmii

Principii de modularizare

- Modulele trebuie sa fie simple si cat mai independente:
  - O modificare a unui modul are influenta minima asupra altor componente
  - O schimbare mica a cerintelor nu conduce la modificari majore ale arhitecturii software
  - Efectul unei conditii de eroare este izolat in modulul care a generat-o
  - Un modul poate fi inteles ca o entitate de sine-statatoare
- Modulele trebuie sa “ascunda” modul de implementare a interfetei lor

Recomandari in proiectarea functionala

- **Minimizarea cuplarii** intre module
  - Minimizarea numarului de elemente prin care comunica modulele
  - Evitarea cuplarii prin structuri de date
  - Evitarea cuplarii prin variabile “steag” (cuplarea prin control)
  - Evitarea cuplarii prin date globale
- Factorizarea functionalitatilor comune in module reutilizabile
- Maximizarea coeziunii interne
- Minimizare fan-out (numarul de module apelate): „Fan-out“ mare: modulul depinde de multe module
- Maximizare fan-in (numarul de module care apeleaza un modul): reutilizare “Fan-in” mare: un numar mare de module depind de el



Diagarama de structura

Recomandari in proiectarea OO

Lorenz (1993): din experienta multor proiecte industriale

- 1. Dimensiunea medie a unei metode (LOC): < 24 LOC (pentru programe C++)
- 2. Numarul mediu de metode/clasa: < 20

medii mai mari indica prea multa responsabilitate in putine clase

3. Numarul mediu de variabile instanta (atribute) / clasa: < 6

mai multe inseamna ca o clasa “face” prea mult

4. Adancimea arborelui de mostenire: < 6

(incepand de la radacina sau de la clasele bibliotecii de dezvoltare)

5. Numarul de relatii subsistem/ subsistem: < valoarea metricii 6

6. Numarul de relatii clasa/ clasa in fiecare subsistem:

trebuie sa fie relativ mare → coeziune mare a claselor din acelasi subsistem

daca o clasa dintr-un subsistem nu interactioneaza cu multe alte clase din subsistem→ ar trebui plasata in alt subsistem

7. Utilizarea atributelor:

daca grupuri de metode dintr-o clasa utilizeaza seturi diferite de atribute

→ clasa ar putea fi divizata in mai multe clase

8. Numarul mediu de linii comentariu / metoda: >1

9. Numarul de rapoarte problema / clasa: mic

10. Numarul de reutilizari ale unei clase:

daca o clasa nu este reutilizata in diferite aplicatii (mai ales o clasa abstracta)

→ ar putea fi necesara sa fie reproiectata

11. Numarul de clase si metode la care s-a renuntat:

ar trebui sa fie relativ constant pe durata dezvoltarii

dezvoltarea incrementala → in spiritul proiectarii iterative OO

Sabloane de Proiectare

(Design Patterns)

Ce este un.. Sablon de Proiectare?

O **problema generica** ce este intalnita in mod repetat (in proiectarea de software)

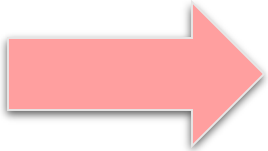
+

Solutia generica pentru acea problema, explrimata folosing:

- Clase, obiecte, functii, etc.
- Text
- Diagrame (e.g. UML)

Aspect Cheie: Generalitatea

Atat problema cat si solutia sunt generale – nu sunt instante concrete



Ele sunt exprimate intr-un mod abstract, ce poate fi ulterior aplicat oricare instante concrete a problemei

SUNT S.P. UTILE ? DA !

- ASISTA REZOLVAREA DE PROBLEME (similare cu probleme intalnite si rezolvate anterior)
- Capteaza si impartaseste

  EXPERIENTA DE PROIECTARE
- LIMBAJ COMUN PT. PROIECTANTI

START: CARTE DE REFERINTA

MUST READ

Design Patterns: Elements of Reusable Object-Oriented Software

Authori: Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides

Cunoscuta ca: GoF Book ("Gang of Four Book")

Publicata in: 1994 – mare succes

A deschis domeniul Sabloanelor de Prelucrare

Elemente Esentiale – 4

NUME

Descrierea Problemei

Descrierea Solutiei

Consecinte

Nume

- Vocabular de proiectare
- Limbaj comun pentru comunicare si documentare

Alegerea unui nume simplu si sugestiv !

Descrierea Problemei

- Cerinte, probleme
- Context
- Conditii de aplicare a sablonului

Descrierea Solutiei

- Elemente ce compun solutia:
  – clase, obiecte, functii, etc.
- Relatii intre elemente
- Responsabilitati
- Colaborari

Consecinte *(ale aplicarii sablonului*

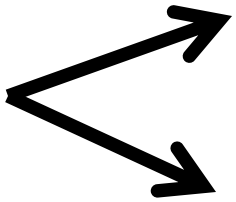
- Efecte pozitive (beneficii)
- Posibile efecte negative

DESCRIEREA CONSECINTELOR ESTE ESENTIALA PENTRU EVALUAREA ALTERNATIVELOR DE PROIECTARE
(daca sa folosim sau nu un sablon intr-o situatie concreta)

Elemente Suplimentare

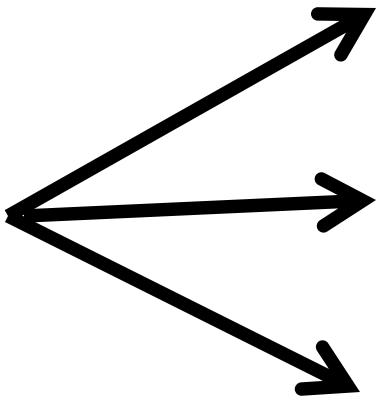
- **Variante**
- **Exemple de implementare**
  (in diferite limbaje de programare)

CLASIFICARE



Generale

Specifice unor domenii



Creationale

Structurale

Comportamentale

Sabloanele generale:

- ✓ *GoF book*
- ✓ http://en.wikipedia.org/wiki/Design_Patterns
- ✓ *etc (Google)*

Sabloane GoF

- Creational Patterns
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Structural Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- Behavioral Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

Descrierea din GoF Book:

1. **Name si Clasificare:**
2. **Intentie:**
   Descrierea scopului salblonului si a motivului pentru care este necesar
3. **Alte nume**
4. **Motivare:**
   Un scenariu concret descriind o problema pe care sablonul o rezolva.
5. **Aplicabilitate:**
   situatii in care sablonul se aplica; cum sa le recunoastem

GoF description.. cont.

6. **Structura:**
   Reprezentare grafica a solutiei
   Foloseste: diagrame de clase,
   diagrame de interactiune, etc.
6. **Participanti:**
   Lista claselor si obiectelor participante si a rolurilor lor
7. **Colaborari:**
   Descrierea modului in care clasele sau obiectele interactioneaza.

GoF description.. cont.

9. **<u>Consecinte</u>:**
    - **Rezultatele (beneficiile) aplicarii**
    - **Efecte colaterale**
    - **Posibile probleme si solutii pentru ele**

GoF description.. cont.

10. **Implementare:** Cum poate fi implementat
11. **Sample Code:**
    Exemple de cod in C++ or Smalltalk
12. **Utilizari cunoscute:**
    Example ale utilizarii sablonului in sisteme reale.
13. **Sabloane asociate:**
    Alte sabloane care au legatura cu acesta (inrudite, asemanatoare sau folosite impreuna)

Example: MFC document-view

Microsoft Foundation Classes – Windows development framework

Most MFC applications use Document-View architecture:

- ✓ **Separate data from presentation**
- ✓ **Data: as Document object**
- ✓ **Presentations: as View objects**
- ✓ **Changes in a view are reflected in all others !**

Doc/View is a particular case of the "Observer" design pattern

Observer

1. **Name:** Observer

2. **Intent:** define a "1 to many" dependency between objects:
   when one object (the subject) changes its state, the others (the observers) are notified.

3. **Also known as:**
   Dependents, Publish-Subscribe.

Observer

4. **Motivation**

   Side effect of partitioning a system in classes is the need to keep consistency between some objects.

   Direct coupling reduces usability.

   Observer offers a very general mean of keeping consistency without direct coupling.

Observer

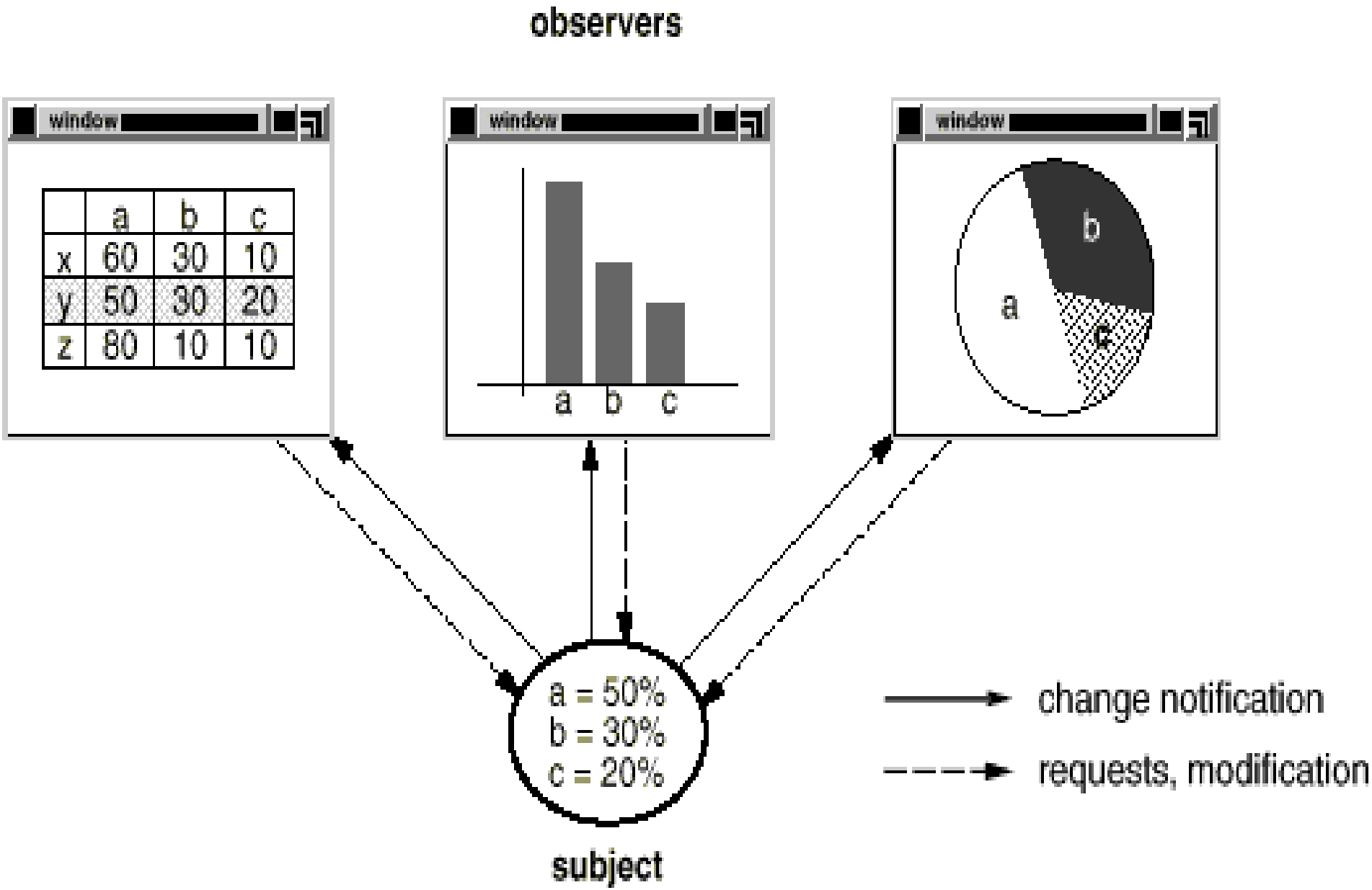
Key actors:

Subject

Observer

- ✓ A subject can have any number of independent observers.
- ✓ All observers are notified each time the subject changes its state.
- ✓ On notification, the observers usually query the subject state to update themselves.

Observer


observers
window
window
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
b
a
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modification

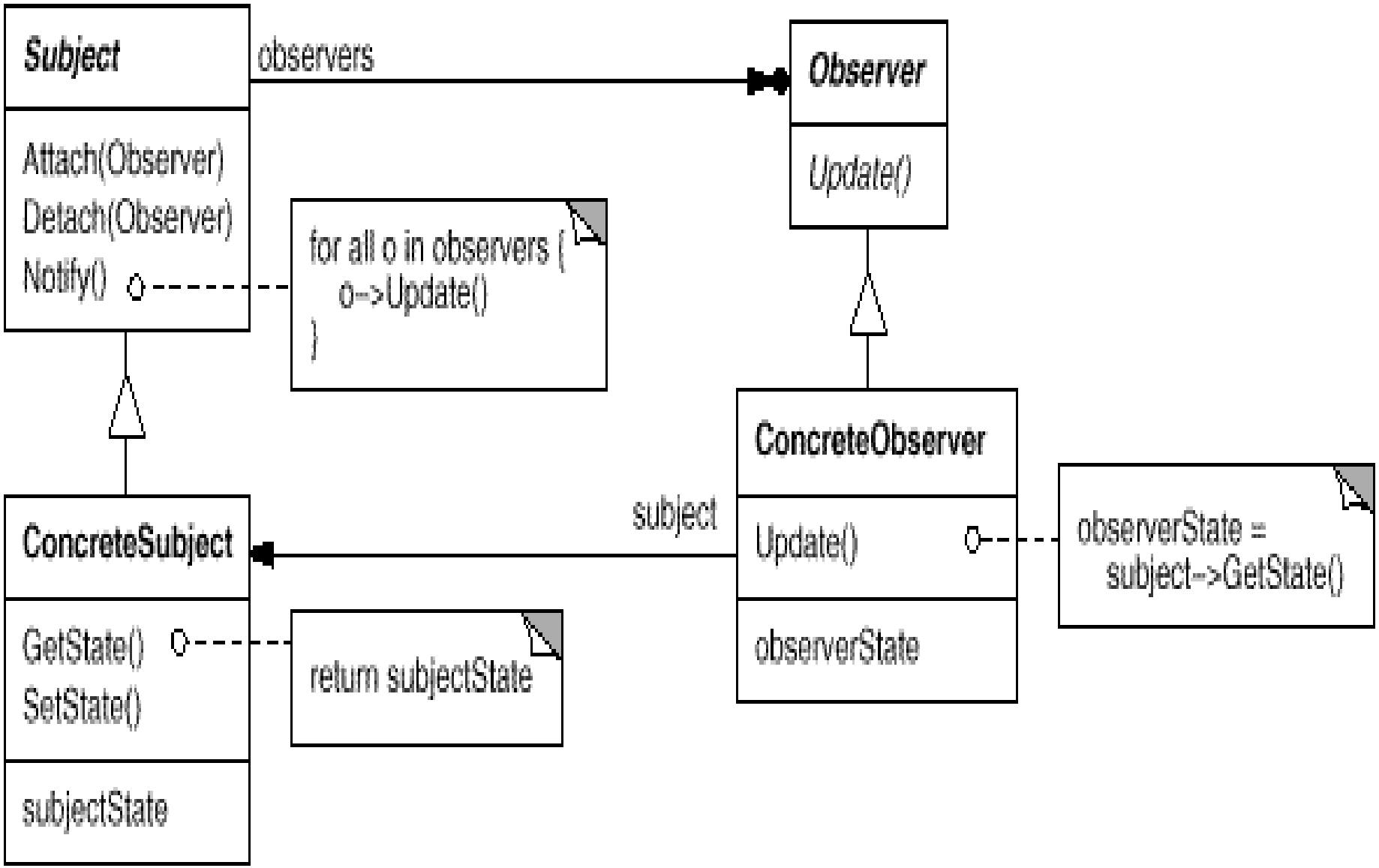
Observer

5. **Applicability:** any of the following situations:
   - When an abstraction has 2 independent aspects, one depending on the other: allows modularizing and reusing them independently
   - When changes in an object can affect any number or types of other objects
   - When an object must notify other objects without making assumptions about who are they, what type are they etc.

Basically: reduce coupling

Observer

6. Structure



Observer

7. Participants

- **Subject**
  - Keeps track of the observers
  - Allows them to Attach/Detach
  - Define the Notify method that notifies all the observers by calling their update method
- **Observer**
  - Defines the Update() interface for receiving notifications
- **ConcreteSubject**
  - Has concrete state of interest for the ConcreteObserver objects.
  - Calls Notify from the base class each time its stat is changed.
- **ConcreteObserver**
  - Keeps a reference to a ConcreteSubject.
  - Updates its state each time it receives a notification, by querying the ConcreteSubject about its state.

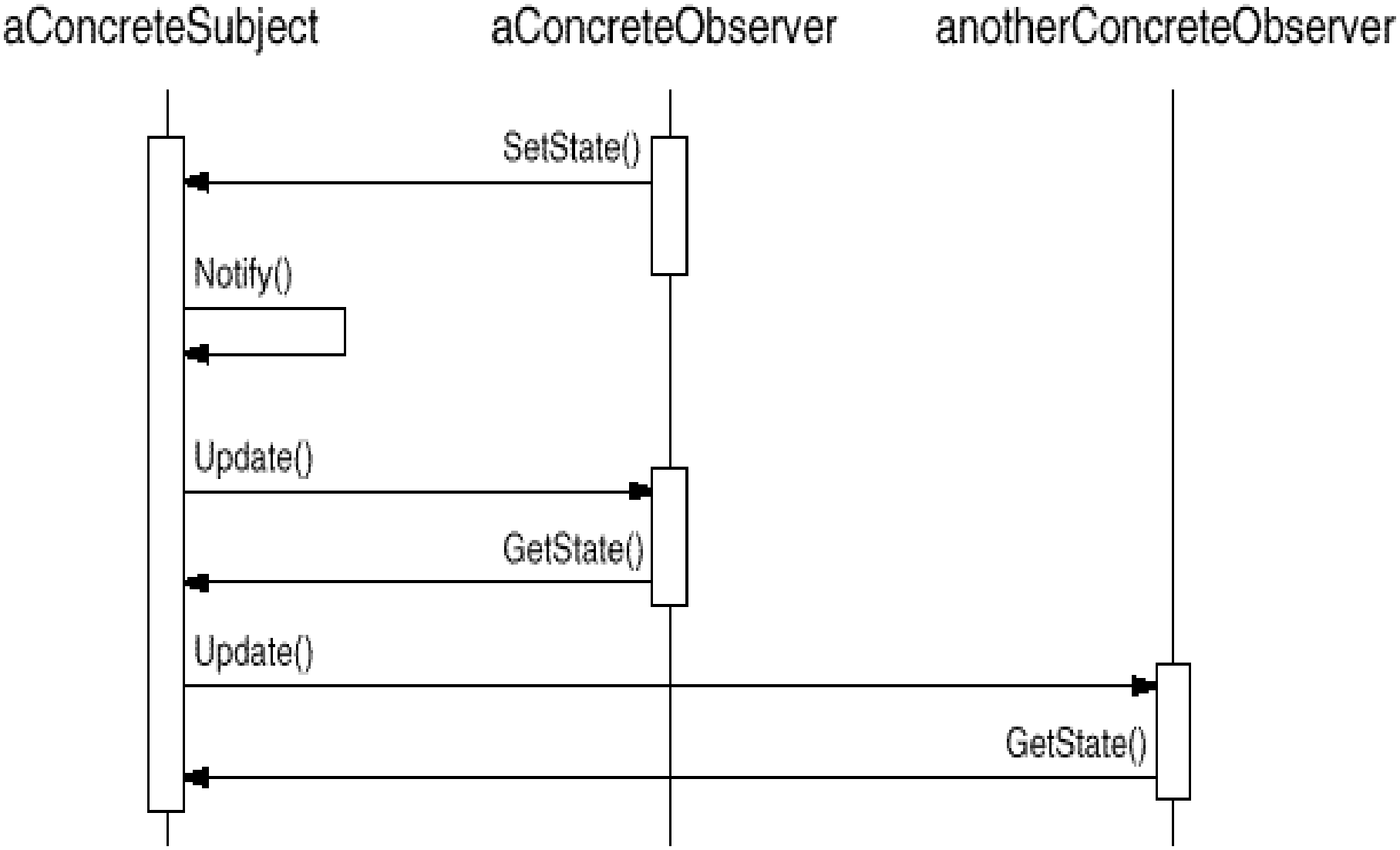
Observer

8. Collaborations

- *ConcreteSubject* notifies his observers each time its relevant state is changed (it does this by calling the base class method)

- After being notified of a change in *ConcreteSubject* state, a *ConcreteObserver* can query it to get information

Observer


aConcreteSubject
aConcreteObserver
anotherConcreteObserver
SetState()
Notify()
Update()
GetState()
Update()
GetState()

Observer

9. Consequences

- *Abstract coupling between Subject & Observer*
- *Support broadcast communication*
- *Issue – solutions?* **EXERCISE**
  - *many updates*
  - *non-relevant updates*

Observer

10. Implementation

..... **EXERCISE**

11. Code example

..... **EXERCISE**

Observer

12. Known uses

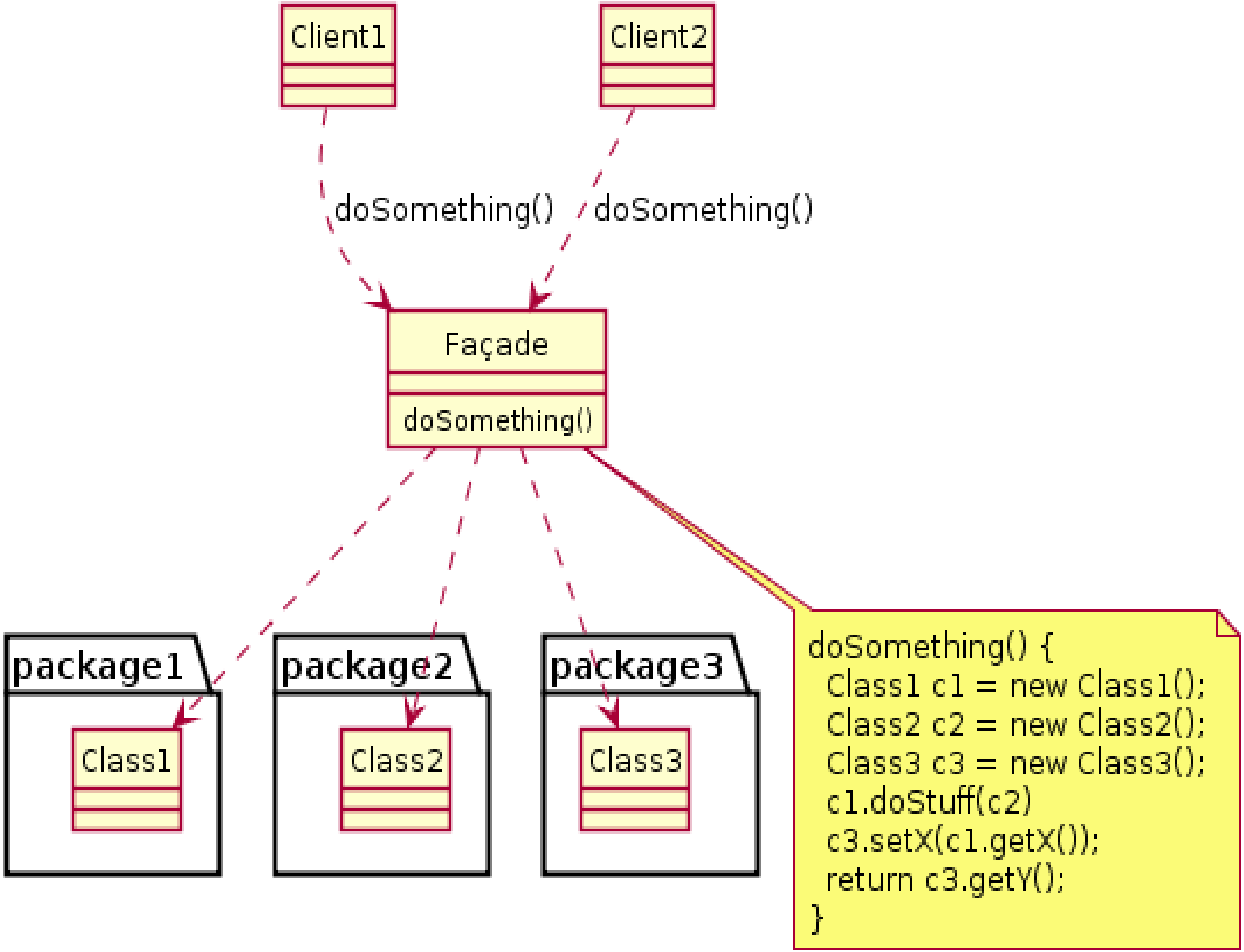
- MFC, ET++ & Smalltalk use it for separating Document (Model) and View.

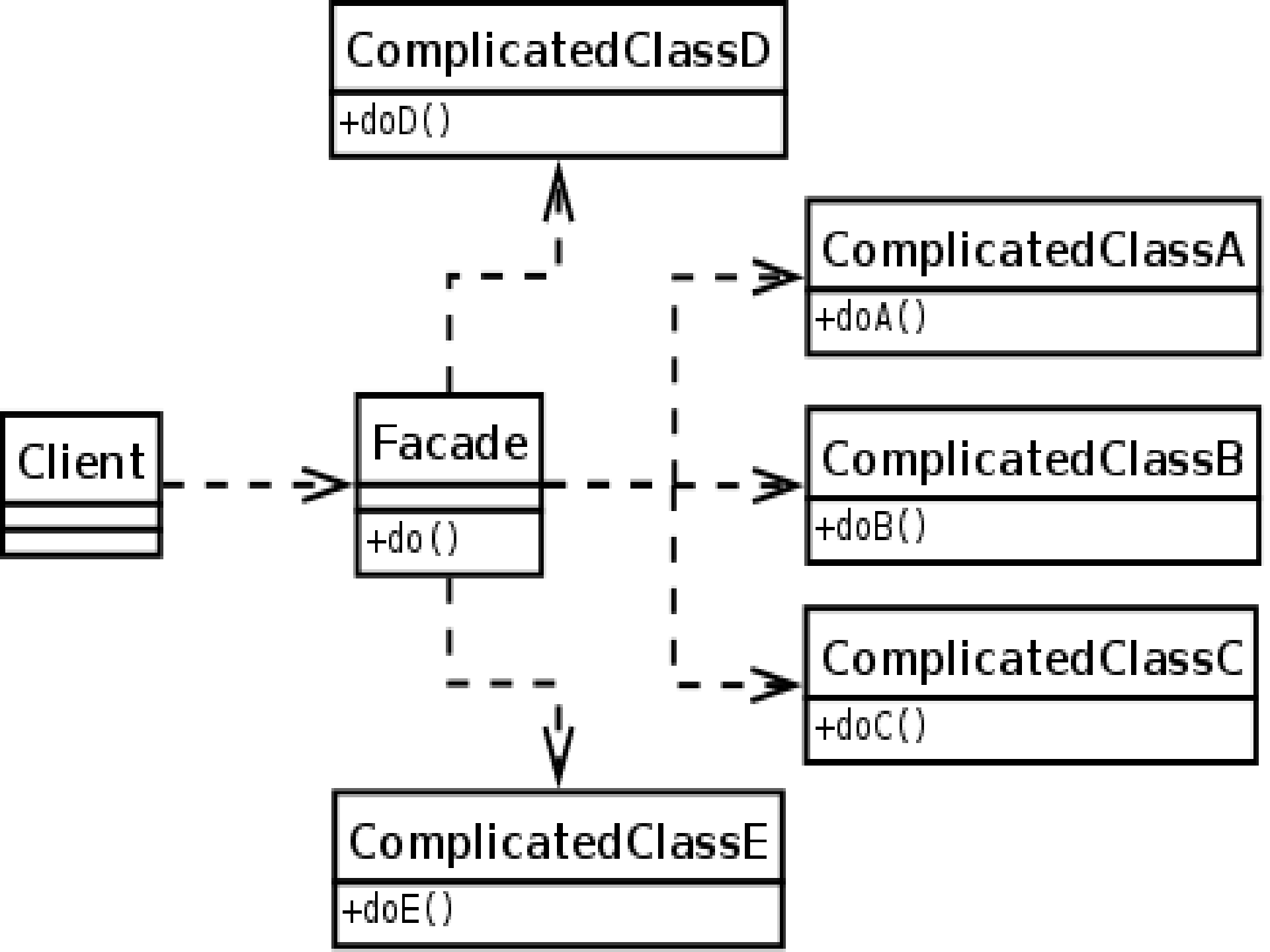
13. Related patterns

- Mediator
- Singleton

Facade

.. Isolate and encapsulate the complexity of a system, providing a simpler interface to a client

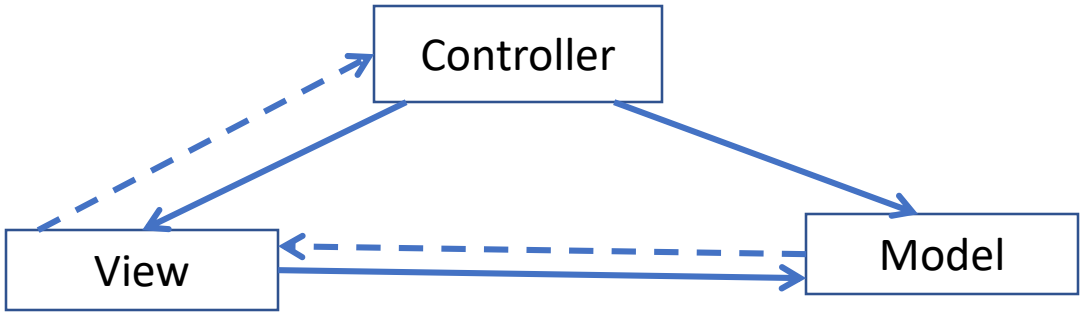

Client1
Client2
doSomething()
doSomething()
Façade
doSomething()
package1
Class1
package2
Class2
package3
Class3
doSomething() {
Class1 c1 = new Class1();
Class2 c2 = new Class2();
Class3 c3 = new Class3();
c1.doStuff(c2)
c3.setX(c1.getX());
return c3.getY();
}


ComplicatedClassD
+doD()
ComplicatedClassA
+doA()
Client
Facade
+do()
ComplicatedClassB
+doB()
ComplicatedClassC
+doC()
ComplicatedClassE
+doE()

Alt Exemplu: MVC

Model-View-Controller

Sablon de arhitectura software de tip "Observer"



Model-View-Controller

- Model: este o reprezentare a datelor specifica 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
- View:
  - reda modelul intr-o forma adecvata interactiunii, de regula un element de interfata utilizator.
  - pot exista mai multe vederi pentru un singur model.
- Controller:
  - controleaza interactiunea cu utilizatorul
  - receptioneaza intrarile utilizator
  - initiaza raspunsuri apeland operatiile obiectului (obiectelor) model, cere actualizarea vederii.

Exemplu de aplicare:

Pentru o aplicatie Web:

Model: server side

Obiectul View poate fi o paginaHTML care afiseaza datele din model si permite introducerea de date intr-un sablon prezentat utilizatorului.

Controller

- Raspunde actiunilor utilizatorului. In cazul unei aplicatii Web, o actiune utilizator este in general o cerere de pagina.
- Controller-ul va determina datele solicitate de utilizator si va solicita obiectului Model (apeland metodele adecvate)sa trasmita datele solicitate pentru afisarea lor de catre obiectul View.

Design patterns in java

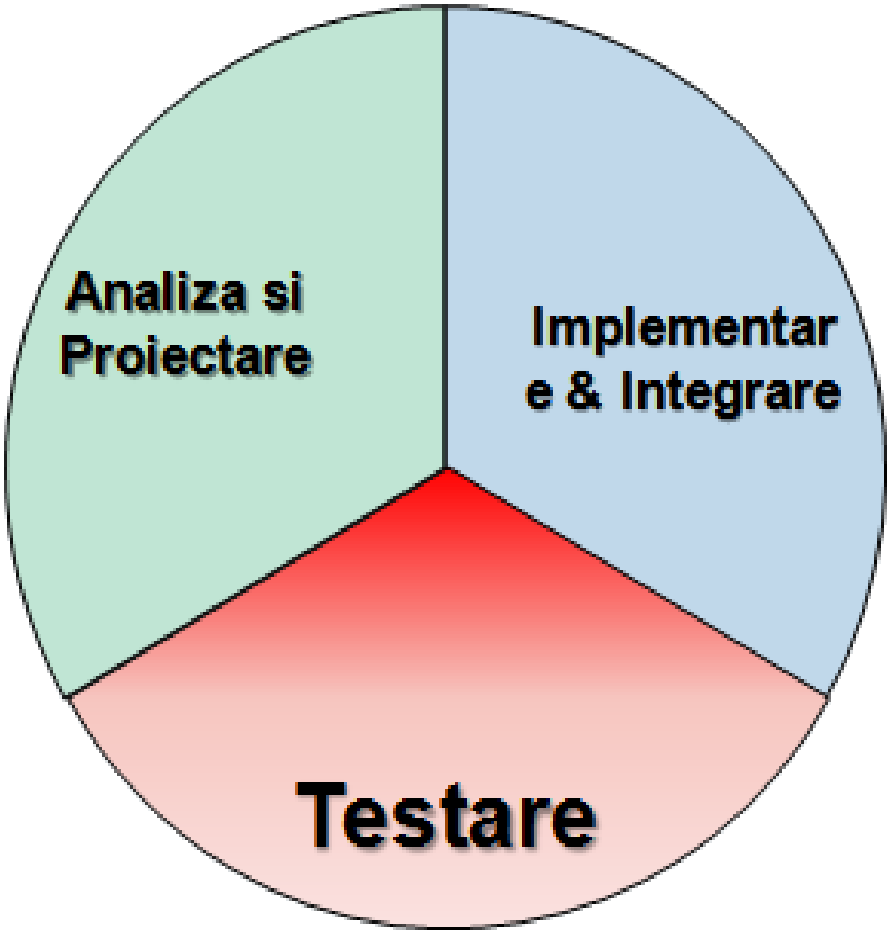
http://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries/2707195#2707195

Patterns..

- **In other software activities:**
  - ✓Analysis
  - ✓Testing
  - ✓Project management

- **Philosophical aspects:**
  - ✓Way of handling big problems (the world)
  - ✓Way of capturing, sharing and using expertise
  - ✓Way of thinking

TESTAREA PROGRAMELOR



DEFINITII

- Un caz de test este un set de date de intrare impreuna cu rezultatele pe care programul ar trebui sa le produca (comportarea asteptata a programului).
- Un test: o executie a programului pentru a exersa un caz de test – se executa programul folosind datele de intrare specificate in cazul de test si se verifica rezultatele obtinute, comparandu-le cu cele asteptate.
- O suita de test: un set de cazuri de test succesive
- Testarea este activitatea de:
  - planificare a activitatilor si a strategiei de testare,
  - conceptie a cazurilor de test
  - executie a testelor
  - evaluare a rezultatelor testelor

  in diferite etape ale ciclului de viata al unui program.

OBSERVATII

- Prin testare nu se poate demonstra corectitudinea unui program: in cele mai multe cazuri este practic imposibil sa se testeze programul pentru toate seturile de date de intrare posibile.

- Testarea poate doar sa demonstreze prezenta erorilor intr-un program: urmareste sa determine o comportare a programului neintentionata de proiectanti sau implementatori

- Tehnicile de testare sunt mult mai costisitoare decât metodele statice (reviziile)

VERIFICARE VS. VALIDARE

- Testarea in vederea verificarii programului:
  - Verificarea: Este programul conform cu specificatia sa?
  - Efectuata la mai multe nivele:

    unitate functionala, modul, componenta, subsistem, sistem.
- Testarea in vederea validarii programului, numita si testare de acceptare:
  - Validarea: Satisface programul cerintele clientului/utilizatorilor
  - Se efectueaza in mediul real de functionare a programului.
  - Prin testarea de acceptare pot fi descoperite si corectate erori, deci se efectueaza si o verificare a programului.

Abordari

- **Traditional:**
  - Scopul testarii: verificarea si validarea programelor.
  - Testare ampla la sfarsitul dezvoltarii
  - Facuta de o echipa diferita

- **Modern:**
  - Testarea ca parte integrata a dezvoltarii
  - Testele sprijina activitatile de creare si sunt distribuite relativ uniform pe parcursul dezvoltarii

TESTAREA PE PARCURSUL CICLULUI DE VIATA AL UNUI PROGRAM

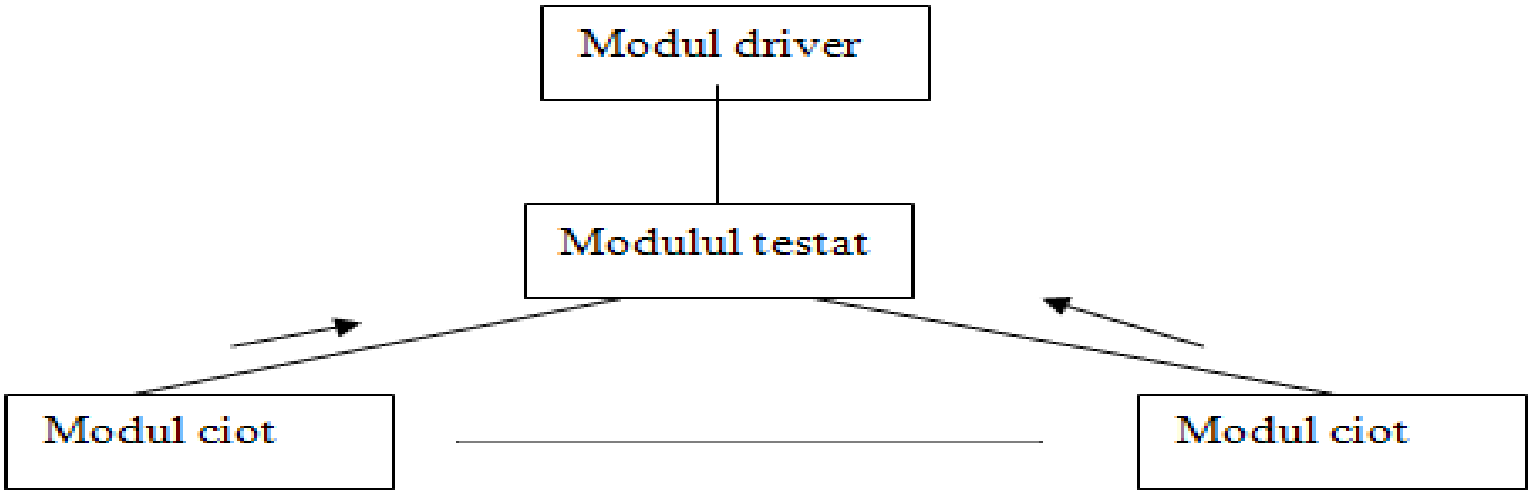
- Testele unitare
- Testele de integrare
- Testele de sistem
- Testele de acceptare

- Teste regresive

TESTELE UNITARE

- O unitate este cea mai mica parte testabila a unei aplicatii:
  - functie/procedura intr-un limbaj de programare procedurala
  - o metoda a unei clase
  - Un pachet
  - O librarie
- Testarea este efectuata de programatorul care o implementeaza, pe baza specificatiei
- In cursul testarii, unitatea este tratata ca o entitate independenta, care nu necesita prezenta altor componente ale programului
- Necesita implementarea de module suplimentare:
  - * module “stub”(ciot)
  - un modul “driver”

STRUCTURA PROGRAMULUI EXECUTABIL PENTRU TESTAREA IZOLATA A UNUI MODUL



- Datele de test pot fi:
  - generate algoritmic de modulul driver
  - preluate dintr-un fisier
  - furnizate de testor intr-o maniera interactiva.

EXEMPLU DE FUNCTIE « CIOT »
PENTRU SORTAREA UNUI VECTOR:

```
void sortare(int n, int *lista)
   { int i;
     printf(" \n Lista de sortat este:");
     for(i=0; i<n; i++)  printf("%d", lista[i]);
     printf(" \n Lista sortat este:");
     // se citeste lista sortata, furnizata de testor
     for(i=0; i<n; i++)  scanf("%d", lista[i]);
    }
```

ALEGEREA CAZURILOR DE TEST

- **Testarea functionala (“black box”)**: datele de test se aleg pe baza specificatiei unitatii testate
- **Testarea structurala (“white box)**: datele de test se aleg pe baza codului unitatii testate:
  - **Testarea fluxului controlului**
  - **Testarea fluxului datelor**
- Alte metode:
  - **Testarea random:** datele de test sunt generate in mod aleator pe baza domeniilor de valori ale variabilelor de intrare
  - **Testarea bazata pe mutatii (Mutation testing):**
    - **sunt introduse defecte in cod**
    - **o suita de test care nu detecteaza defectul introdus este considerata ineficienta**

TESTAREA CLASELOR

- **Testarea unei clase presupune:**
  - testarea separata a fiecarei functii membru
  - testarea comportarii in timp a obiectelor clasei
- **Testarea "Alpha-Omega":** trecerea unui obiect al clasei prin toate starile sale, de la creare pana la distrugere, prin apelul tuturor metodelor clasei sale cel putin odata.
- Testarea Alpha-Omega este o testare minimala a unei clase: este urmata de alte teste, bazate pe specificatia functionala a clasei sau pe acoperirea codului.
- **Ordinea de apel a metodelor unei clase depinde de tipul clasei**:
  - "Non-modal" – clasa care accepta orice mesaj in orice stare: ordinea de apel oarecare
  - "Quasi-modal" - clasa in care constrangerile privind ordinea mesajelor se schimba in acelasi timp cu starea obiectelor clasei. Exemplu: clasele de tip container sau colectie (stiva plina/goala, etc.).
    - Se doreste acoperire tip "orice mesaj in orice stare" (de ex. push() pentru o stiva plina).
  - "Modal" - clasa care are constrangeri permanente si fixe privind ordinea mesajelor.
    - Se creaza un model cu starile obiectelor si tranzitiile intre ele: diagrama de stari UML.
    - Se creaza un obiect al clasei si se apeleaza metodele intr-o ordine care sa asigure trecerea obiectului prin toate starile si toate tranzitiile ("acoperirea" tuturor starilor si a tranzitiilor).

PLATFORME DE TESTARE UNITARA

Platformele de testare unitara reduc efortul (timpul) de testare permitand:

- Definirea cazurilor de test si organizarea lor intr-o suita de testare
- Initializarea mediului de test
- Executia suitei de testare
- Inregistrarea rezultatelor executiei suitei de testare
- Prezentarea sintetica a rezultatelor executiei suitei de teste

Platforma JUnit

- JUnit: platforma (framework) de testare pentru limbajul Java (http://www.junit.org).
- **Platforme similare**: Nunit (pentru C#), PyUnit (pentru Pyton), fUnit (pentru Fortran), CPPUnit (pentru C++), s.a., denumite in mod colectiv xUnit.

  http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

TESTELE DE INTEGRARE

- Sunt dedicate verificarii interactiunilor dintre module, grupuri de module, subsisteme, pana la nivel de sistem.
- Exista mai multe metode de realizare a testelor de integrare.
- Este necesara implementarea de module "ciot" si module "driver".
- Numarul de module "driver" si de module "ciot" necesare in testele de integrare depinde de ordinea in care sunt integrate modulele.
- Testele de integrare necesita, de asemenea, instrumente de gestiune a versiunilor si a configuratiilor.

METODE DE INTEGRARE

- **Metoda "big-bang“**
  - Sunt integrate intr-un program executabil toate modulele existente la un moment dat.
  - Modulele "driver« si "ciot" necesare sunt de asemenea integrate.
  - Metoda este periculoasa caci toate erorile apar in acelasi timp si localizarea lor este dificila.
- **Integrare progresiva**
  - In fiecare pas se adauga ansamblului de module integrate numai un singur modul.
  - Erorile care apar la un test provin din ultimul modul integrat.
  - 2 metode:
    - Integrare ascendenta
    - Integrare descendenta

INTEGRAREA ASCENDENTA

- Se incepe prin testarea modulelor care nu apeleaza alte module, apoi se adauga progresiv module care apeleaza numai modulele deja testate, pana cand este asamblat intregul sistem.
- Metoda necesita implementarea cate unui modul "driver" pentru fiecare modul al programului (si nici un modul "ciot").
- Avantajele
  - Nu sunt necesare module "ciot".
  - Modulele "driver" se implementeaza mult mai usor decat modulele "ciot". Exista instrumente care produc automat module "driver".
- Dezavantajele
  - 1. Programul pe baza caruia se efectueaza validarea cerintelor este disponibil numai dupa testarea ultimului modul.
  - 2. Corectarea erorilor descoperite pe parcursul integrarii necesita repetarea procesului de proiectare, codificare si testare a modulelor.
  - 3. Principalele erori de proiectare sunt descoperite de abia la sfârsit, cand sunt testate modulele principale ale programului, ceea ce, in general, conduce la reproiectare si reimplementare.

INTEGRAREA DESCENDENTA

- Se incepe prin testarea modulului principal, apoi se testeaza programul obtinut prin integrarea modulului principal si a modulelor direct apelate de el, si asa mai departe.
- Metoda presupune implementarea unui singur modul "driver" (pentru modulul principal) si a cate unui modul "ciot" pentru fiecare alt modul al programului.
- In fiecare pas este inlocuit un singur modul "ciot" cu cel real.
- Avantajele
  - 1. Erorile de proiectare sunt descoperite timpuriu, la inceputul procesului de integrare, atunci când sunt testate modulele principale ale programului.
  - 2.Programul obtinut este mai fiabil caci principalele module sunt cel mai mult testate.
  - 3. Sistemul exista dintr-o faza timpurie a dezvoltarii si deci se poate exersa cu el in vederea validarii cerintelor – aspect foarte important in privinta costului dezvoltarii sistemului.
- Dezavantajele
  - 1. Este necesar sa se implementeze cate un modul "ciot" pentru fiecare modul al programului, cu exceptia modulului principal.
  - 2. Este dificil de simulat prin module "ciot" componente complexe si componente care contin in interfata structuri de date.
  - 3. In testarea componentelor de pe primele nivele, care de regula nu afiseaza rezultate, este necesar sa se introduca instructiuni de afisare, care apoi sunt extrase, ceea ce presupune o noua testare a modulelor.

TEHNICI HIBRIDE

tehnici hibride, euristici – minimizeaza dezavantaje prezentate anterior

- **integrarea UI + tracking/logging.**
  Aceasta permite continuarea integrarii in conditii mai bune de observare a comportarii programului.
- **integrarea modulelor critice**
  pentru a testa intens, in timp, buna lor functionare
- **Integrarea « sandwich »**
  - Se integreaza modulele de pe nivelul ierarhic cel mai coborat (cel mai mult apelate) prin tehnica « de jos in sus »
  - Se integreaza modulele de pe nivelul ierarhic cel mai ridicat prin tehnica « de sus in jos »
  - Restul modulelor se integreaza intr-o ordine convenabila.

INTEGRAREA CONTINUA

- Propusa initial de XP, utilizata in prezent in numeroase companii, indiferent de proces
- Principii:
  - Code repository
  - Regular comits (e.g. daily); each comit must build (compile) correctly
  - Build server
  - Clone production environment
  - Self testing build
  - Transparent testing results

TESTELE DE SISTEM

- Sunt teste ale sistemului de programe si echipamente complet.
- Sistemul este instalat si apoi testat in mediul sau real de functionare.
- Sunt teste de conformitate cu specificatia cerintelor software :
  - teste functionale, prin care se verifica satisfacerea cerintelor functionale
  - teste prin care se verifica satisfacerea cerintelor ne-functionale :
    - de performanta,
    - de fiabilitate,
    - de securitate, etc.
- Adesea, testele de sistem ocupa cel mai mult timp din intreaga perioada de testare.

TESTELE DE ACCEPTARE

- Sunt teste de conformitate cu produsul solicitat, conform contractului cu clientul (->Specificatia cerintelor utilizatorilor).
- Aceste teste sunt uneori conduse de client.

- Pentru unele produse software, testarea de acceptare are loc in doua etape:
  1. Testarea alfa: se efectueaza impreuna cu clientul folosindu-se specificatia cerintelor utilizator
  2. Testarea beta: programul este distribuit unor utilizatori selectionati, realizandu-se astfel testarea lui in conditii reale de utilizare.

TESTELE REGRESIVE

- Teste executate dupa corectarea erorilor, pentru a se verifica daca in cursul corectarii nu au fost introduse alte erori.
- Aceste teste sunt efectuate in timpul:
  - Dezvoltarii
  - Intretinerii
- Pentru usurarea lor este necesar sa se arhiveze toate testele efectuate in timpul dezvoltarii programului, ceea ce permite, in plus, verificarea automata a rezultatelor testelor regresive.

>> unelte pentru arhivarea si rularea automata a testelor