Ingineria programelor (Software engineering)

INTRODUCERE

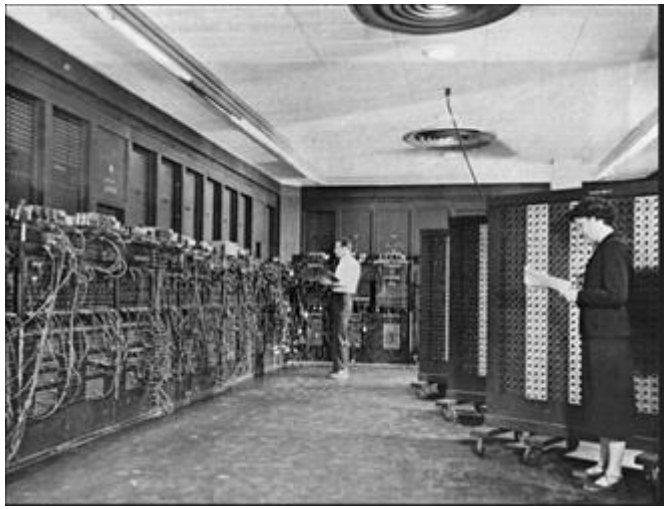
Prof. univ. dr. ing. Florica Moldoveanu

UPB, Automatică şi Calculatoare
2020-2021

Inceputurile

❖**Primul calculator electronic digital programabil**: ENIAC (Electronic Numerical Integrator And Computer) -1946

- construit la universitatea Pensilvania
- a functionat neintrerupt in perioada 1946-1955
- cântărea 27 t, avea aprox. 2,6 m x 0,9 m x 26 m, ocupa 63 m², și consuma 150 kW
- programarea + introducerea datelor: prin switch-uri de pe panourile functionale



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

❖ **Primul calculator electronic digital creat in Romania**: 1957, la IFA (Institutul pentru Fizica Atomica)

❖ **Primul dispozitiv de intrare**: cititor de cartele

❖ **Primul limbaj de programare de nivel inalt**: FORTRAN

- primul compilator de Fortran: 1954-1957, pentru IBM 704
- ultimele versiuni standard ale limbajului FORTRAN: 2003 (cu suport pentru POO), 2008, 2018.

Criza software(1)

Anii 1970 – "**Criza software**"

Cauzele:

• Evolutia rapida a tehnologiilor hardware si ieftinirea echipamentelor de calcul.

• Intelegerea beneficiilor folosirii calculatoarelor: rapiditatea obtinerii rezultatelor, acuratetea rezultatelor, s.a.

↓

• Cresterea cerinţelor de dezvoltare de programe, de dimensiune şi complexitate din ce in ce mai mari.

↓

• Necesitatea cresterii productivitatii activitatilor de dezvoltare software (timp prea mare de dezvoltare) şi a imbunatatirii calitatii produselor software (erori in functionare)

• Insuficient personal calificat pentru satisfacerea acestor cerinte.

Criza software(2)

Concluzii:

Metodele de dezvoltare software existente la acea vreme – inadecvate cerintelor

- Efortul de dezvoltare creste mai mult decat liniar raportat la dimensiunea programului.
- Programele trebuie sa poata fi usor de inteles si de adaptat de persoane diferite de cele care le-au dezvoltat.
- Programele nu sunt entitati statice, ele evolueaza in timp datorita schimbarii cerintelor si a mediului de utilizare.

Dezvoltarea de software trebuia sa devina o industrie de sine statatoare.

Grady Booch: The net result is that building and maintaining software is hard and getting harder; building **quality software** in a **repeatable** and **predictable** way is harder still.

Software Engineering(1)

➢A devenit necesara o **disciplina** care sa furnizeze cadrul pentru construirea de software de calitate în mod repetabil şi predictibil.

➢ **Software Engineering (Ingineria programelor)**

Termenul: *dezvoltarea programelor trebuie să fie fundamentată teoretic și ghidata de metode validate în practică, la fel ca alte domenii inginerești.*

➢**Scopul disciplinei:** definirea de tehnici de "fabricatie" justificate de teorie sau de practica →

poate fi software-ul tratat ca alte produse ingineresti?

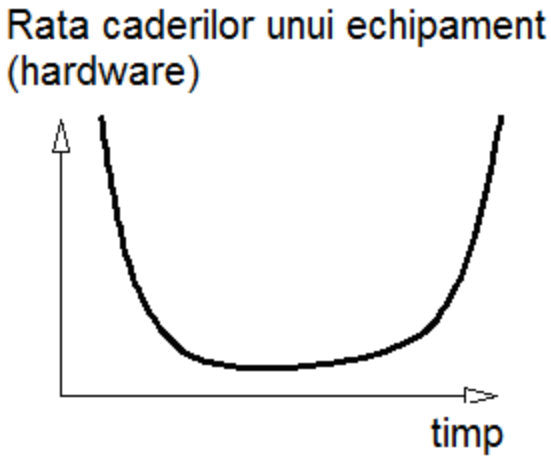
Software-ul se deosebeste de alte produse, care sunt fabricate:

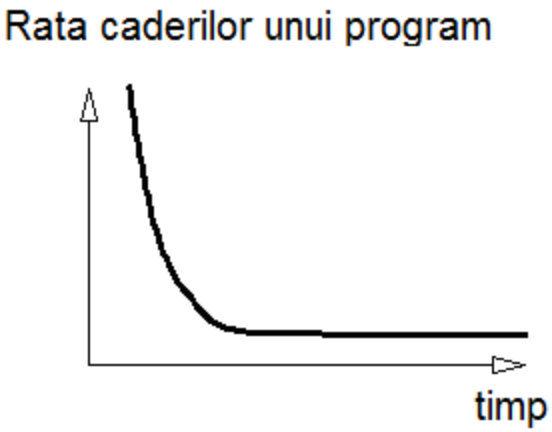
- nu este un produs fizic
- un produs hardware este dezvoltat (conceptie, proiectare, constructie prototip, testare prototip), apoi fabricat in serie
- un produs software este doar dezvoltat nu si fabricat; nu exista un proces de fabricatie software
- programele nu pot fi complet "asamblate" din componente – orice program nou trebuie sa raspunda la cerinte specifice

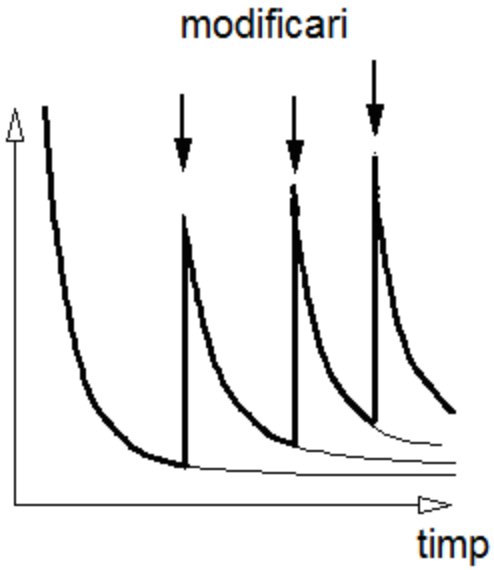
Software Engineering(2)

- un produs hardware nu poate fi modificat pe parcursul utilizarii sale pentru a fi adaptat la cerinte noi

- software-ul nu “îmbătrâneşte”







Software Engineering si crizele software (1)

- Prima criza software: perioada ‘60 – ‘70
  - Cauza: Necesitatea de a dezvolta programe de dimensiuni mari, intr-un timp mai scurt, mai fiabile
  - Rezolvare: Trecerea de la limbaje de asamblare la limbaje de programare de nivel inalt (Fortran, C – începutul anilor ‘70)
- A 2-a criza software: perioada ‘80 – ‘90
  - Cauza: Incapacitatea de a dezvolta programe de dimensiuni foarte mari, complexe, cu participarea unui numar mare de programatori
  - Rezolvare: - Trecerea la limbaje de programare orientate obiect (C++, Java, C#)
    - Instrumente de dezvoltare mai puternice
    - Noi metodologii de dezvoltare: şabloane de proiectare, sabloane arhitecturale, revizia specificatiilor, a testelor, a codului
- A 3-a criza software: dupa 2010
  - Cauze: - Performanta procesarii secventiale a devenit nesatisfacatoare
    - Necesitatea stocarii si procesarii de volume de date mari
  - Rezolvare: Noi modele de programare, paralelizare, noi limbaje de programare, standardizare, instrumente pentru paralelizare si testare, ş.a.

Software Engineering - definitie

- Una dintre definitiile termenului “Software Engineering” (IEEE Standard Glossary of Software Engineering Terminology,” IEEE std 610.12-1990):

  Software Engineering este:

  “Aplicarea unei abordări sistematice, disciplinate şi măsurabile în dezvoltarea, operarea şi întreținerea software-ului, adică aplicarea ingineriei pentru software. De asemenea, studiul unor asemenea abordări.”

- **Produs software =**

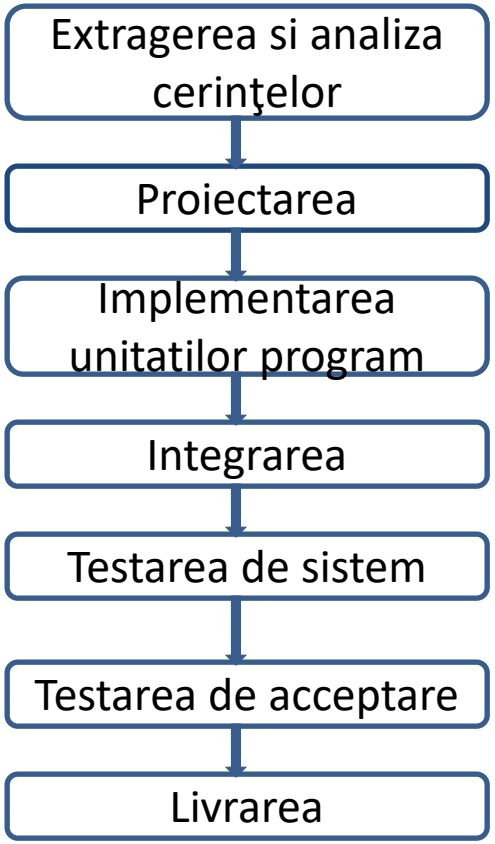
  cod sursa, cod executabil, biblioteci

  +

  documentaţii (de specificare a cerintelor, de proiectare, de instalare, de utilizare)

Activităţi în dezvoltarea produselor software

- Activităţi tehnice:



Specifici **CE** îţi propui sa faci, pentru a satisface cerintele clientului/viitorilor utilizatori

Decizi **CUM** vei face

Creezi mai întâi părţile

Asamblezi părţile pentru a obtine produsul

Verifici că produsul rezultat functioneaza asa cum ti-ai propus (cum ai specificat)

Verifici împreună cu clientul şi utilizatori finali ca produsul satisface asteptarile lor

Instalezi produsul în mediul său de operare

- Activităţi de asigurare a calitatii
- Activităţi de management al proiectului de dezvoltare software

Extragerea si analiza cerintelor (1)

- ***Extragerea cerinţelor*: identificarea si definirea cerintelor utilizator**
- **Cerintele utilizator**
  - Descriu punctul de vedere al utilizatorilor: CE doresc viitorii utilizatori de la viitorul produs.
  - Sunt cerinte de: functionare, performanta, securitate, interfata utilizator, interfete de comunicatie, s.a.
  - Definite în ***Documentul cerintelor utilizator (User Requirements Document) – URD***
  - In URD cerintele sunt exprimate in limbaj natural si folosind notatii grafice usor de inteles de client si viitorii utilizatori.
  - URD devine parte din contractul cu clientul.
  - URD este documentul de referinţă pentru testarea de acceptare.

Extragerea si analiza cerintelor (2)

- ***Analiza cerinţelor :***

  - rezolvarea posibilelor conflicte intre diferite cerinte utilizator,

  - prioritizarea cerintelor,

  - rafinarea cerintelor utilizator prin exprimarea lor într-o manieră mai formală

- Rezultatul analizei cerintelor:

  – **specificat într-un document** numit, de regula

  Documentul Cerintelor Software (Software Requirements Document) – **SRD**

  - adesea, SRD contine si

    - cerintele utilizator – caz in care nu mai este necesar URD

    - cerinte de sistem (hardware)

➢ **Documentul de specificare a cerinţelor (SRD):**

- se foloseste pentru estimarea costurilor dezvoltarii si pentru planificarea dezvoltarii;

- este documentul de referinţă pentru testarea de sistem si testarea de acceptare

Proiectarea (1)

- ***Proiectarea arhitecturală*** **(numită şi *proiectarea de sistem*)**
  - Se stabileste arhitectura hardware-software a sistemului care va implementa cerinţele: se definesc **subsistemele şi interfeţele lor, se decide asupra tehnologiilor software care vor fi folosite in implementare şi asupra componentelor hardware ale sistemului.**
  - Toate cerintele specificate trebuie sa fie acoperite de arhitectura sistemului.
  - Se alege solutia optima dintre alternativele posibile de implementare a cerintelor.
  - Rezultat: **ADD** (Architectural Design Document); contine: descrierea subsistemelor şi a interfeţelor lor, repartitia artefactelor software la momentul executiei pe echipamentele sistemului.
  - Diferitele subsisteme definite în ADD sunt implementate de echipe diferite:
    - ADD se foloseste pentru specificarea testelor de integrare.

Proiectarea (2)

•***Proiectarea de detaliu***

– Se descompun subsistemele pana la nivel de module (unităţi program) care se implementeaza in limbajul de programare ales (functii, clase).

– Rezultat: **DDD** (Detailed Design Document); contine: descrierea interfetei si a rolului fiecarui modul, specificatia functionala a fiecarui modul, comunicatia intre module, procesele din sistem şi sincronizarea lor, s.a.

– DDD se foloseste pentru definirea testelor unitare

Implementarea modulelor

Implementarea cuprinde <u>codificarea si testarea separata a modulelor</u> definite in etapa de proiectare de detaliu.

- Este cea mai bine stapanita si cea mai bine “utilata” dintre toate activitatile de dezvoltare software. Exista:
  - Limbaje de programare, compilatoare, instrumente de localizare a surselor de erori (debbug-ere) – integrate in medii de dezvoltare
  - Instrumente software pentru testarea automata
- Reprezinta circa 15-20% din efortul total de dezvoltare a unui produs program.

Integrarea si testarea

Integrarea

- Modulele care au fost codificate şi testate independent sunt integrate treptat in subsisteme, pana la nivel de sistem.
- Se verifica interactiunea si comunicarea intre componentele integrate prin **teste de integrare.**
- Rezultatul integrarii: o versiune executabila a produsului software

Testarea de sistem

- Se verifica daca produsul obtinut (sistemul hard-soft) **satisface toate cerintele specificate**
- Efectuata in interiorul companiei care dezvolta produsul software.

Testarea de acceptare

- Se verifica impreuna cu clientul daca sistemul **satisface cerintele utilizator specificate**
- Produsul software este instalat si testat in mediul in care va opera
- Efectuata de o echipa de testare independenta care include clientul si viitori utilizatori
- Testare alfa/beta; testarea alfa: testarea efectuata impreuna cu clientul.
- Testarea beta: produsul este distribuit unor utilizatori selectati, care vor raporta eventualele caderi produse in timpul utilizarii.

Operarea si Intretinerea

- **Operarea: - utilizarea efectiva a software-ului in mediul real de functionare**

  - începe dupa livrarea software-ului la client

- **In timpul operarii pot apărea necesităţi de modificare:**
  - Corectie defecte (bug-uri) descoperite in timpul operarii
  - Imbunatatire functii existente
  - Adaptare la noi tehnologii
  - Adaugare noi functii

- **Intreţinerea (mentenanţa): efectuarea modificarilor necesare în timpul operarii.**

- poate fi efectuata de catre echipe / persoane diferite de cele care au participat la dezvoltarea produsului software;

- poate necesita efectuarea oricarei activitati tehnice din perioada dezvoltarii;

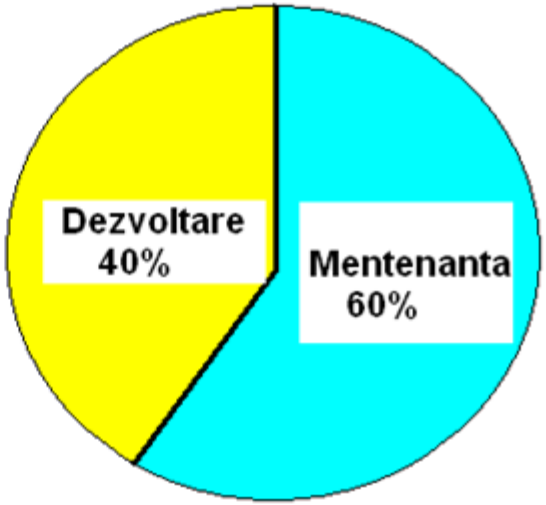
- costul întretinerii depinde de calitatea proiectarii, a documentarii şi testarii.

Distributia efortului
pe parcursul vieţii unui produs software

Viața unui produs software = perioada de dezvoltare + perioada de operare

Dezvoltare

Dezvoltare/Mentenanta



Asigurarea calitatii

- **Scop: asigurarea cerintelor tehnice si a standardelor de calitate pentru**
  - procesul de dezvoltare
  - produsul final
- **Activitati:**
  - Alegerea modelului de dezvoltare
  - 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 artefactelor procesului de dezvoltare, definirea instrumentelor de masurare

Managementul proiectului (1)

Activitati:

- Scrierea propunerii pentru obtinerea proiectului de dezvoltare a produsului software
- Estimari asupra necesarului de resurse umane pentru dezvoltarea unui produs software
- Decizii asupra (re-) alocarilor de resurse umane pe activitati si etape ale dezvoltarii
- Stabilirea si planificarea etapelor de dezvoltare
- Revizii ale documentelor elaborate pe parcursul dezvoltarii
- Selectia si evaluarea personalului
- Scrierea si prezentarea de rapoarte pe parcursul procesului de dezvoltare
- Studiul de fezabilitate

Managementul proiectului (2)

Studiul de fezabilitate

- Trebuie sa preceada initierea proiectului software

- Se determina obiectivele proiectului

- Se analizeaza procesul care ar trebui imbunatatit/ produsul pe care ar trebui sa-l inlocuiasca noul produs software

- Se estimeaza costurile

- Pe baza studiului de fezabilitate se poate decide că:

- Proiectul poate sa inceapă (este realizabil)

- Proiectul se abandoneaza:

    - Realizarea produsului software nu aduce beneficii clientului (*Return on Investment - ROI*)

    - Costurile de dezvoltare sunt mai mari decat finantarea oferita de client

    - Realizarea nu este posibila cu echipamentele pe care le are/vrea clientul

    - Etc.

Riscurile unui proiect software

De ce poate eşua un proiect software?

- **Factori de experienta:**
  - a managerului
  - a echipei
  - a organizatiei in care se dezvolta produsul
- **Factori de planificare:**
  - estimarea necesarului de resurse umane
  - estimarea perioadelor de timp pentru diferite activitati
  - definirea responsabilitatilor
- **Factori tehnologici:**
  - noutatea tehnologica (tehnologii inca nevalidate in practica)
  - metodele de dezvoltare
  - instrumentele de dezvoltare
- **Factori externi:**
  - calitatea specificatiei cerintelor: specificatia cerintelor incompleta, inconsistenta, ambigua
  - stabilitatea cerintelor (cerintele se modifica pe parcursul dezvoltarii)
  - stabilitatea si disponibilitatea altor factori de care depinde proiectul: mediul de utilizare, intarzieri in achizitionarea unor echipamente necesare in dezvoltare, s.a.

MODELE DE DEZVOLTARE SOFTWARE - 1

Prof. univ. dr. ing. Florica Moldoveanu

Ingineria programelor

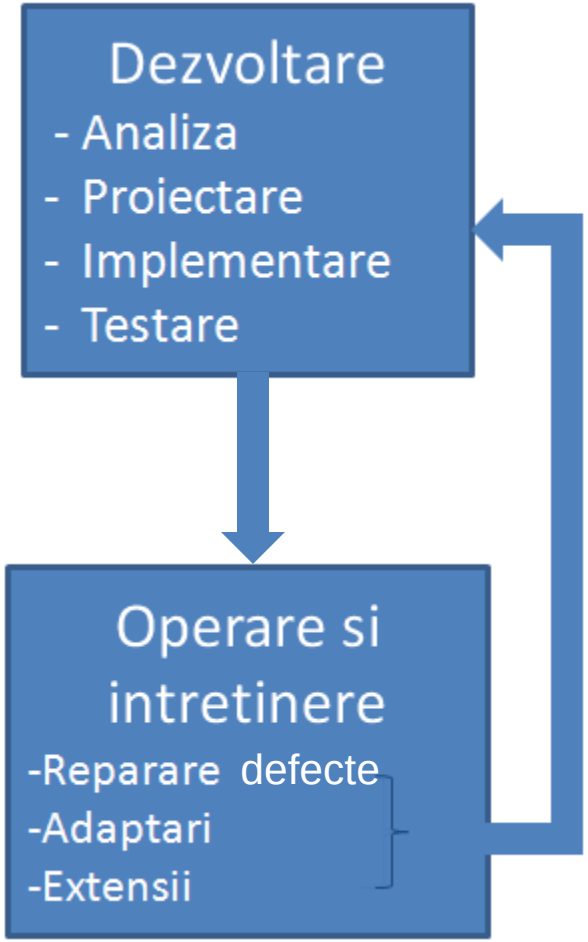
UPB, Automatică şi Calculatoare

2020-2021

Ciclul de viata al unui program

(Software life cycle)

- O secventa ciclică de etape in existenta programului, care include toate activitatile necesare pentru dezvoltarea, operarea si intretinerea sa.



Modele ale procesului de dezvoltare software

Software development life cycle (SDLC) models

Model de dezvoltare: descriere generală a procesului de dezvoltare (ghid)

Scopuri:

- Optimizarea procesului de dezvoltare, pentru **obtinerea de produse de calitate, in mod reproductibil**, in limitele costurilor si termenelor de livrare prevazute.
- Minimizarea riscurilor de eşec al proiectului.

Doua abordari distincte:

- **Dezvoltarea liniară (într-un singur ciclu de dezvoltare):**

  - Modelul cascadă (Waterfall), modelul în V, dezvoltarea pe baza de prototip.

Dezvoltarea iterativă şi incrementală (în mai multe cicluri de dezvoltare):

  - RUP (Rational Unified Process), modele agile (SCRUM, XP, AUP şi altele).

Modelul “In cascada” (Waterfall) - Clasic si cel iterativ - (1)

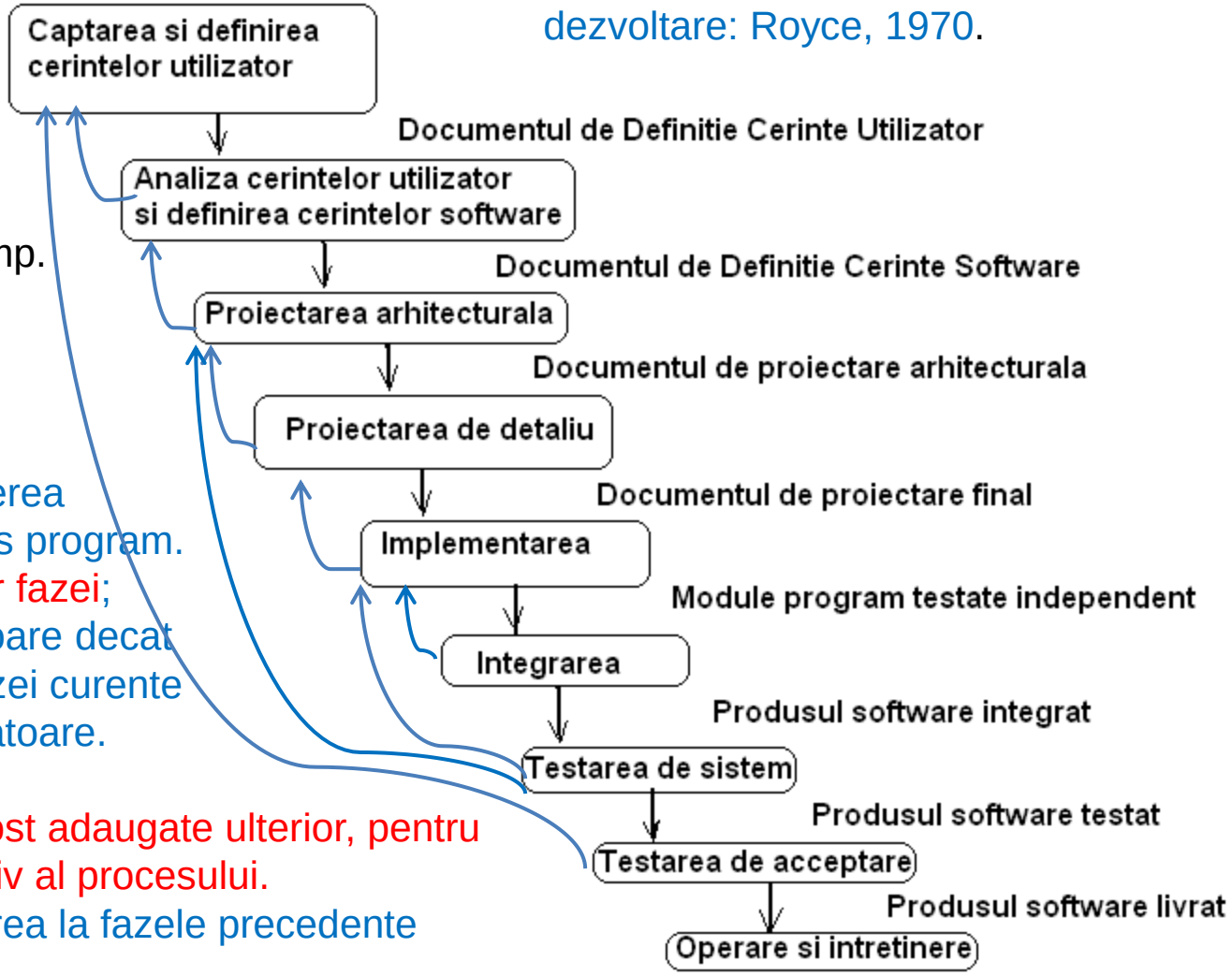
- Procesul de dezvoltare: succesiune de faze ce se înlănțuie liniar, de la analiza cerintelor pana la livrarea la client.

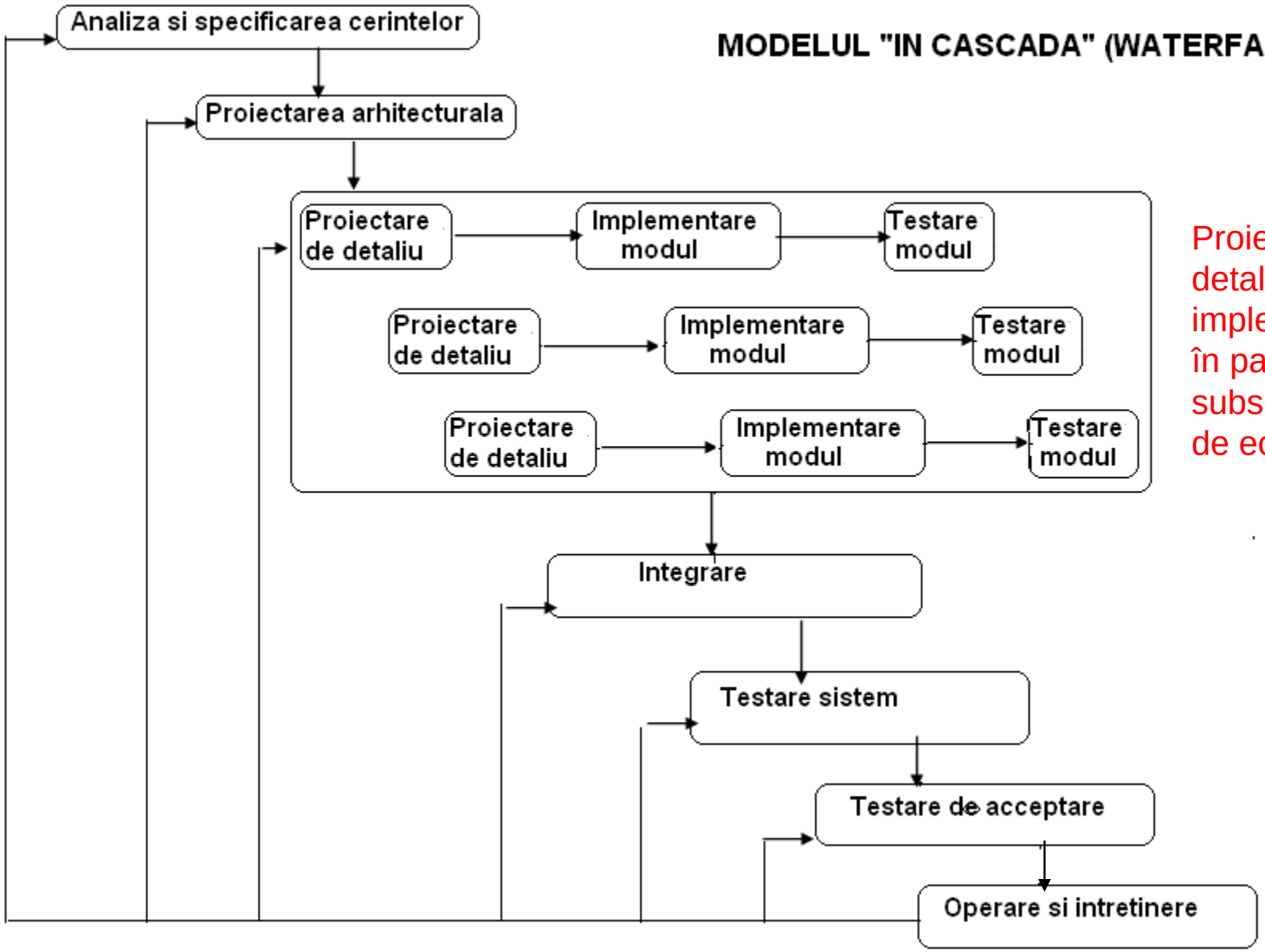
-Fazele sunt planificate în timp.

-Fiecare faza:
- corespunde unei activitati
- trebuie sa se termine la o anumita data prin producerea unui document sau produs program.
- include revizia rezultatelor fazei;
- nu se trece la faza urmatoare decat atunci cand rezultatele fazei curente sunt considerate satisfacatoare.

- Sagetile ascendente au fost adaugate ulterior, pentru a reflecta caracterul iterativ al procesului.
- NU se incurajeaza revenirea la fazele precedente

- Primul model de dezvoltare: Royce, 1970.





Proiectarea de detaliu si implementarea în paralel a subsistemelor, de echipe diferite

Modelul “In cascada” (Waterfall) (3)

Avantaje:

- Produsul software este bine documentat.
- Permite un bun management al proiectului:
  - –planificarea resurselor umane pe etape
  - –estimari de cost destul de exacte

Dezavantaje:

- Un produs executabil, care sa demonstreze functionarea sistemului este disponibil tarziu → risc crescut de neacceptare a produsului
- Multe defecte sunt descoperite tarziu; in special cele de proiectare → cost crescut (sunt reluate activitatile de proiectare, implementare, testare).
- **Inflexibil la modificarea cerintelor pe parcursul dezvoltarii**
- Toate **riscurile** sunt incluse intr-un singur ciclu de dezvoltare

Modelul “In cascada” (Waterfall) (4)

Nu se recomanda pentru:

• Proiectele în care cerintele sunt vagi, neclare

• Proiectele în care cerintele se pot schimba pe parcursul dezvoltarii

• Proiectele de lunga durata

Adecvat pentru:

• Proiectele în care cerintele sunt bine întelese de la început şi nu se modifica pe parcursul procesului de dezvoltare.

• Experienta ultimelor decenii a demonstrat ca modelul este valoros.

• Este utilizat şi în prezent de multe comanii mari pentru anumite tipuri de produse software.

MODELE DE DEZVOLTARE SOFTWARE -2

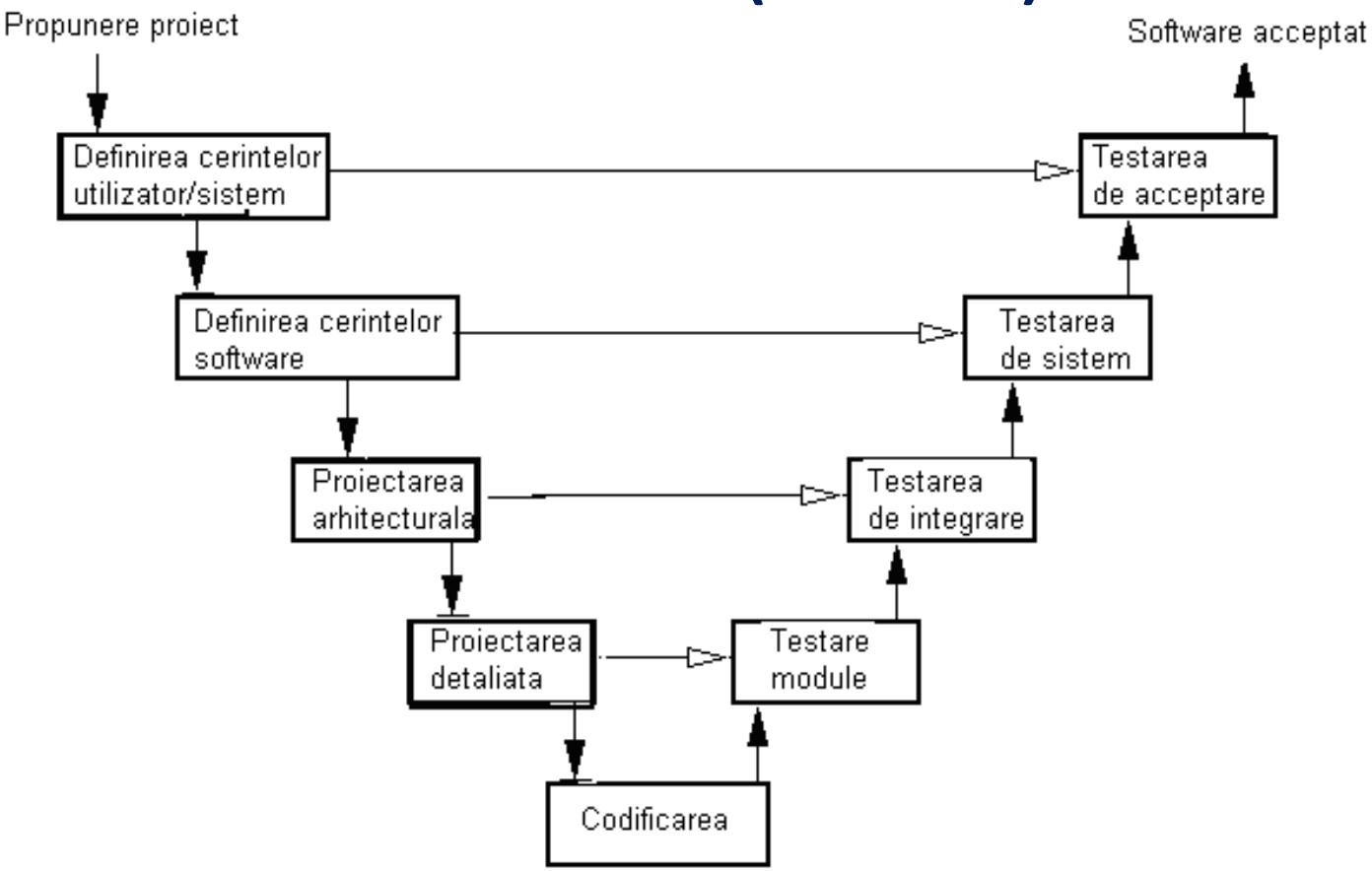
Prof. univ. dr. ing. Florica Moldoveanu

Ingineria programelor

UPB, Automatică şi Calculatoare

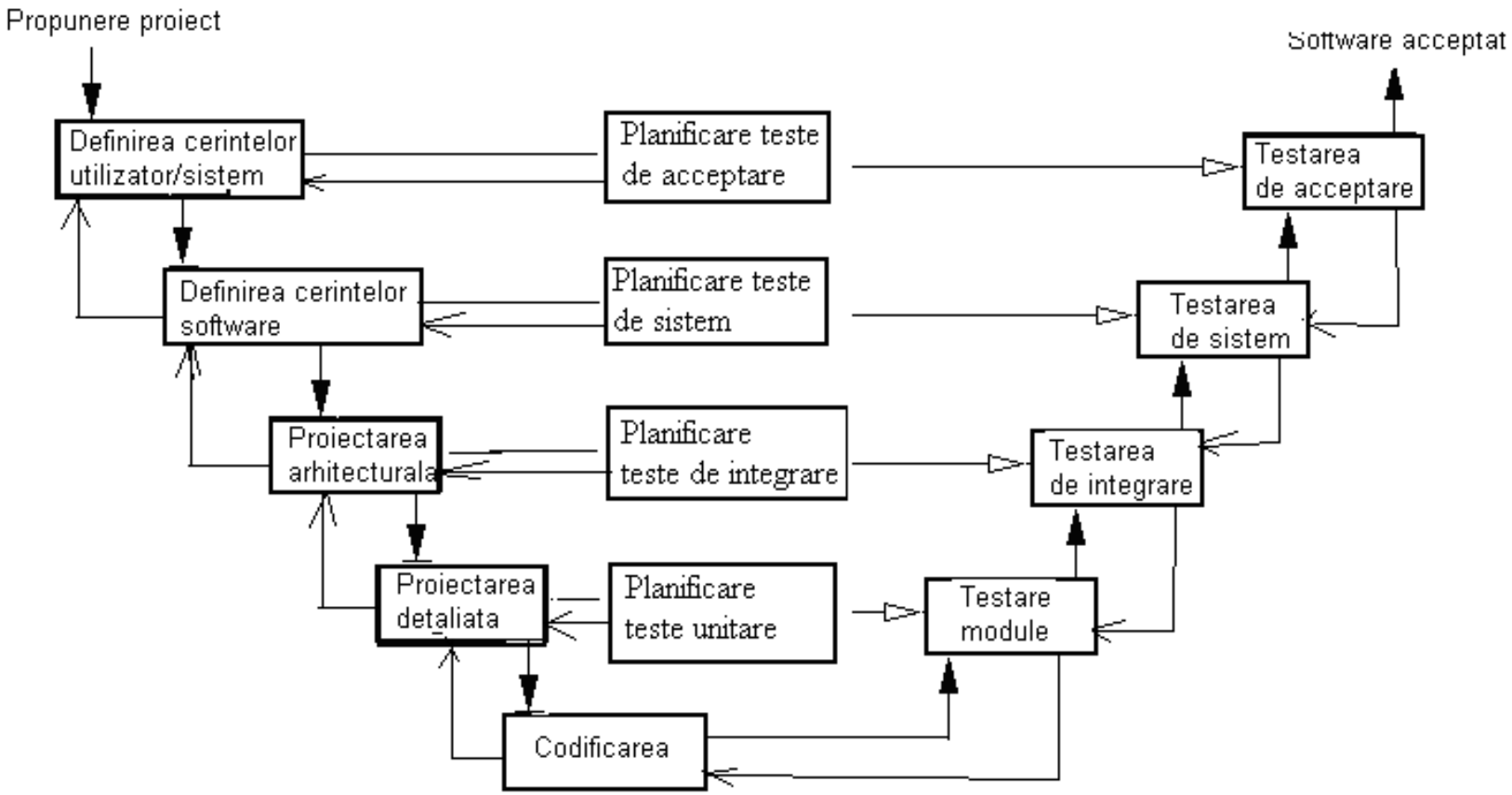
2020-2021

Modelul “In V” (V model)



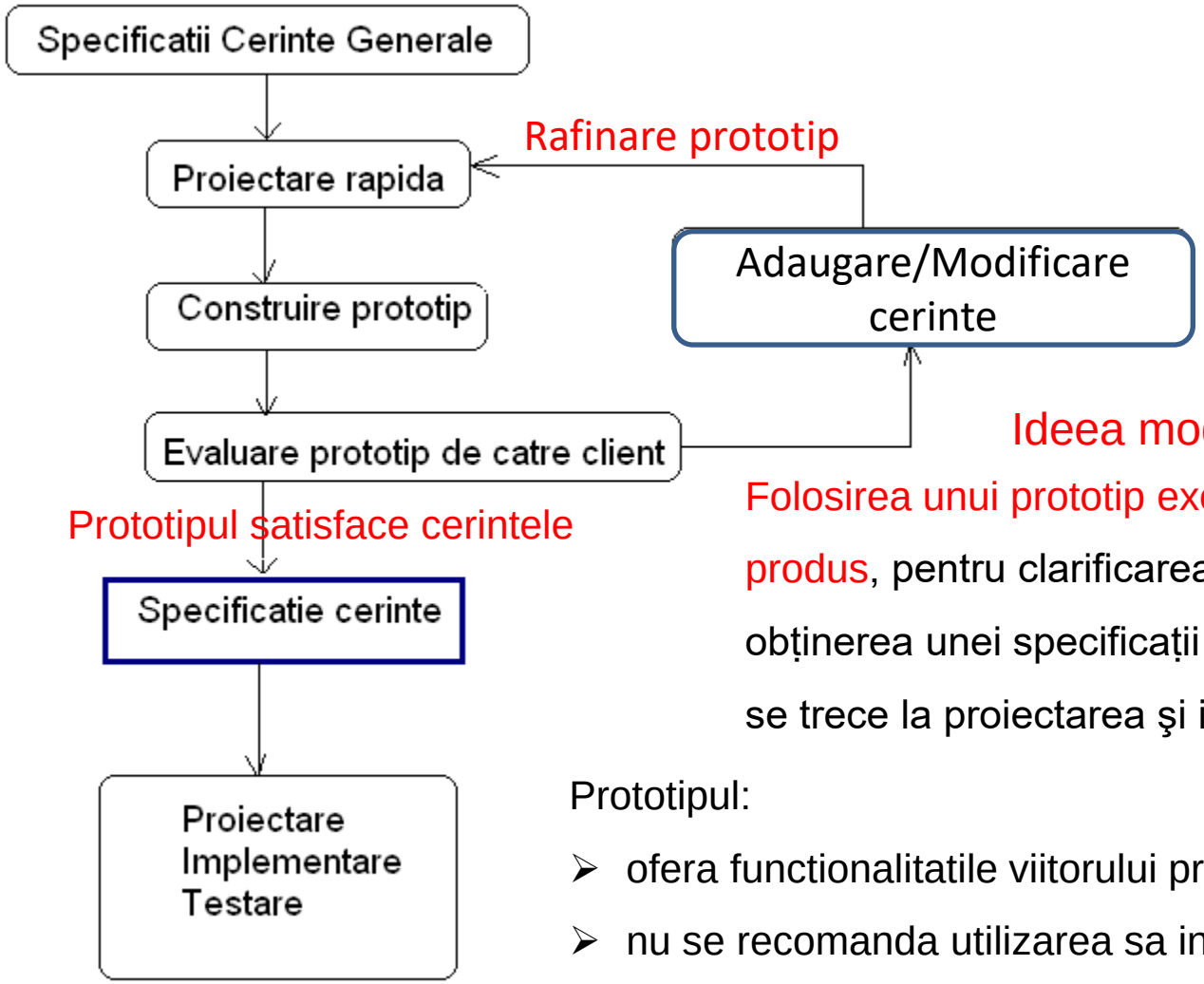
- Este o varianta a modelului cascada, care include elaborarea planurilor de test in fazele de specificare si proiectare.
- Săgețile orizontale evidențiază corelarea dintre etapele de creare a artefactelor software (documente, cod) şi cele de verificare.

MODELUL "IN V" detaliat



Avantaj fata de modelul cascada: şanse de succes mai mari datorita elaborarii planurilor de test în primele etape ale procesului de dezvoltare.

Dezvoltarea pe baza de prototip



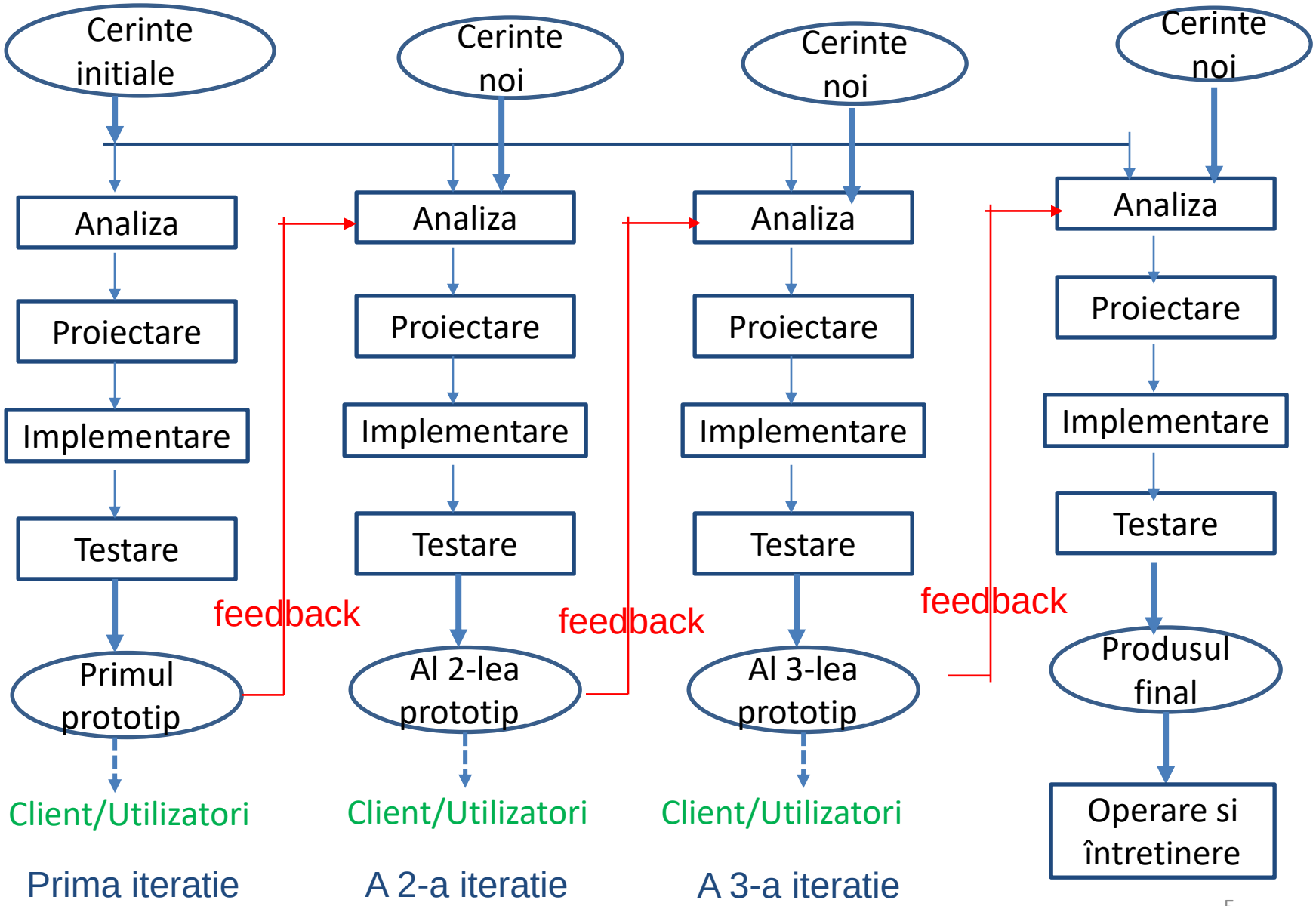
Ideea modelului:

Folosirea unui prototip executabil al viitorului produs, pentru clarificarea cerintelor utilizator şi obținerea unei specificații a cerințelor înainte de a se trece la proiectarea şi implementarea produsului

Prototipul:

- ofera functionalitatile viitorului produs si interfata utilizator
- nu se recomanda utilizarea sa in dezvoltarea produsului final, chiar daca este implementat in acelasi limbaj

Modelul Iterativ si Incremental (1)



Modelul Iterativ si Incremental (2)

Caracteristici generale

- Produsul software este dezvoltat in mai multe iteratii.
- Fiecare iteratie include un “ciclu de dezvoltare cascada” si se incheie cu un produs executabil care poate fi livrat clientului.
- In fiecare iteratie se adauga noi functionalitati (cerinte) la ultimul prototip:
  - Cerinte prevazute initial
  - Cerinte noi – aparute pe parcursul iteratiei anterioare
- Analiza cerințelor într-o iterație include feedback-ul la utilizarea prototipului anterior.

Modelul Iterativ si Incremental (3)

Iteratiile

❖ Scopul fiecarei iteratii este de a produce un produs executabil prin care se poate demonstra partilor interesate în proiect o parte dintre functionalitatile viitorului produs.

❖ Durata unei iteratii depinde de modelul de dezvoltare şi tipul de proiect; poate fi de cateva saptamani, una sau mai multe luni.

❖ Cu cat o iteratie este mai scurta cu atat mai repede se obtine feedback-ul partilor interesate.

❖Pentru prima iteratie se alege un subset de cerinte care corespund cazurilor de utilizare principale ale produsului (pun in evidenta aspectele cheie ale produsului).

❖ In fiecare iteratie se implementeaza un subset de cerinte prevazute initial dar si cerinte noi (neprevazute initial), tinand cont si de feedback-ul partilor interesate in proiect: client, utilizatori finali, alții.

❖ Obiectivele unei iteratii se stabilesc pe baza evaluarii iteratiilor precedente.

Modelul Iterativ si Incremental (4)

Iteratiile (cont)

❖ In fiecare iteratie se creaza un nou prototip prin modificarea prototipului anterior → proiectare, codificare, testare.

➢ Arhitectura produsului software trebuie sa fie flexibila la schimbari (extensibila, sa nu presupuna modificarea sa la fiecare noua iteratie)

❖ Analiza intr-o iteratie trebuie sa includa evaluarea evolutiei produsului: evolutia numarului de defecte descoperite in fiecare nou prototip, a complexitatii codului si a arhitecturii, a efortului de actualizare.

➢ Metricile sunt suportul pentru determinarea eficientei procesului si a calitatii produsului: sunt importante nu numai valorile lor absolute ci si evolutia lor in timp.

❖Documentatia este construita treptat, in timpul fiecarei iteratii.

Modelul Iterativ si Incremental (5)

Dezvoltarea iterativă şi incremetală poate fi:

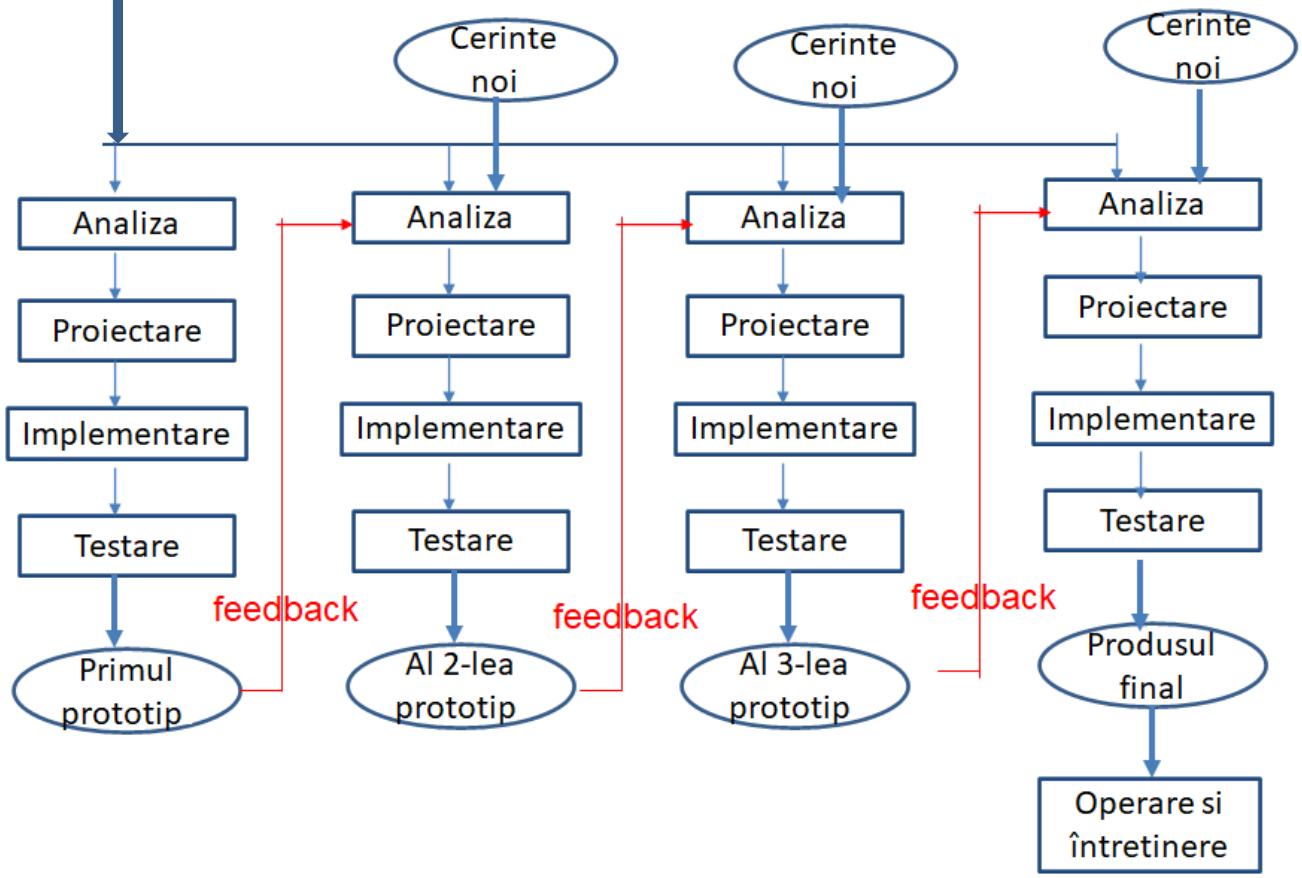
- Proces planificat – Exemplu: RUP (Rational Unified Process)

  Inainte de începerea iterațiilor se efectueaza o analiza de nivel inalt:

  - Se analizeaza: scopul sistemului, obiectivele de nivel înalt, viitorii utilizatori, principalele functionalitati si constrangeri de operare
  - Se definesc cerintele de nivel inalt ale sistemului
  - Se planifica iteratiile si se aloca cerintele pe iteratii
  - Se defineste o arhitectura software de nivel inalt
  - Durata iteratiilor nu este fixa: saptamani – luni, in functie de activitatile planificate
- Proces "agil" – Exemple: SCRUM, FDD (Feature-Driven Development), XP, ş.a.
  - Nu se pune accentul pe planificarea iteratiilor, cerintele sunt colectate pe parcurs
  - Iteratiile sunt foarte scurte (1-2 saptamani), indiferent de proiect
  - Se pune accentul pe feedbackul clientului/utilizatorilor
  - Clientul este implicat in procesul de dezvoltare (stabilirea obiectivelor urmatoarei iteratii, prioritizarea cerintelor)
  - Documentaţie minimală

Dezvoltarea iterativa – process planificat (1)

Analiza de nivel inalt
-Se defineste o "viziune" asupra sistemului
-Se analizeaza "domeniul problemei"/mediul de operare
-Se definesc cerintele de nivel inalt ale sistemului
-Se planifica iteratiile si se aloca cerintele pe iteratii
-Se defineste o arhitectura software de nivel inalt



Dezvoltarea iterativa – process planificat (2)

Analiza de nivel inalt

❖ **Se defineste o "viziune" asupra sistemului**: scopul său, obiectivele de nivel înalt, cine îl va utiliza, principalele functionalitati si constrangeri de operare.

❖ In unele cazuri, **se analizeaza "domeniul problemei"** (business modeling). Scopul:

-de a se intelege structura si dinamica organizatiei tinta (in care va fi implementat sistemul), problemele din organizatia tinta si imbunatatirile potentiale;

-de a se asigura o aceeasi intelegere a organizatiei tinta de catre client, utilizatorii finali si dezvoltatori;

-de a se deriva cerintele de sistem in scopul sprijinirii organizatiei tinta

❖ **Se definesc cazurile de utilizare ale viitorului sistem si actorii** (cei care vor interactiona cu sistemul). Cazurile de utilizare exprima cerintele functionale de nivel inalt ale sistemului.

❖ **Se dezvolta un plan initial al proiectului:** se impart cerintele in subseturi care vor fi implementate in diferite iteratii, se prioritizeaza cerintele.

❖ **Se defineste o arhitectura de nivel inalt a produsului software.**

Rational Unified Process (RUP)

❑ Rational Unified Process este un **proces de inginerie software**, dezvoltat de corporatia Rational Software (devenita divizie IBM din 2003) si integrat cu suita de instrumente de dezvoltare oferita de IBM.

❑ **Furnizeaza o abordare disciplinata** de asignare a sarcinilor si responsabilitatilor intr-o organizatie care dezvolta software.

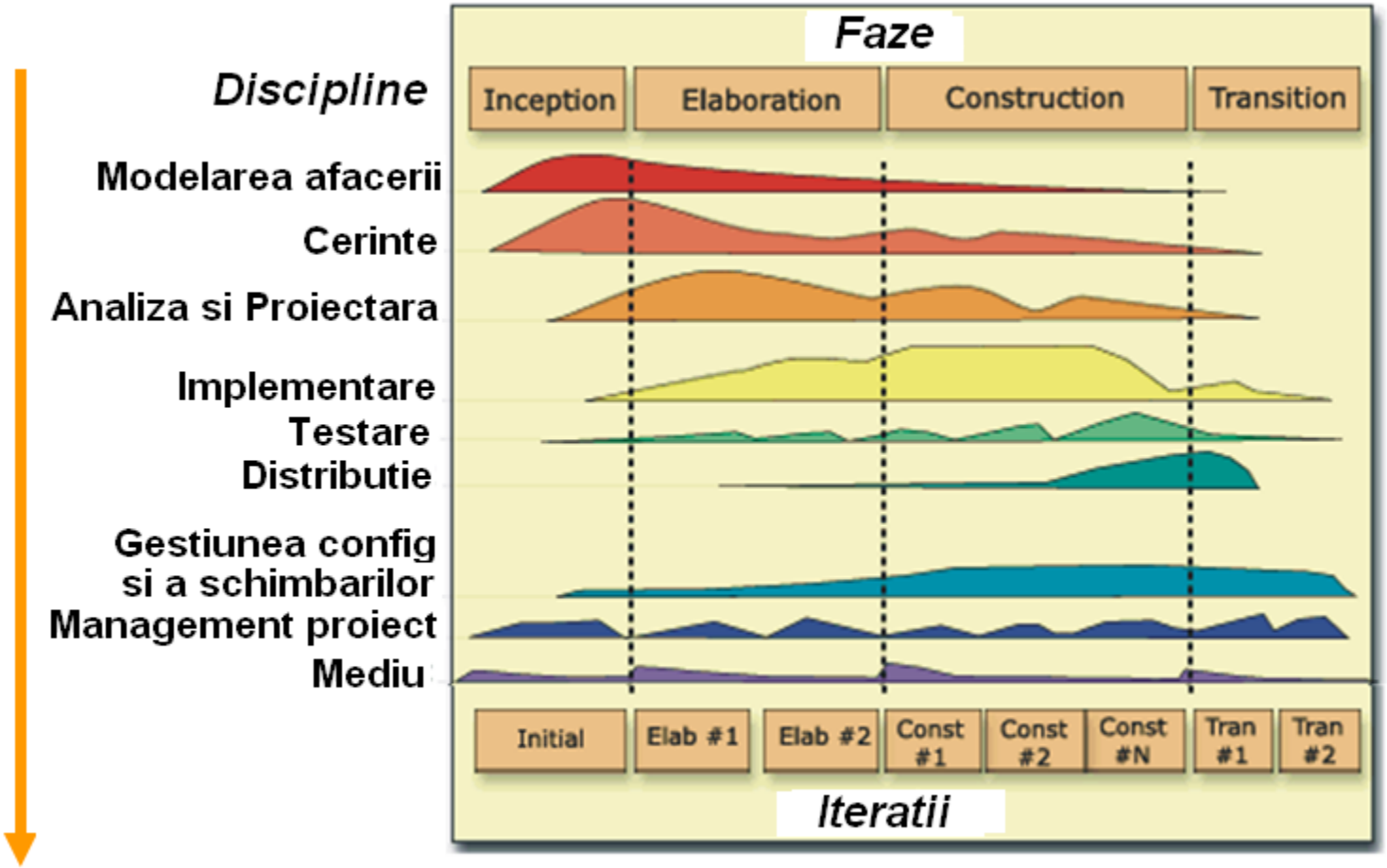
❑ RUP este o **platforma proces** care poate fi adaptata si extinsa pentru a satisface necesitatile organizatiei care o adopta.

❑ **Proces de dezvoltare iterativă, dirijat de urmatoarele principii**

- Diminuarea timpurie a riscurilor:
  - Primele iteratii trebuie sa adreseze riscurile cele mai mari
  - Riscul trebuie evaluat pe tot parcursul dezvoltarii: riscul se schimba in timp
  - Lista riscurilor actualizată este intrare pentru dezvoltarea planului iteratiei urmatoare
- Definirea timpurie a arhitecturii: in primele iteratii, testata prin prototipuri executabile
- Utilizarea de metrici obiective pentru evolutia procesului si a produsului dezvoltat

Dezvoltarea iterativa in RUP(1)

Cele doua dimensiuni ale procesului



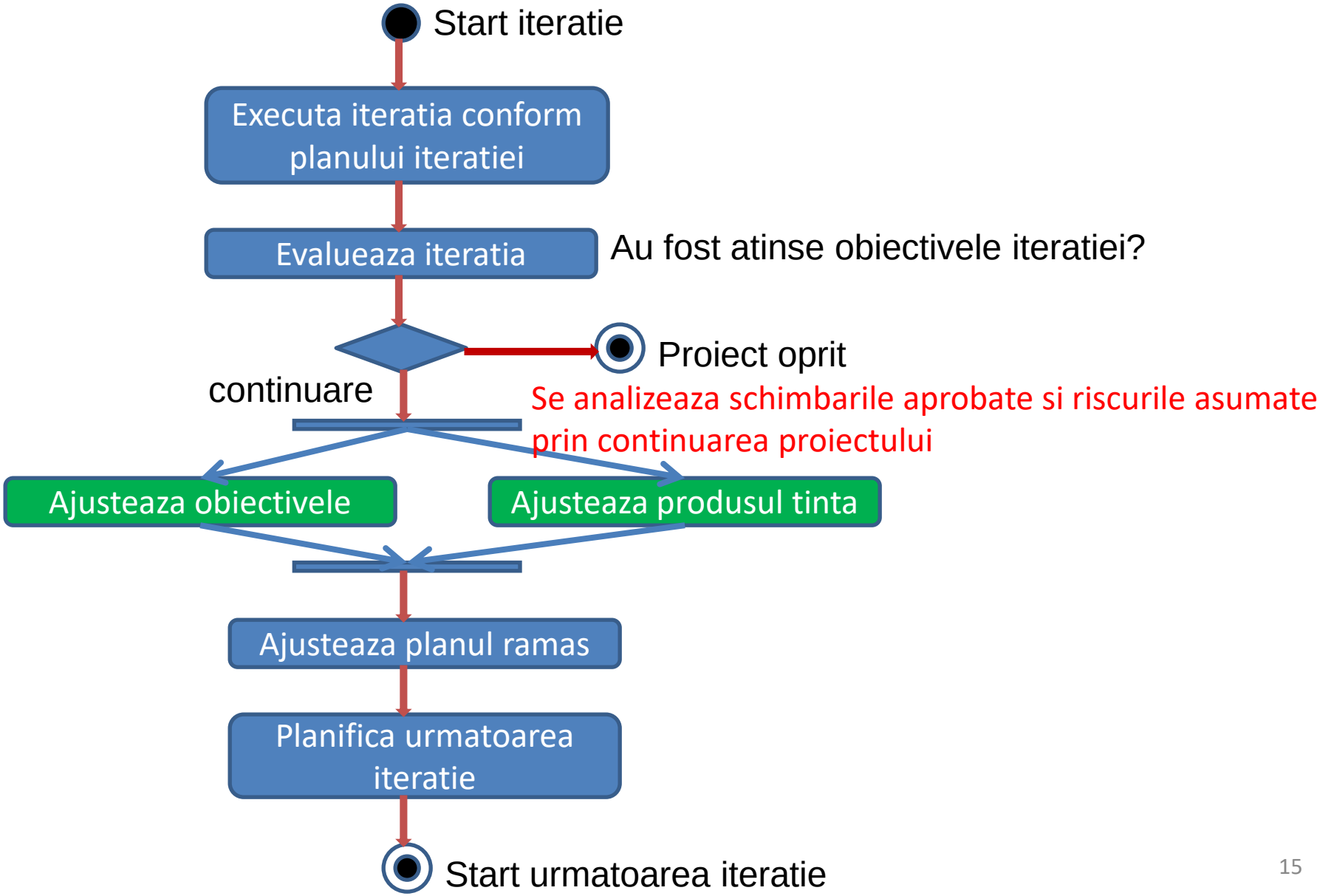
Continutul - aspectul static: activitati, artefacte, fluxuri de lucru

Timpul - aspectul dinamic: cicluri, faze, iteratii

Dezvoltarea iterativa in RUP(2)

- **Procesul de dezvoltare a unui produs software consta din mai multe cicluri de dezvoltare (iteratii)**, la sfarsitul fiecarui ciclu obtinandu-se un **produs executabil** care poate fi distribuit utilizatorilor. El satisface anumite cerinte specificate.
- **Un ciclu de dezvoltare este divizat in 4 faze consecutive**:
  - Faza de inceput (Inception) –se dezvolta o viziune asupra produsului final si o descriere a afacerii
  - Elaborarea (Elaboration) - se specifica cerintele produsului si se proiecteaza arhitectura
  - Construirea (Construction) - se construieste produsul adaugand toate componentele la arhitectura
  - Tranzitia (Transition) - produsul software este transferat comunitatii de utilizatori
- **Fiecare faza se incheie cu un jalon (milestone)** - se verifica satisfacerea anumitor criterii si se decide: intreruperea proiectului, refacerea fazei, trecerea la faza urmatoare
- **Fiecare faza se poate desfasura in mai multe iteratii**

Desfasurarea unei iteratii in RUP



Modelul Iterativ si Incremental - Concluzii

Avantaje:

• In fiecare iteratie se obtine un produs executabil, care satisface o parte din cerintele utilizator. Opus modelului cascada in care un produs executabil este disponibil la sfarsitul procesului de dezvoltare.

• Prototipurile pot fi livrate clientului/utilizatorilor:

→ feedback-ul partilor interesate in proiect (utilizatori, client, s.a) este distribuit pe intreg parcursul dezvoltarii.

• Flexibil la schimbarea cerintelor: cerintele noi sau modificate pot fi incoporate in urmatorul prototip.

• Depanarea si testarea mai usor de efectuat intr-o iteratie – se testeaza cerintele adaugate

• Produsul este mai fiabil decat intr-o dezvoltare in cascada: aspectele cele mai importante ale sistemului sunt cel mai mult testate.

• Riscurile de eşuare a proiectului sunt reduse.

Dezavantaje (riscuri) – in special intr-un proces neplanificat

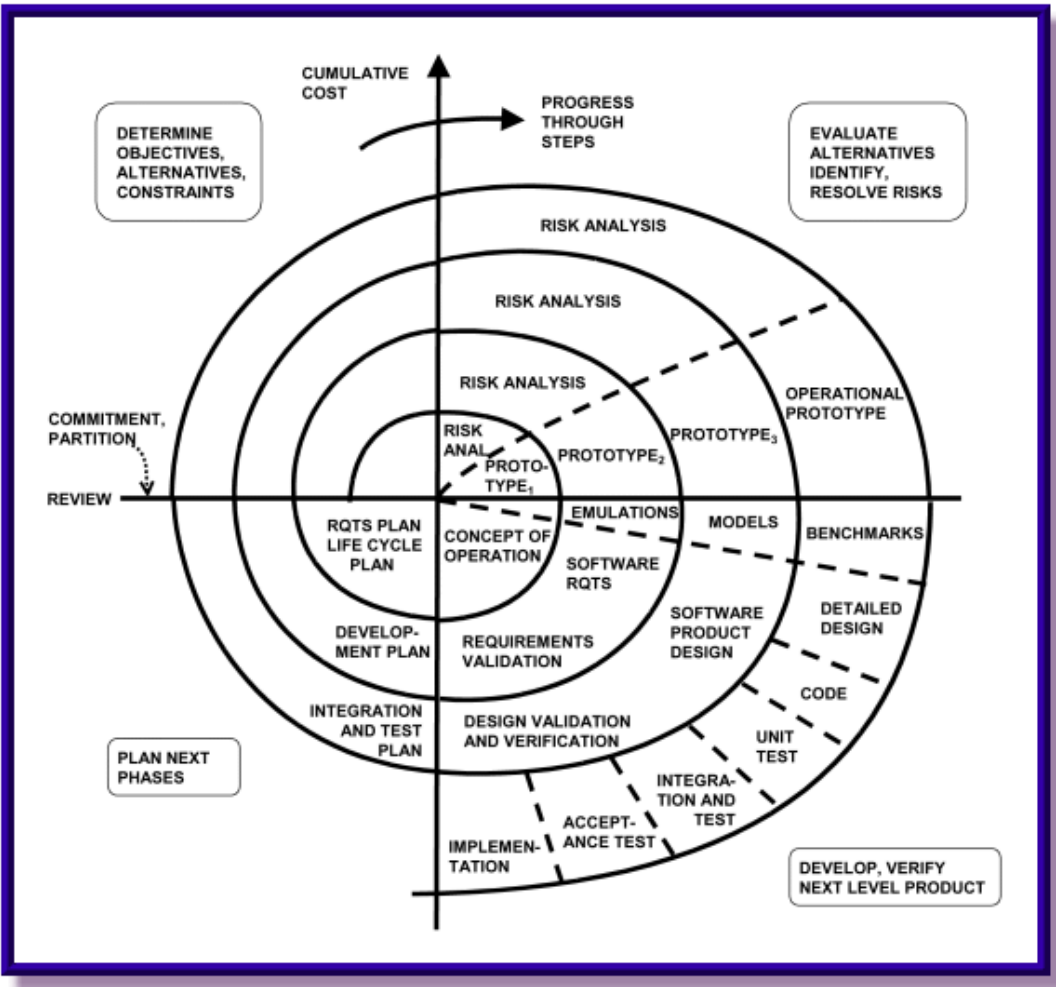
• Arhitectura initiala a produsului software poate fi degradata → testare dificila, intretinere dificila

• Abordarea incrementala se poate transforma usor intr-una 'construieste si repara' (build and fix)→ afecteaza calitatea produsului final: numarul de defecte ramase este mare, intretinerea dificila, s.a.

Modelul in Spirala

Boehm, 1986: http://en.wikipedia.org/wiki/Spiral_model

-Model focalizat pe analiza riscurilor.
-Combina ideea prototiparii cu dezvoltarea in cascada si dezvoltarea iterativa
-Adecvat proiectelor mari si critice.



- Fiecare iteratie (ciclu de dezvoltare) consta in repetarea a 4 faze si se termina cu un prototip al viitorului produs:

1. **Determinarea obiectivelor,** a constrangerilor, generarea alternativelor. Se colecteaza si studiaza cerintele, prin comunicare continua cu clientul.
2. **Evaluarea alternativelor**: analiza riscurilor, construirea de prototipuri.
3. **Dezvoltarea si verificarea urmatorului nivel al produsului:**

1) cerinte utilizator; 2) cerinte software;

3) proiectarea; 4) implementarea, integrarea, testarea

4. Evaluarea versiunii curente a produsului.

Planificarea urmatorului ciclu.

La sfarsitul unei iteratii, prototipul obtinut este evaluat de client. Se decide daca se va efectua o noua iteratie.

Lecturi recomandate

1. https://acodez.in/12-best-software-development-methodologies-pros-cons/
2. The seductive and dangerous V Model: http://www.clarotesting.com/page11.htm
3. Modern Methods of Software Development: https://task.gda.pl/files/quart/TQ2015/04/tq419v-c.pdf
4. Rational Unified Process: A Best Practices Approach: http://www.eecg.toronto.edu/~jacobsen/courses/ece1770/slides/rup.pdf
5. Kruchten, P. ,The Rational Unified Process: An Introduction**,** Third Edition, Addison-Wesley, Pearson Education, Inc., NJ, 2004
6. https://www.geeksforgeeks.org/software-engineering-spiral-model/

Extragerea cerintelor (Requirements elicitation)

Prof. univ. dr. ing. Florica Moldoveanu

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

Extragerea cerintelor

Scopul: identificarea si definirea cerințelor utilizator

- Se poate efectua în fazele de început ale procesului de dezvoltare sau pe intreg parcursul procesului de dezvoltare - în 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” (User Requirements) este sintetizat in “**Documentul de specificare a cerintelor**”.
- O **cerinţă** este:
  - o functionalitate pe care viitorul produs trebuie sa o ofere – cerinta functionala
  - o constrângere pe care produsul trebuie să o satisfacă pentru a fi acceptat de client – cerinta nefunctionala

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 cerinţelor

- Studiul de piaţă: exista produse similare? Ce va aduce nou viitorul produs?
- Domeniul in care va opera produsul: de ex. bancar, clinic, controlul in timp real al proceselor, etc; anumite cerinte sunt specifice domeniului.
- Participantii la procesul de extragere a cerintelor – fiecare participant are interese/cerinte specifice
- Fluxul de activități (procesul business) existent într-o organizatie, pe care noul sistem informatic trebuie sa-l îmbunatatesca – introducerea noului sistem nu trebuie sa produca schimbari neplanificate.
- Studiul de fezabilitate: este produsul realizabil cu satisfacerea cerintelor specificate, d.p.d.v. tehnic si financiar (costurile prevazute/negociate)?

Tehnici de extragere a cerintelor

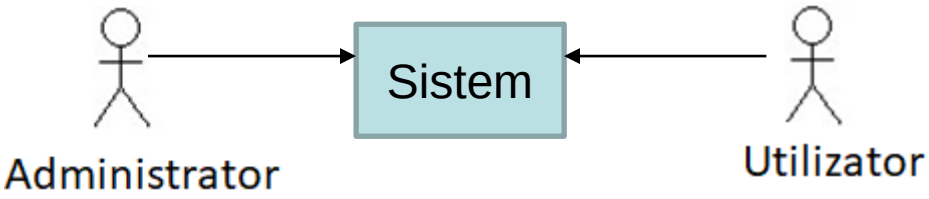
Difera, în functie de natura produsului software:

- Produs nou destinat pietei
  - Studiul produselor software similare existente – noul produs trebuie sa ofere ceva in plus: functionalitate, usurinta in utilizare, tehnologie noua, etc.
- Sistem informatic
  - Studiul fluxului de activitati al organizatiei in care va fi implementat sistemul informatic, entitatile, taskurile, etc, modul în care poate fi imbunatatit fluxul de activitati 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 de definire a cerintelor

Definirea cerinţelor utilizator include urmatoarele activitati efectuate de dezvoltator:

1. **Identificarea actorilor:** entitatile externe sistemului care vor interactiona cu sistemul



2. **Identificarea şi descrierea scenariilor de utilizare:** se definesc scenarii pentru functionalitatile tipice care vor fi furnizate de viitorul sistem.
   - Scenariile sunt exemple concrete de utilizare a viitorului sistem
   - Exprimate in limbaj natural: uşureaza comunicarea dezvoltatorului cu utilizatorii viitorului sistem şi înţelegerea domeniului aplicatiei

Actualizeaza baza de date
(scenariu de utilizare -1)

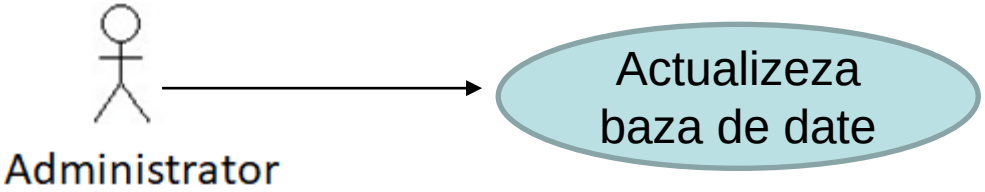
1. Login
2. Adauga date in tabelul “consum”
3. Logout

Activităţi de definire a cerintelor

Actualizeaza baza de date
(scenariu de utilizare -2)

1. Login
2. Modifica date in tabelul “consum”
3. Eroare la salvarea tabelului

3. 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.



Cazul de utilizare
“Actualizeaza baza de date”
unifica scenariile de utilizare in interactiunea dintre Administrator si sistem

4. Rafinarea cazurilor de utilizare: se detaliaza cazurile de utilizare descriind comportarea sistemului în prezenţa erorilor şi a condiţiilor exceptionale

5. Identificarea relatiilor dintre cazurile de utilizare

6. Identificarea şi definirea cerintelor nefunctionale: constrangeri privind performanta sistemului, consumul de resurse, cerinte de securitate, etc.

Identificarea actorilor (1)

- Un **actor** **este un rol** **pe care o entitate externa îl joaca in raport cu sistemul**:
  - Un utilizator direct al sistemului (un utilizator direct poate juca mai multe roluri)
  - Un echipament extern sau alt sistem care comunica cu sistemul analizat

1. Se determina tipurile de persoane care vor avea o legatura cu noul 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 determina **tipurile de utilizatori directi si operatiile pe care acestia le vor efectua cu sistemul,** sarcini, abilitati de lucru cu calculatorul, cunostintele lor tehnice.

2. Se determina entitatile externe sistemului (alte sisteme sau echipamente), care vor comunica cu sistemul.

Identificarea actorilor (2)

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 accesarea sa se propune o interfaţă Web.

Identificarea actorilor (3)

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 gestiona abonatii şi cartile din biblioteci: vor înregistra noi carti sau vor elimina pe unele existente, vor actualiza numarul de carti din fiecare exemplar, etc.

➢ **Identificarea actorilor**: **pe baza operatiilor efectuate cu sistemul de utilizatorii direcţi**. Se identifica 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 **Abonat** şi **Bibliotecar**, reprezentati în UML astfel:

Identificarea actorilor (4)

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?

Scenarii de utilizare (1)

- Un **scenariu** este o descriere narativa 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.

Exemple de scenarii de utilizare a SGCB pentru functionalitatea “Imprumut cărți”**:**

Scenariul Imprumut -1

1. Un utilizator acceseaza interfata web a sistemului in sectiunea pentru imprumut carti si completează rubricile rezervate numelui de utilizator şi parolei de acces, 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”.

Scenarii de utilizare (2)

Scenariul Imprumut -2

1. Un utilizator acceseaza interfata web a sistemului in sectiunea pentru imprumut carti si completează rubricile rezervate numelui de utilizator şi parolei de acces, apoi apasa butonul “Submit”.

2. Sistemul preia datele şi verifica identitatea utilizatorului.

3. Sistemul afişează mesajul: „Ati depasit numarul maxim de cărti ce pot fi împrumutate. Restituiti o parte dintre ele”.

Scenariul Imprumut -3

1. Un utilizator acceseaza interfata web a sistemului in sectiunea pentru imprumut carti si completează rubricile rezervate numelui de utilizator şi parolei de acces, 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 (3)

Intrebari care pot ajuta la identificarea scenariilor:

- Care sunt operatiile pe care un anumit actor doreste sa le efectueze cu sistemul?
- 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 care descriu aceeasi functionalitate 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) – descrie scenariul in care imprumutul poate fi efectuat

1. Un utilizator acceseaza interfata web a sistemului in sectiunea pentru imprumut carti si completează rubricile rezervate numelui de utilizator şi parolei de acces, 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 “Puteți lua cartea de la ghişeul bibliotecii cu adresa ….”.

Cazuri de utilizare (3)

Alternative:

La pasul 3:

3a) Utilizatorul nu este inregistrat ca abonat.

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.

Sesiunea este încheiată de sistem cu 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** la executia cazului de utilizare **(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 mai importante.

Cazuri de utilizare (6)

- De regula, preconditia precede descrierea cazului de utilizare iar postconditia o succede, dar este posibil ca ambele sa fie specificate fie inaintea descrierii cazului de utilizare, fie dupa.

De exemplu, pentru cazul de utilizare anterior:

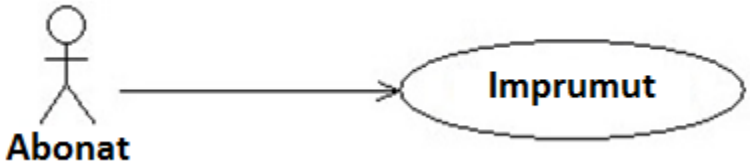
Preconditie:

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 grafic 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**:



Extragerea cerințelor continuare

Prof. univ. dr. ing. Florica Moldoveanu

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

Activităţi de extragere a cerintelor

- din cursul anterior-

1. **Identificarea actorilor:** entitatile externe sistemului care vor interactiona cu sistemul. Pentru sistemul considerat, SGCB (Sistemul de gestiune a cartilor din mai multe biblioteci), au fost identificati 2 actori: **bibliotecarul şi abonatul**.
2. **Identificarea şi descrierea scenariilor de utilizare:** se definesc scenarii pentru functionalitatile tipice care vor fi furnizate de viitorul sistem.
3. **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.
4. **Rafinarea cazurilor de utilizare**
5. **Identificarea şi reprezentarea relatiilor dintre cazurile de utilizare**
6. **Identificarea şi definirea cerintelor nefunctionale**

Cazuri de utilizare

- din cursul anterior-

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** la executia cazului de utilizare **(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 stari UML pentru descrierea comportamentului in timp al entitatilor din domeniul aplicatiei.

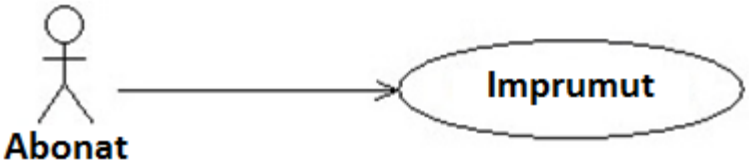
Descrierea unui caz de utilizare

- din cursul anterior-

Un caz de utilizare descrie o functionalitate a sistemului în raport cu un actor; este o abstractizare a tuturor scenariilor care descriu acea functionalitate.

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 SGCB declanşat de Abonat



Fluxul de baza (secvenţa tipica de paşi)

1. Un utilizator acceseaza interfata web a sistemului in sectiunea pentru imprumut carti si completează rubricile rezervate numelui de utilizator şi parolei de acces, 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”.

Descrierea unui caz de utilizare

- din cursul anterior-

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”.

La fluxul de bază se pot adauga o *preconditie* şi o *postconditie*.

Rafinarea cazurilor de utilizare

In acest pas se adauga detalii la cazurile de utilizare, care descriu comportarea sistemului în prezenţa defectelor şi a condiţiilor exceptionale.

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

Exceptii

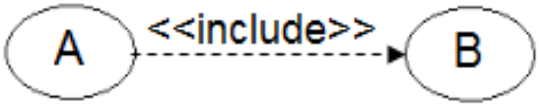
La pasul 2: Sesiunea este încheiata de sistem cu mesajul “Eroare de acces la lista de utilizatori”.

La pasul 6: Sesiunea este incheiata de sistem cu mesajul “Eroare de acces la lista de abonati”.

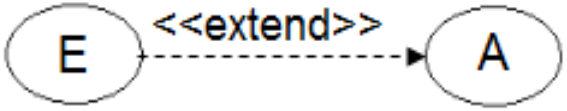
Relaţii între cazurile de utilizare - (1)
- reprezentare in UML -

Intre cazurile de utilizare se pot stabili relatii de:

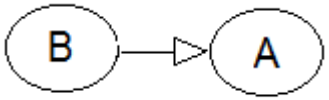
- **Includere** – un caz de utilizare A include, în unul sau mai multi pasi, comportamentul descris într-un alt caz de utilizare, B.



- **Extindere** – un caz de utilizare E extinde un alt caz de utilizare, A, daca A include comportamentul descris in E, numai in anumite conditii.



- **Generalizare/specializare**: similara conceptual cu relatia de generalizare dintre clase

B este o specializare a cazului de utilizare A: 

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)

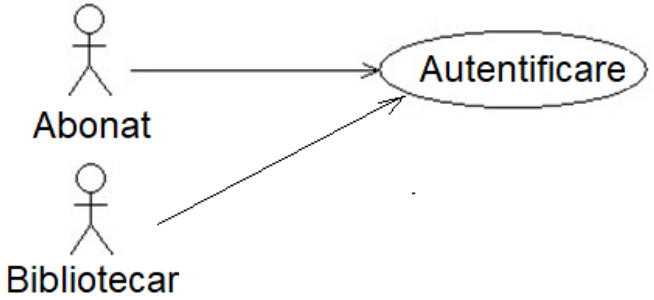
Relatia de includere - exemplificare

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

- operatia de autentificare trebuie sa fie inclusa in orice caz de utilizare al sistemului.

Autentificarea consta dintr-o secventa de pasi, care:

- poate fi inclusa in fiecare caz de utilizare, sau
- descrisa intr-un caz de utilizare separat care va fi inclus in toate celelalte cazuri de utilizare



Cazul de utilizare "Autentificare"

Fluxul de baza

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.

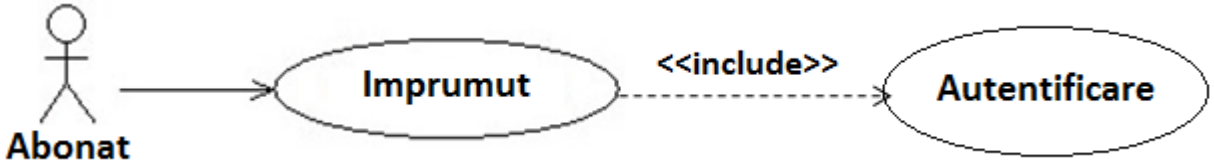
Postconditie: utilizatorul este autentificat in sistem

Relatii între cazurile de utilizare (3)

Alternative:

La pasul 3: Sistemul afiseaza mesajul: „Nume de utilizator inexistent sau combinatia nume_utilizator-parola este incorecta” şi încheie sesiunea.

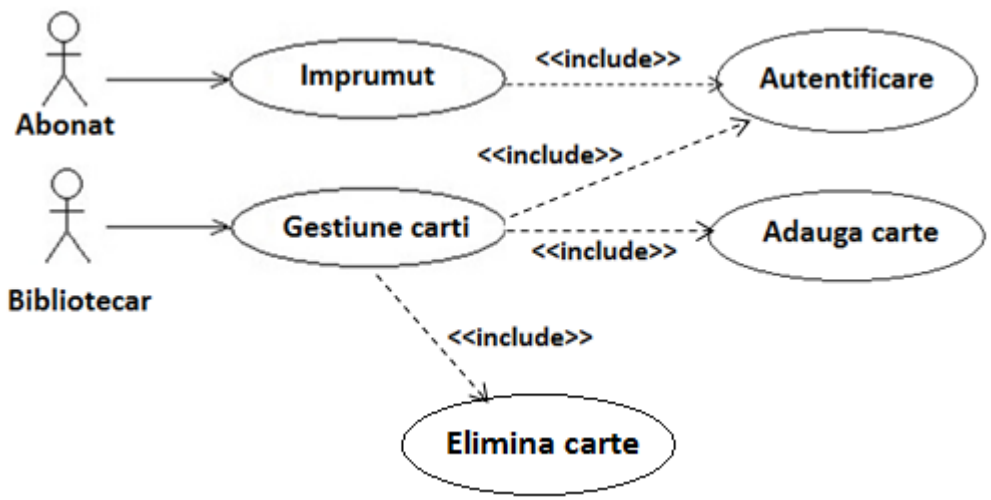
Cazul de utilizare “autentificare” va fi inclus in cazul de utilizare “Imprumut” si în toate celelalte cazuri de utilizare:

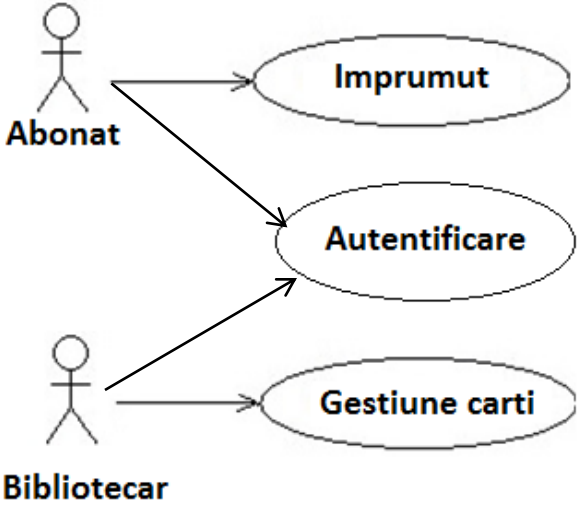


Cazul de utilizare “Imprumut”

1. Utilizatorul executa cazul de utilizare “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…….

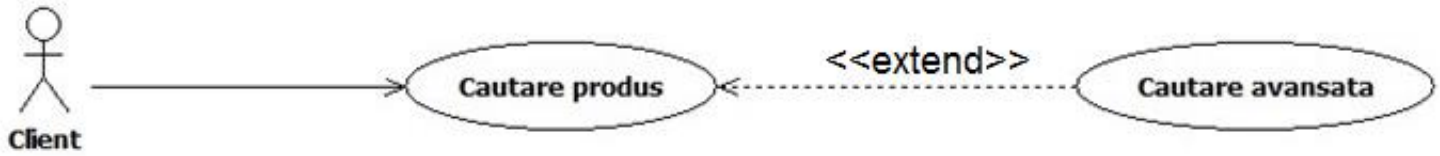
Aceasta presupune că înainte de orice utilizare a sistemului, utilizatorul trebuie sa execute cazul de utilizare ***Autentificare*** terminat prin autentificarea sa în sistem.

Relatii între cazurile de utilizare (5)

Relatia de extindere - exemplificare

Utilizatorul unui site 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. Site-ul întoarce mesajul "nici un rezultat" şi utilizatorului îi este oferită opţiunea de căutare avansată.
3. **Daca** utilizatorul selectează opţiunea de căutare avansată **atunci** se continua cu pasul 4 **altfel** se continua cu pasul 6
4. Se executa cazul de utilizare "Cautare avansata"
5. Site-ul 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 sau apare in mai multi pasi ai unui caz 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 multi actori/cazuri de utilizare, având caracteristici comune, sunt specializări ale unui actor/caz de utilizare mai general.

Relatii între cazurile de utilizare (7)

Relatia de generalizare

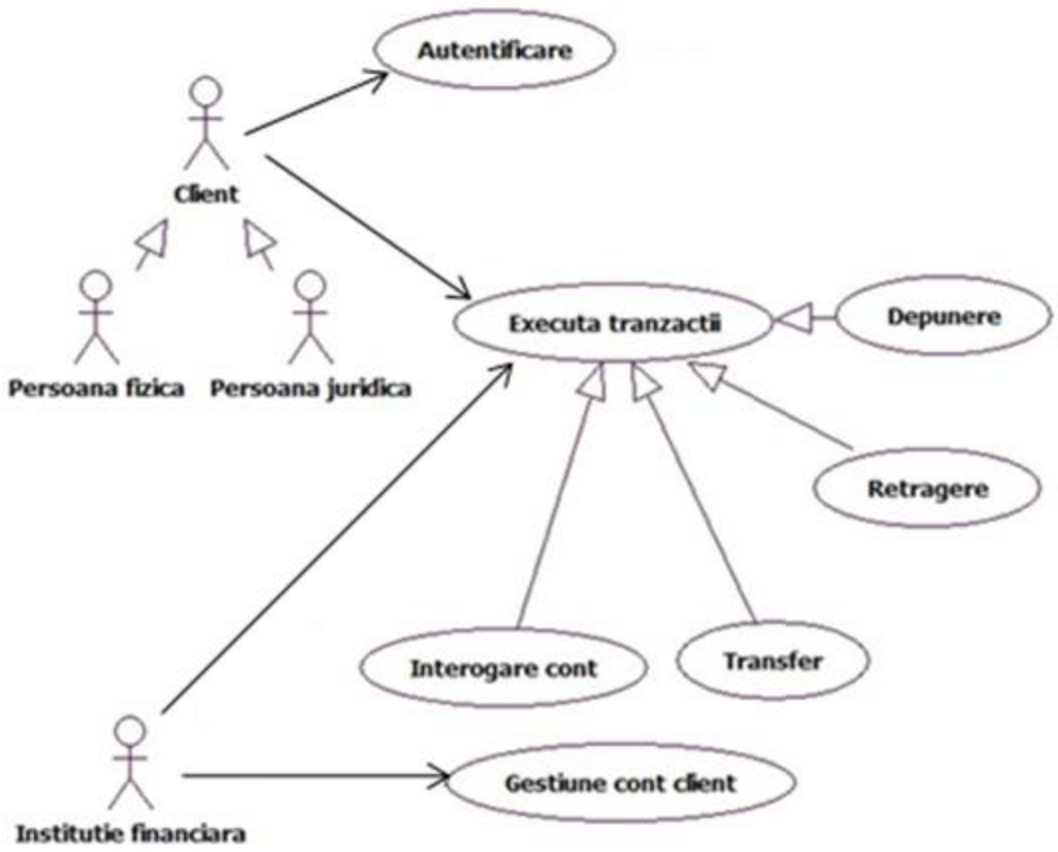
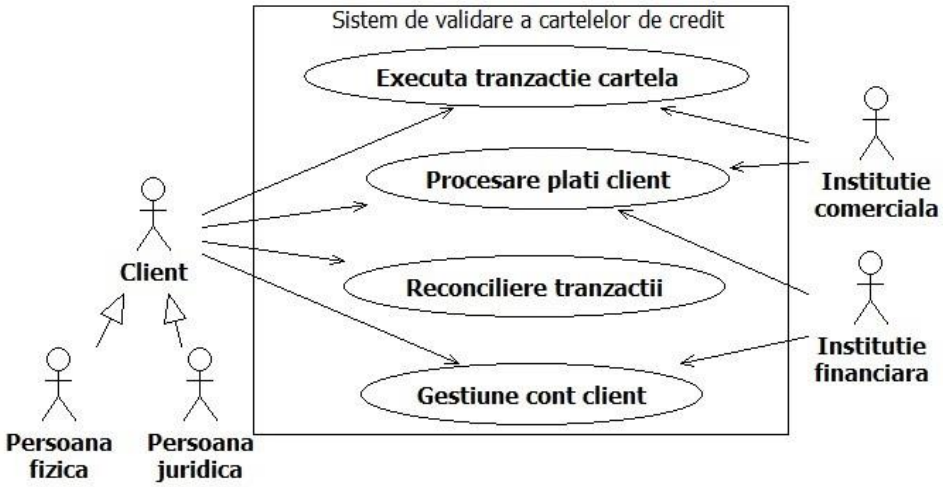


Diagrama de context

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



Modelul cazurilor de utilizare (Use case model)

Cuprinde:

- Descrierea actorilor
- Descrieri ale principalelor scenarii de utilizare
- Descrierile tuturor cazurilor de utilizare
- Diagrame de cazuri de utilizare
- Diagrama de context

Se foloseşte pentru:

- Specificarea cerințelor funcționale
- Validarea arhitecturii sistemului
- Generarea cazurilor de test

Cerinte nefunctionale (1)

Identificarea cerintelor nefunctionale

Cerintele nefunctionale sunt cerinte care nu sunt asociate cu un caz de utilizare specific.

Pot fi:

- Constrangeri:
  - De comunicare hardware-software, compatibilitate cu alte sisteme software, interfata utilizator, constrangeri de operare, legislative.
- Cerinte de calitate a produsului: fiabilitate, portabilitate, adaptabilitate, performanţă, disponibilitate, securitate, siguranta in functionare, standarde.
- Cerinte de planificare a proiectului: termene, produse livrate, resurse necesare, cerinte de verificare a produsului final, cerinte de asigurare a calitatii produsului.

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 sau valori individuale:

  valoare acceptabila: [min-max]; valoare ideala.

  Ex. timpul de raspuns: < 1 sec

Cerinte de interfata

- Interfetele noului produs cu diferite omponente hardware/software
- Interfete interne / externe ale produsului software
- Se definesc prin:
  - Protocoalele de comunicatie de utilizat
  - Fluxul de date prin interfata
  - Evenimentele care declanseaza schimburi de date prin interfata, etc

Cerinte nefunctionale (3)

Cerinte de operare

- Modul de comunicare a sistemului cu operatorii umani, aspecte fizice si ergonomice ale interfetei utilizator:
  - Dialogul operator-sistem
  - Aspectul ecranului in timpul dialogului
  - Stilul limbajului de comenzi
  - Usurinta de învatare a modului de operare cu sistemul (perioada invatarii): stilul interfetei, scheme de culoare, help online, documentatia de utilizare.

Cerinte impuse resurselor fizice

  - Puterea de prelucrare
  - Memoria necesara
  - Spatiul pe disc

Cerinte nefunctionale (4)

Cerinte de intretinere

- Usurinta de reparare a defectelor, de îmbunatatire sau adaptare la schimbarea cerintelor. Exemplu: "Timpul de reparare a unui defect nu va depasi niciodata o saptamana“.

Cerinte de fiabilitate

- Frecventa acceptata a caderilor software (cadere: o comportare a sistemului neconforma cu specificatia); poate fi specificata pe categorii/tipuri de caderi:
- Exprimate folosind metrici ca 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 de portabilitate

  – "Sa poata rula pe calculatorul X/sub sistemul de operare Y 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 securitate

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

Cerinte de siguranta in functionare (Safety)

- Protectia împotriva distrugerilor cauzate de caderile software. De exemplu, ce trebuie sa se întample în cazul producerii unei caderi software:
  - Degradare treptata
  - Continuare dintr-un anumit punct
  - Sa fie implementate tehnici de izolare a caderilor

Cerinte nefunctionale (6)

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 asigurare a calitatii

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

Cerinte legislative

- Conditii de certificare, de licentiere, etc

Documentarea cerintelor utilizator(1)

Cerintele extrase sunt de regula definite într-un **document general de Specificare a cerintelor software**, care include atât cerințele utilizator cât şi rezultatul analizei cerintelor.

In cadrul documentului sunt definite:

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

Documentarea cerintelor utilizator(2)

4. **Actorii**: rolurile entitatilor externe care interactioneaza direct cu sistemul (om sau alta entitate 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*

- principalele scenarii de utilizare a sistemului de catre diferiti actori.

- schițe ale interfeței utilizator pentru diferite scenarii de utilizare

- cazurile de utilizare ale sistemului şi diagrame de cazuri de utilizare

6. Cerintele nefunctionale

Documentul cerintelor software (Software Requirements Document)

Continutul general al documentului (finalizat dupa analiza cerintelor)

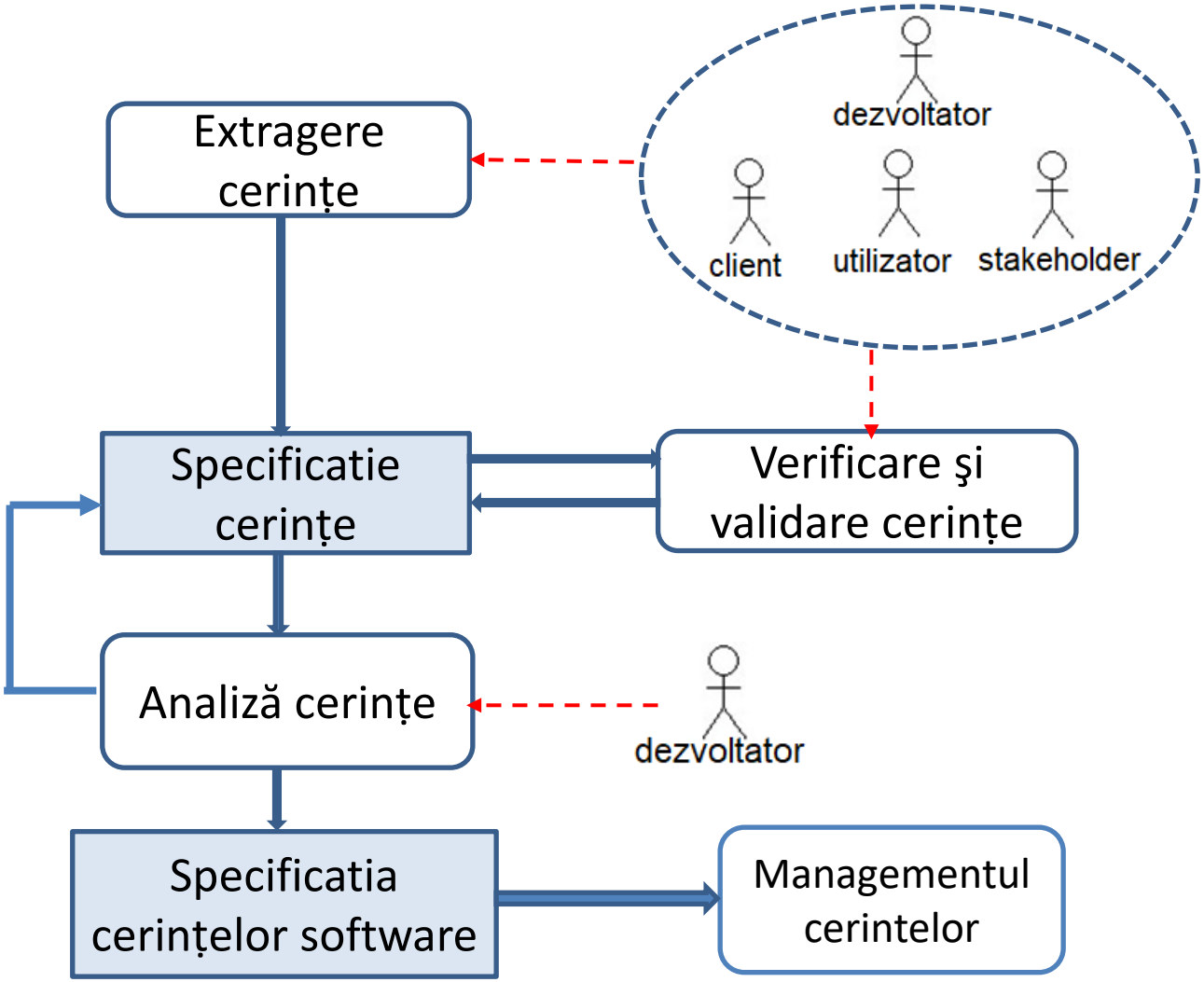
1. Introducere
    - 1.1. Scopul documentului
    - 1.2. Domeniul aplicatiei (in care se incadreaza/va functiona noul produs)
    - 1.3. Scopul noului produs
    - 1.4. Obiectivele şi criteriile de succes ale proiectului. Produse similare.
    - 1.5. Referinţe (la sisteme existente, studii de fezabilitate, etc.) – daca este cazul
    - 1.6. Lista de definitii si abrevieri
2. Sistemul curent (daca sistemul propus va inlocui un sistem)
3. Sistemul propus (noul produs)
    - 3.1. Descriere generala
    - 3.2. Descrierea categoriilor de utilizatori directi/indirecti ai sistemului
    - 3.3. Mediul de operare – daca este cazul
    - 3.4. Cerinte de sistem (echipamente de calcul, dispozitive, comunicatia in retea)
    - 3.5. Cerinte functionale (in limbaj natural)
    - 3.6. Cerinte nefunctionale
    - 3.7. Modele ale sistemului
        - 3.7.1. Actorii si cazurile de utilizare prin care interactioneaza
        - 3.7.2. Descrierea principalelor scenarii de utilizare
        - 3.7.3. Descrierea cazurilor de utilizare ale sistemului (interfata utilizator la executia CU)
        - 3.7.4. Diagrama de context
        - 3.7.5. Modelul dinamic – definit in etapa de analiza a cerintelor
        - 3.7.6. Modelul obiect – definit in etapa de analiza a cerintelor

Procesul cerințelor în dezvoltarea software

Prof. univ. dr. ing. Florica Moldoveanu

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

Procesul cerintelor



Verificarea si validarea cerintelor

- Verificarile sunt efectuate pe tot parcursul perioadei de specificare a cerintelor (poate fi intreaga durata de dezvoltare)
- Prin **validare** se urmareste sa se stabileasca împreuna cu toate partile interesate in proiect că:
  - cerintele sunt **complete, consistente, neambigue si corecte;**
  - au fost **bine întelese şi în acelaşi fel** de catre toate partile interesate în proiect;
  - satisfac standardele impuse.
- Modul uzual de verificare si validare consta in **revizia documentului de specificare** de catre un grup din care fac parte reprezentanti ai dezvoltatorului, clientul şi reprezentanti ai utilizatorilor.

Analiza cerintelor (1)

- Este responsabilitatea dezvoltatorului.
- **Participanti**: ingineri de sistem, ingineri software, analisti, manageri, persoane responsabile de asigurarea calitatii.
- **Scop:**
  - **Descoperirea erorilor** în cerintele formulate: cerintele sunt actualizate apoi revazute împreuna cu clientul si celelalte parti participante la extragerea cerintelor.
  - **Rezolvarea conflictelor dintre cerintele** diferitelor tipuri de utilizatori sau dintre cerinte functionale si ne-functionale.
  - **Clasificarea cerintelor dupa diferite criterii**:

    - Operationale, de calitate, cerinte de produs/proces, etc.
    - Importanța, prioritatea intr-o dezvoltare iterativa
    - Stabilitatea (modificarea lor pe parcursul dezvoltarii sau chiar si dupa livrare)
  - **Structurarea si formalizarea cerintelor extrase** – permite descoperirea erorilor in cerintele formulate, identificarea si rezolvarea problemelor dificile la începutul procesului de dezvoltare.

Analiza cerintelor (2)

- **In etapa de analiza cerintele sunt structurate si formalizate in modelul de analiza.**
- **Modelul de analiza - model conceptual al sistemului. Cuprinde:**
  - **Modelul obiect**, alcatuit din **clase, diagrame de clase si diagrame de obiecte**.

- **Obiectele** din modelul de analiza sunt entitati din domeniul aplicatiei (obiecte conceptuale).

Exemple: cărți, utilizatori, abonati in aplicatia SGCB;

studenti, profesori, cataloage, cursuri – intr-o aplicatie de planificare a studentilor la cursuri

- **Clasele** din modelul de analiza sunt clase conceptuale.

  - In etapa de proiectare se decide cum vor fi implementate clasele: clase intr-un lb. POO sau tabele ale unei baze de date relationale.

- **O diagrama de clase conceptuala** poate fi schema conceptuala a unei baze de date

Analiza cerintelor (3)

- **Modelul dinamic**, alcatuit din **diagrame de interactiune si diagrame de stari**

  - **O diagrama de interactiune** (de secventa sau de colaborare) reprezinta **interactiunile** dintre un actor şi sistem sau interactiuni între obiecte conceptuale, **în timpul unui singur scenariu**.

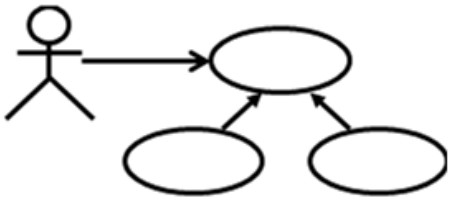
  - Interactiunile sunt reprezentate prin mesaje schimbate intre participanti

  - **O diagrama de stari** reprezinta **comportamentul in timp** al obiectelor dintr-o clasa de obiecte sau comportamentul in timp al intregului sistem.

  - Comportamentul in timp este redat prin stări, tranzitii intre stari si evenimente care determina tranzitiile.

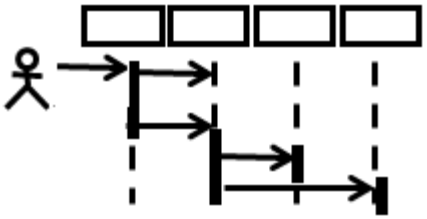
Analiza cerintelor (4)

Specificaţia funcţionala

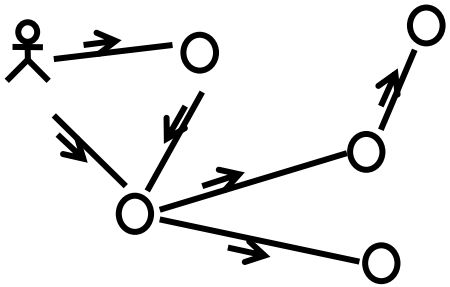


Modelul cazurilor de utilizare

Obiecte conceptuale

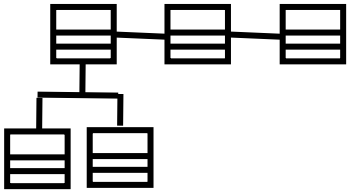


Diagrame de secventa care modeleaza scenarii



Diagrame de colaborare (de comunicare, in UML 2.x)

→ In analiza orientata obiect, se trece de la descrierea functionala a cerintelor structurata în raport cu actorii la o structurare obiect, pornind de la scenarii si cazuri de utilizare.



Diagrame de clase conceptuale

Managementul cerintelor

Scop: Managementul schimbarilor cerintelor pe parcursul dezvoltarii

- Pot aparea cerinte noi
- Se pot schimba cele existente →cerintele trebuie sa fie trasabile
- Prioritatile cerintelor se pot schimba pe parcursul dezvoltarii
- Mediul de operare si cel tehnic se pot modifica pe parcusul dezvoltarii

Managementul cerintelor:

- Include activitati de verificare si validare
- Necesita instrumente software care sa asiste procesul cerintelor

Cat timp sa se consume cu definirea cerintelor?

- Timpul consumat **depinde de modelul de dezvoltare** (scurt in modelele agile).
- Aproximativ 20-25% din timpul total al proiectului trebuie alocat extragerii, analizei si specificarii cerintelor.
- 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 înteles de multe persoane.
- Documentul cerintelor trebuie actualizat la orice modificare sau adaugare de cerinte:

  → managementul schimbarilor, asistat de instrumente speciale.

Calitatile si avantajele unei bune specificatii a cerintelor software(1)

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

Calitatile - specificatiile cerintelor trebuie sa fie:

- **Corecte -** sa reprezinte cu acuratete sistemul de care clientul are nevoie si pe care dezvoltatorul intentioneaza sa-l construiasca.
- **Neambigue** – fiecare cerinta definita trebuie sa aibă o singura interpretare
- **Complete** – toate aspectele sistemului, functionale si nefunctionale, inclusiv situatiile exceptionale, trebuie sa fie descrise in documentul cerintelor.
- **Consistente** – sa nu existe contradictii între diferite cerinte specificate.
- **Clasificate** dupa importanţă şi stabilitate.

Calitatile si avantajele unei bune specificatii a cerintelor software(2)

- **Verificabile**. Trebuie evitate cerinte ca : ”sistemul va furniza un raspuns rapid”, “sistemul nu va cadea niciodata”, etc.
  - O cerinta este verificabila daca se poate demonstra prin teste ale sistemului ca este implementata.
- **Realizabile** – sa poata fi implementate.
- **Trasabile**: usor de urmarit in diferite artefacte ale procesului de dezvoltare.
  - Este o calitate critica pentru testarea sistemului si pentru evaluarea efectului schimbarilor in cerinte.

Calitatile si avantajele unei bune specificatii a cerintelor software(3)

Avantajele

- Documentul cerintelor software contine o descriere completa a produsului

  → sta la baza contractului dintre client si dezvoltator.

- Reduce efortul de dezvoltare: revizia documentului cerintelor poate evidentia omisiuni, inconsistente şi neînţelegeri care pot fi corectate mai usor în aceasta etapa decat mai tarziu.
- Sta la baza estimarii costurilor si a planificarii procesului de dezvoltare.
- Permite planificarea testelor de sistem.
- Uşureaza transferul produsului la noi utilizatori sau pe platforme noi.
- Serveste ca baza pentru viitoarele îmbunatatiri sau modificari ale produsului.

Modelarea sistemelor software

Prof. univ. dr. ing. Florica Moldoveanu

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

Modele ale sistemelor software (1)

Modelarea: parte esenţială în orice proiect software, în special în proiectele mari.

Sistem software - sistem alcatuit din componente interconectate: programe (cod sursa, cod obiect), biblioteci, fisiere, baze de date, cazuri de test, documentatia de instalare, documentatia de utilizare, si altele.

Modelele: reprezentari abstracte ale sistemului

- **create** în etapele care preced codificarea:
  - Extragerea şi analiza cerinţelor
  - Proiectarea arhitecturala
  - Proiectarea de detaliu
- **utilizate**:
  - **înainte de codificare,** pentru a verifica:
    - daca funcţiile prevazute sunt complete si corect modelate,
    - dacă toate cerinţele utilizatorilor sunt acoperite de arhitectura proiectată,
    - daca arhitectura este robusta si extensibila.
  - **după codificare**, pentru verificarea si validarea sistemului.

Modele ale sistemelor software (2)

Modelele de specificare a cerinţelor:

- Exprima cerinţele impuse sistemului
- Corespund unei vederi externe asupra sistemului
- Se folosesc de către client, viitorii utilizatori ai sistemului, experţi ai domeniului în care va opera sistemul, analişti, echipa de verificare şi validare a sistemului.
- Se folosesc pentru validarea arhitecturii sistemului
- Se folosesc pentru generarea cazurilor de test

Modelele de proiectare:

- Redau arhitectura sistemului, subsistemele şi interfeţele lor, distributia proceselor în sistem şi sincronizarea lor.
- Redau realizarea fizică a sistemului, echipamentele din componenţa sa şi repartiţia artefactelor software pe echipamente.

UML - Unified Modelling Language (1)

- Este un limbaj folosit pentru modelarea sistemelor software (http://www.uml.org).
- Este independent de metoda/metodologia de dezvoltare folosita.
- Aparut din necesitatea unei standardizari a elementelor de modelare folosite in metodele de dezvoltare orientata obiect din anii ’90 (OMT – Object Modeling Technique, OOA-Object Oriented Analysis, OOD-Object Oriented Design, s.a).
- Prima versiune UML a fost publicata in 1996 (rezultatul colaborarii intre cei trei lideri recunoscuti in domeniul metodologiilor orientate obiect: Booch, Rumbaugh, şi Jacobson).
- Au urmat numeroase versiuni, între care, cele mai importante în privinţa modificărilor aduse:

UML 1.0: ianuarie 1997 ; UML 1.1: noiembrie 1997 – propunerea adoptată de OMG (Object Management Group; http://www.omg.org/)

UML 1.3: 2000; UML 1.4: 2001; UML 1.5: 2003

UML 2.0: august 2005

UML 2.3: mai 2010

UML 2.4.1: august 2011– versiunea utilizata in prezent

UML 2.5.1: decembrie 2017

UML - Unified Modelling Language(2)

UML este un limbaj pentru:

- Vizualizare şi comunicare
- Specificarea şi construirea sistemului
- Documentare

Permite modelarea structurală şi comportamentală a unui sistem software:

Modele comportamentale (aspecte dinamice ale sistemului)	**Modele structurale (structura statica a sistemului)**
Cazuri de utilizare	Diagrame de clase
Diagrame de cazuri de utilizare	Diagrame de obiecte
Diagrame de interacţiune	Diagrame de componente
Diagrame de stări	Diagrame de distribuţie
Diagrame de activitate	Diagrame de pachete

UML
Diagrame de interactiune

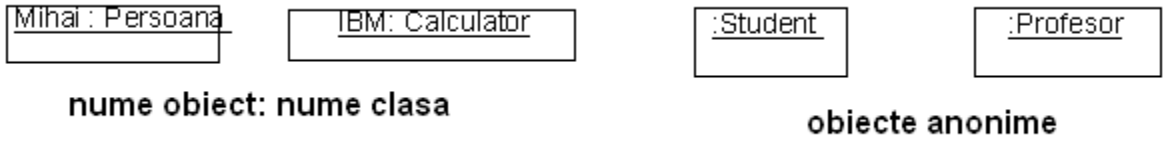
Prof. univ. dr. ing. Florica Moldoveanu

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

Diagrame de interactiune

- Se pot utiliza la doua niveluri:
  - In **analiza cerintelor**, pentru reprezentarea scenariilor de utilizare.
  - In **proiectarea de detaliu**, pentru a ilustra comunicarea dintre obiecte la momentul execuției aplicației.
- Conțin obiecte şi mesaje schimbate între obiecte.
- **Sunt doua tipuri de diagrame de interactiune**
  - Diagrame de secventa - evidentiaza ordonarea în timp a mesajelor
  - Diagrame de colaborare – evidentiaza relatiile structurale dintre obiecte

Reprezentarea obiectelor in UML



Diagrame de secventa

- **O diagrama de secventa folosita in analiza cerintelor:**
  - Contine:
    - Obiectele (entitati din domeniul aplicatiei) si actorii care participa într-un scenariu (sau în fluxul de baza al unui caz de utilizare)
    - Secventa de mesaje schimbate între obiecte sau între actori şi obiecte/sistem
  - Ilustrează ordonarea in timp a mesajelor

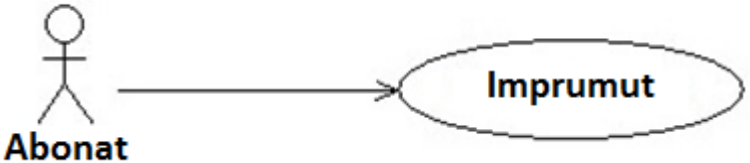
- **Comportamentul unui obiect**, ca urmare a unor stimulari externe, este reprezentat prin **operatii.**

- **Operatiile unui obiect sunt declanşate prin mesaje** trimise de alte obiecte, de obiectul însuşi sau de un actor.

Obiect1 $\xrightarrow{\textbf{mesaj: execută operația X}}$ Obiect2

De la scenarii la diagrame de secventa

Exemplu: Cazul de utilizare ”Imprumut” al SGCB

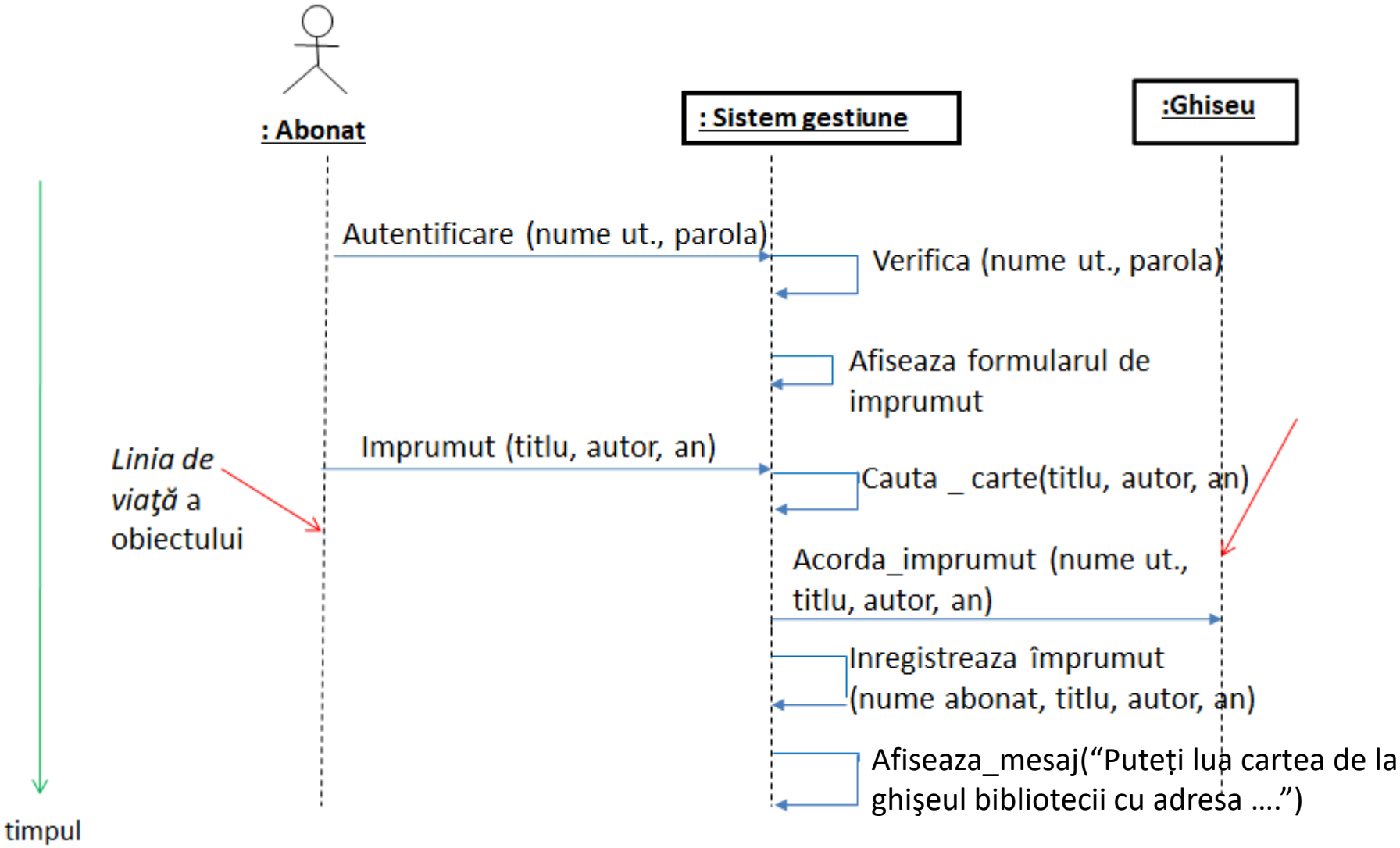


Fluxul de baza (secvenţa tipica de paşi)

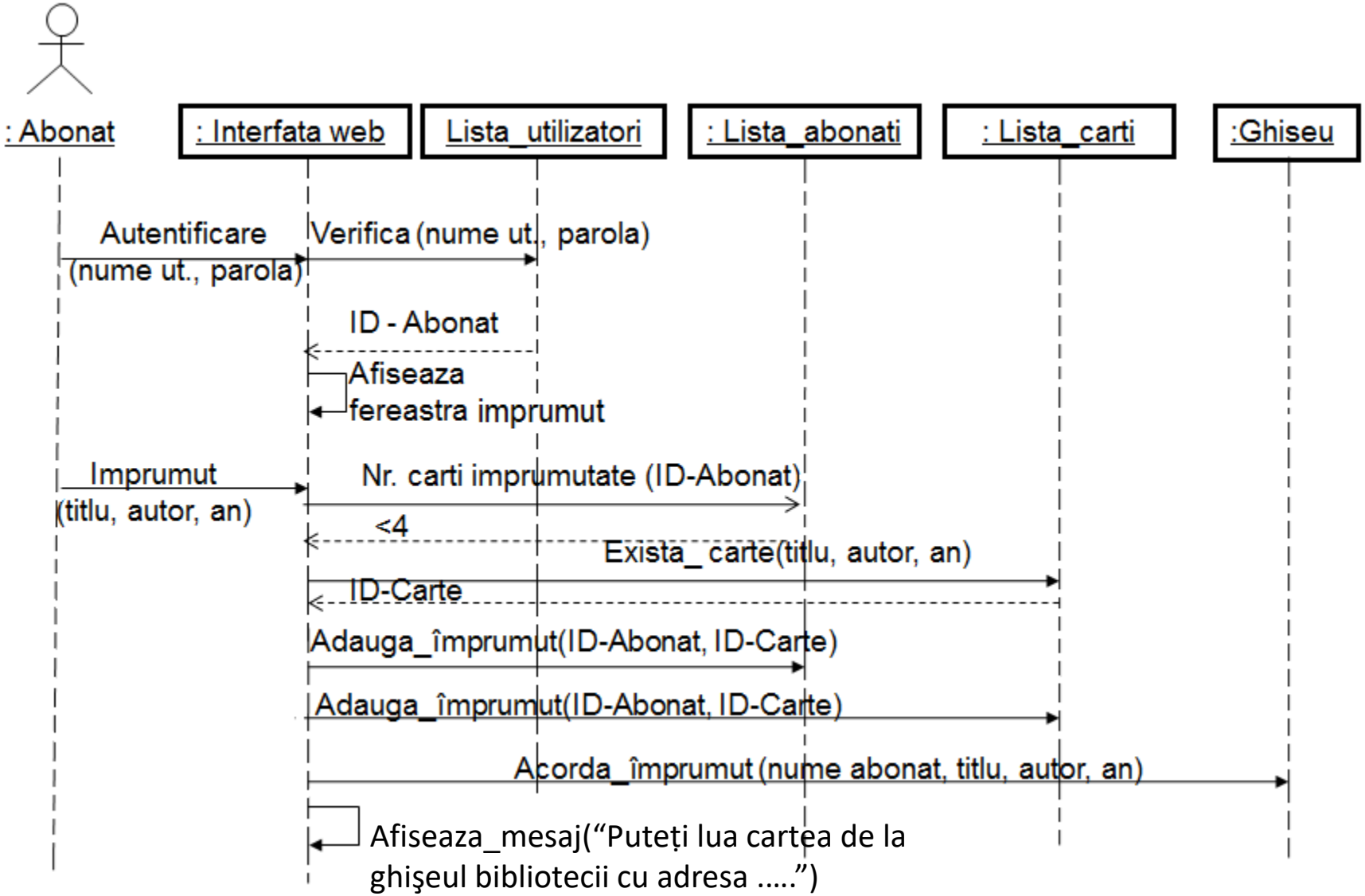
1. Un utilizator acceseaza interfata web a sistemului in sectiunea pentru imprumut carti si completează rubricile rezervate numelui de utilizator şi parolei de acces, apoi apasa butonul “Submit”.
2. Sistemul preia datele şi verifică identitatea utilizatorului.
3. Sistemul afişează formularul de împrumut.
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 “Puteți lua cartea de la ghişeul bibliotecii cu adresa ….”.

In fiecare pas se execută o operatie declanşata de actor sau de sistem.

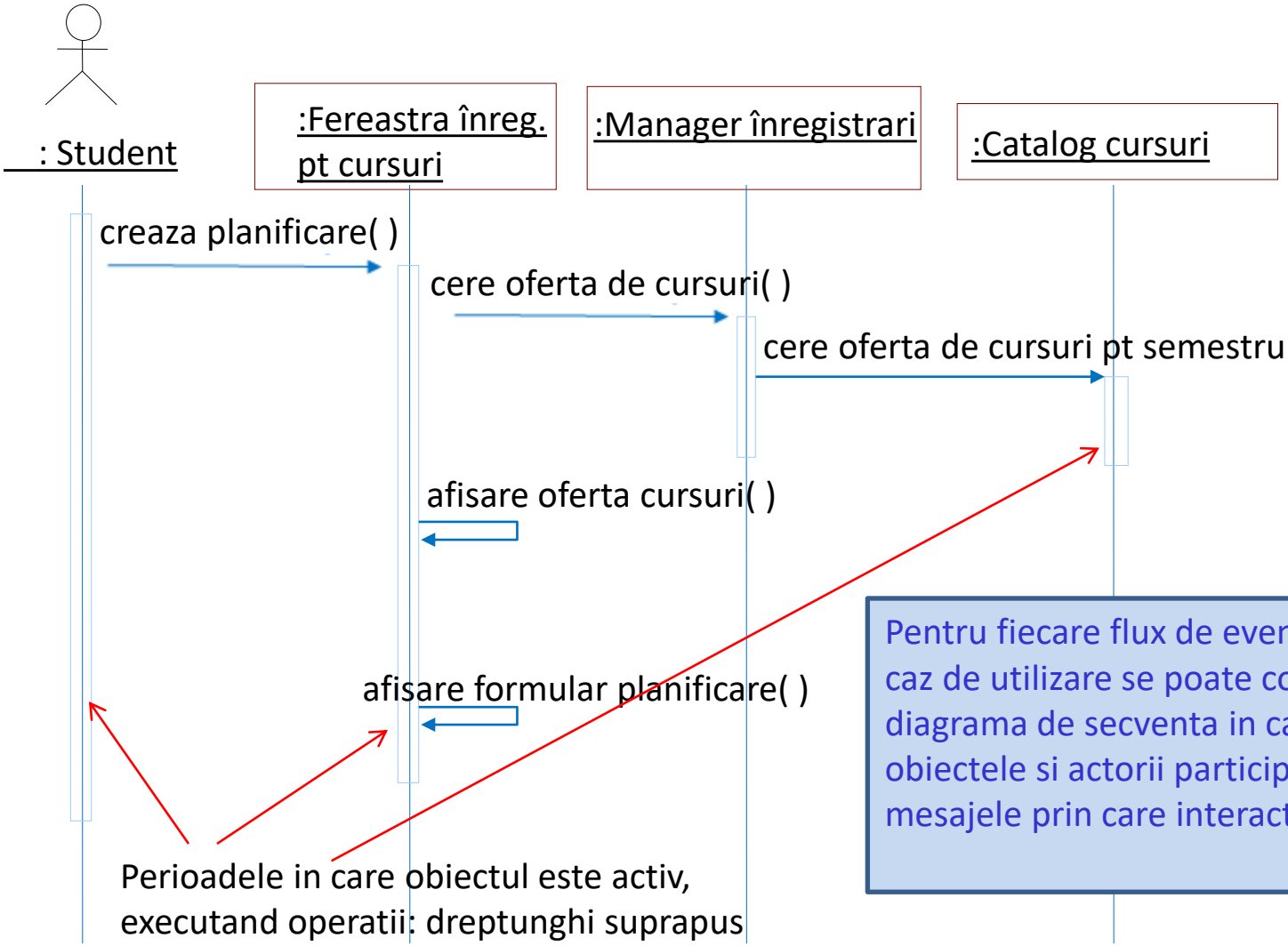
Reprezentarea fluxului de bază al cazului de utilizare “Imprumut” (1)



Reprezentarea fluxului de bază al cazului de utilizare “Imprumut” (2)



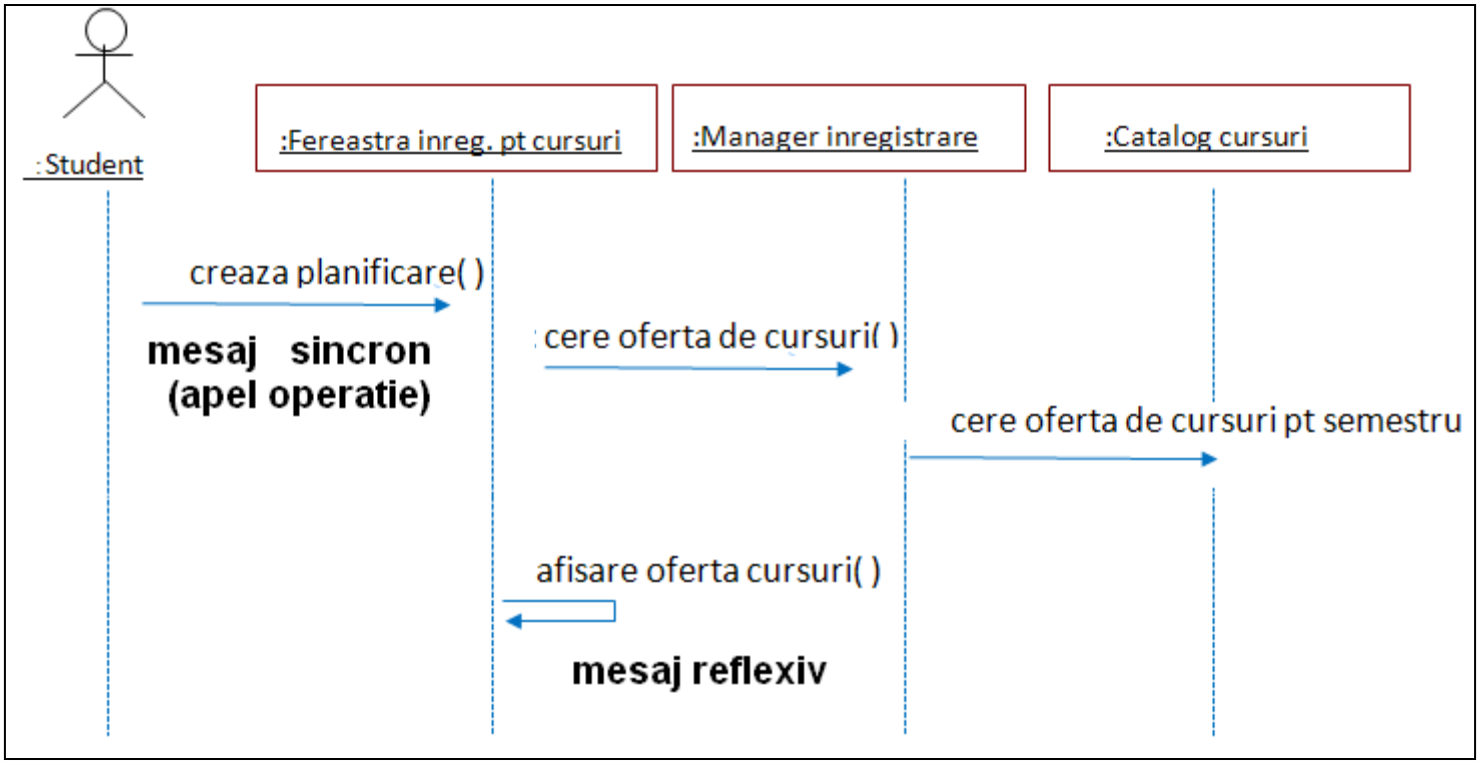
Reprezentarea unui scenariu de înregistrare la cursuri



Pentru fiecare flux de evenimente al unui caz de utilizare se poate construi o diagrama de secventa in care apar obiectele si actorii participanti precum si mesajele prin care interactioneaza.

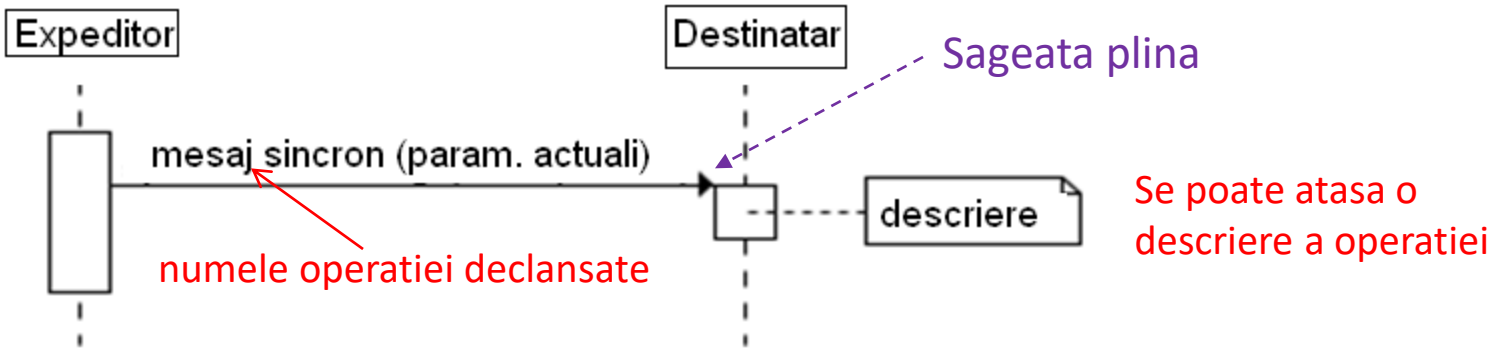
Perioadele in care obiectul este activ, executand operatii: dreptunghi suprapus peste linia de viata.

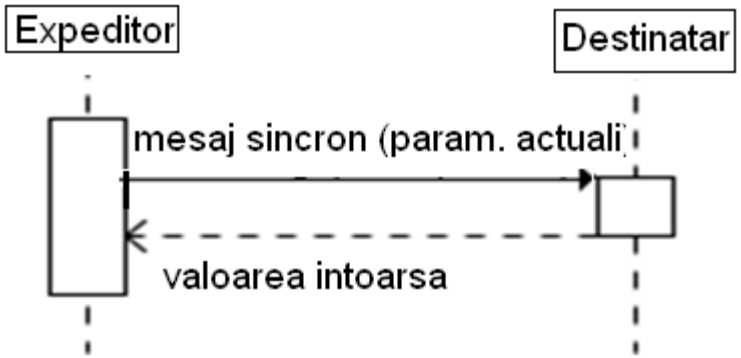
Diagrame de secventa
– Tipuri de mesaje - (1)



Diagrame de secventa
- Tipuri de mesaje – (2)

Mesaje sincrone: expeditorul asteapta terminarea operatiei declansate





Notă: reprezentarea revenirii la expeditor după un mesaj sincron este opţională

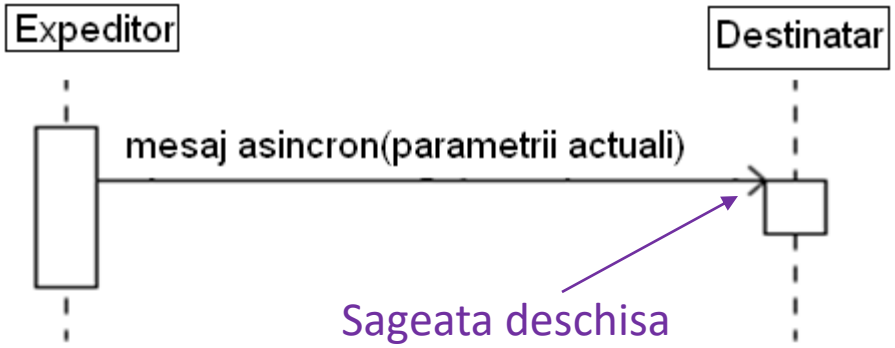
Diagrame de secventa
– Tipuri de mesaje – (3)

Mesaje asincrone: nu este intrerupta executia expeditorului pe durata executiei operatiei declansate

Exemple de mesaje asincrone, în proiectarea de detaliu:

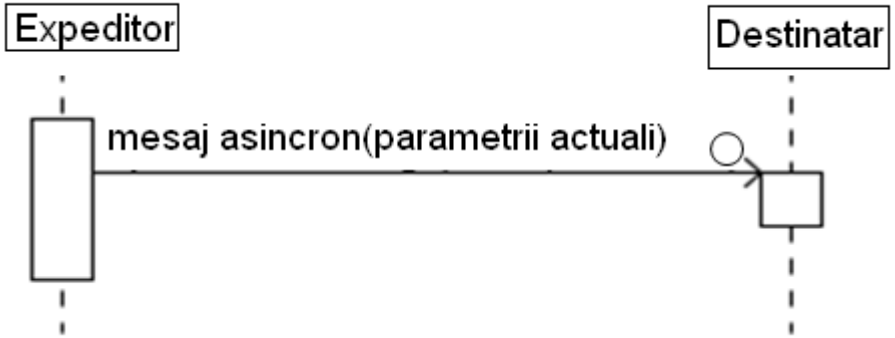
-Crearea unui fir de executie

-Comunicarea cu un fir de executie



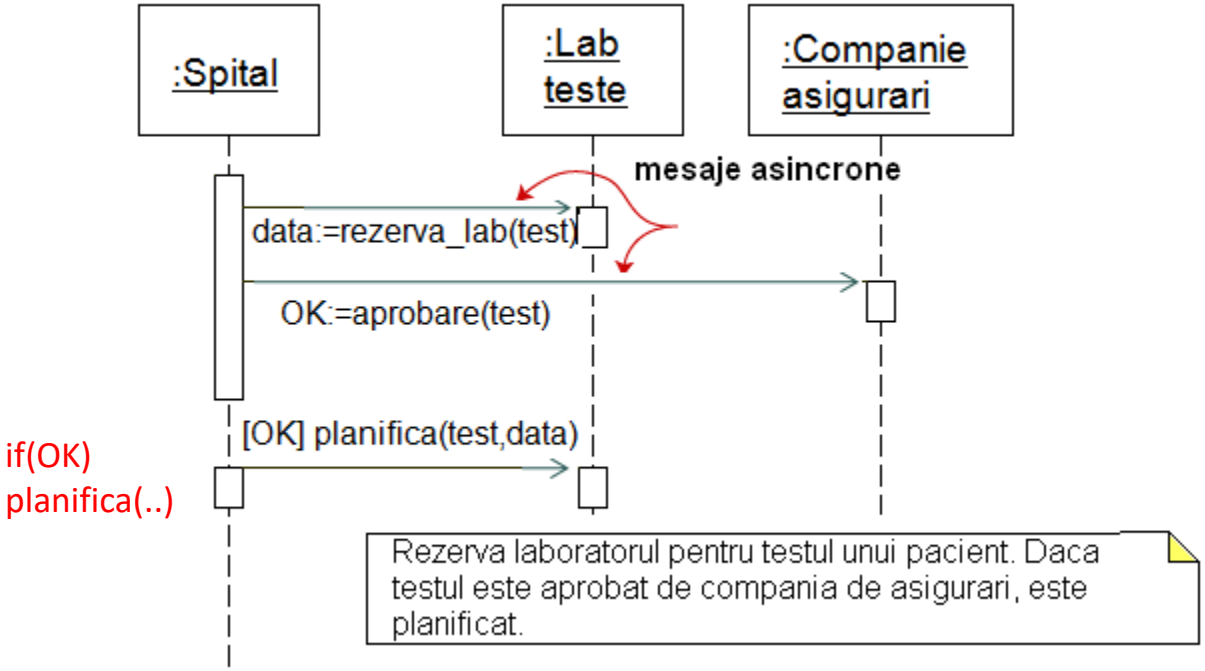
Mesaje cu time-out: trimitere sincrona cu blocarea expeditorului pe timp limitat

• Comunicatia nu are loc daca destinatarul nu ia in considerare mesajul în perioada în care expeditorul asteapta.



Diagrame de secventa
– Tipuri de mesaje – (4)

Mesaje asincrone: exemplu, în analiza cerințelor

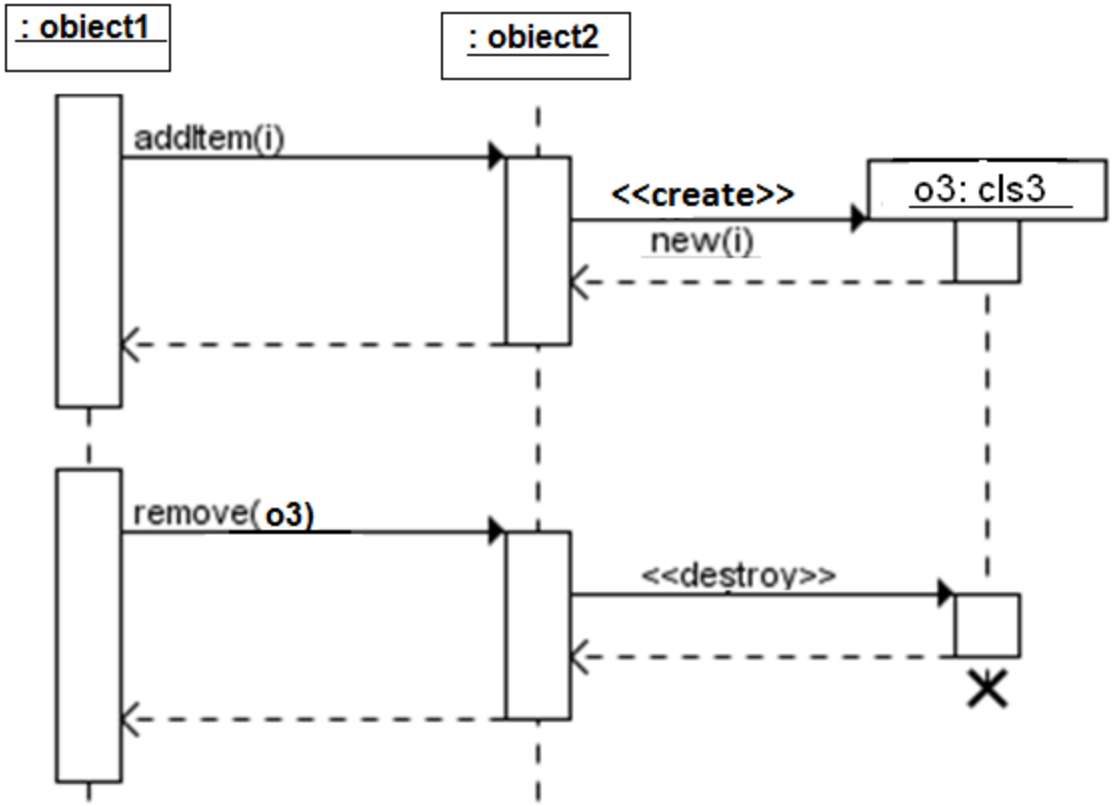


Diagrame de secventa

– Tipuri de mesaje – (5)

Crearea si distrugerea obiectelor: se reprezinta folosind stereotipurile create si destroy

Stereotip: mecanism de extindere UML

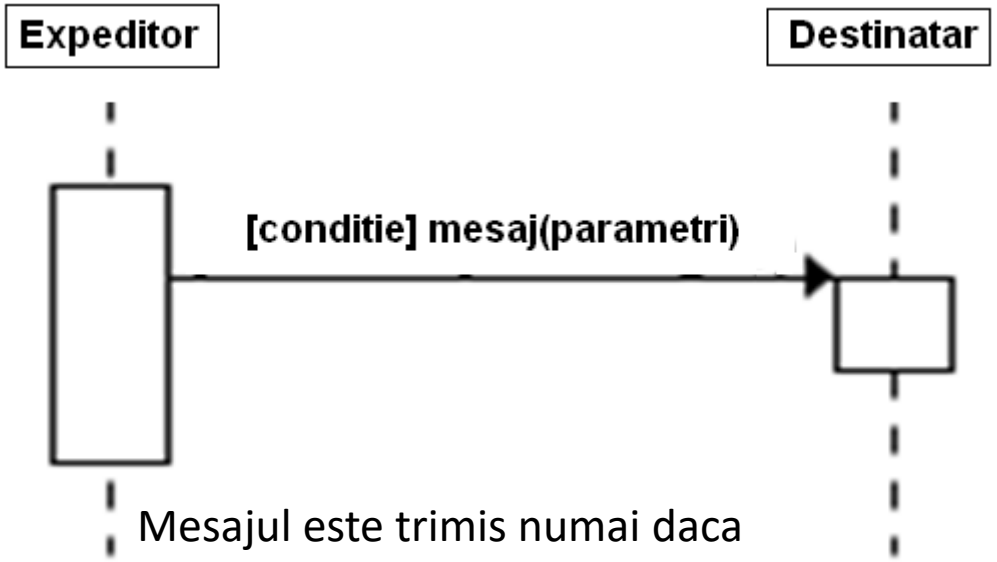


Diagrame de secventa

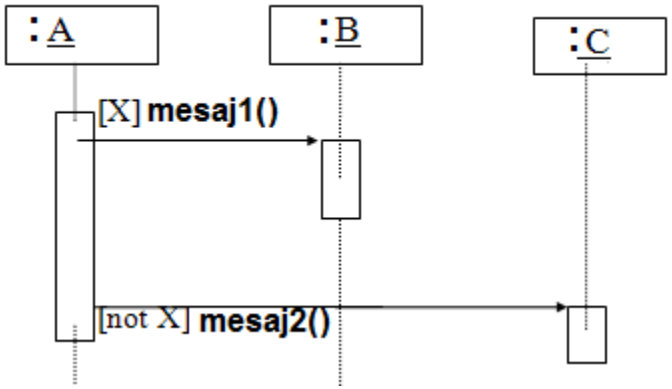
– Trimiterea condiționată a mesajelor -

Se reprezinta folosind gărzi

Gardă: [expresie booleana]

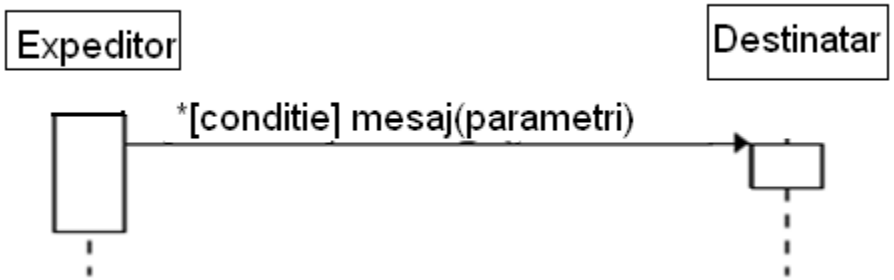


Mesajul este trimis numai daca rezultatul evaluarii conditiei este true.



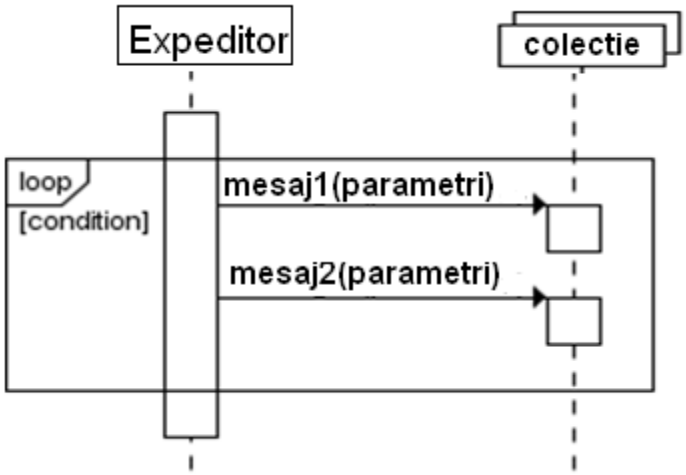
Diagrame de secventa

- Trimiterea repetată a mesajelor -



Trimite mesajul cat timp este indeplinita conditia

Trimite mesajul pe rand elementelor colectiei cat timp este indeplinita conditia

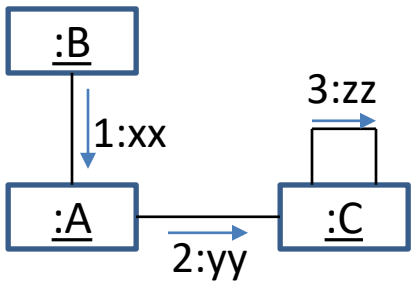


Ambele mesaje sunt transmise cat timp este indeplinita conditia

Diagrame de colaborare (1)

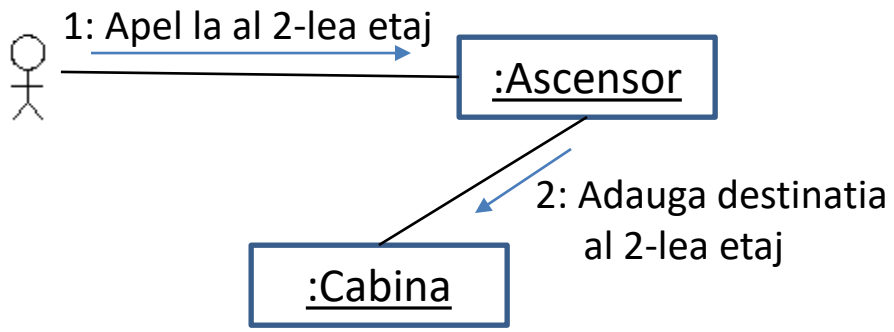
Evidențiaza relațiile structurale dintre obiecte (legaturile dintre ele).

Contin: obiecte, legaturile dintre obiecte, mesajele schimbate intre obiecte

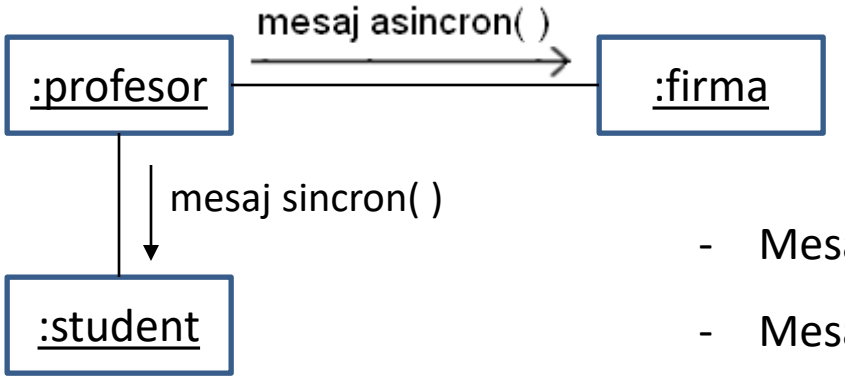


1. *Obiectul B trimite mesajul xx catre obiectul A*
2. *Obiectul A trimite mesajul yy catre obiectul C*
3. *Obiectul C isi trimite mesajul zz (mesaj reflexiv)*

- Relatiile structurale sunt reprezentate prin “legaturi” – linii care conecteaza obiectele.
- Mesajele schimbate intre obiecte sunt reprezentate de-a lungul legaturilor.
- Ordinea de trimitere a mesajelor este indicata printr-un numar amplasat în faţa mesajului.

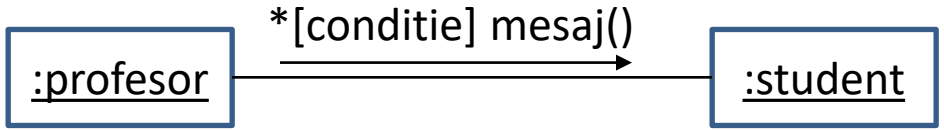


Diagrame de colaborare (2)



- Mesaje sincrone (sageata plina)
- Mesaje asincrone (sageata deschisa)

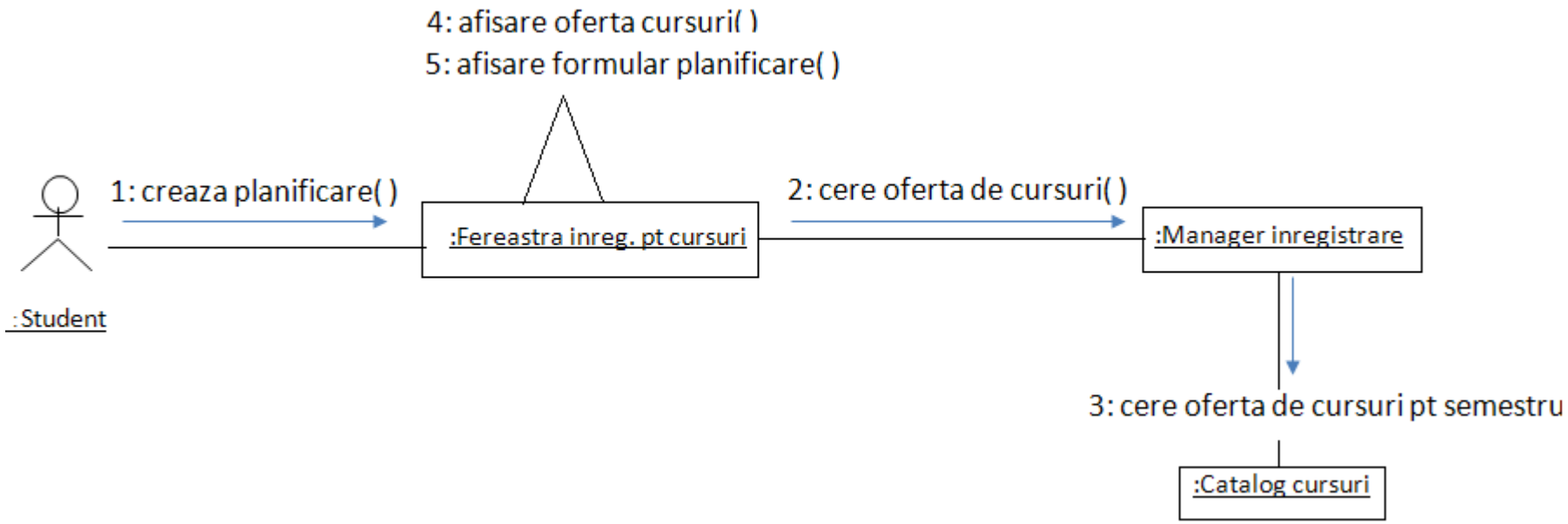
Trimiterea condiționată a unui mesaj



Trimiterea repetata a unui mesaj

Diagrame de colaborare (3)

Diagrama de colaborare echivalenta cu diagrama de secvență a scenariului "Inregistrare la cursuri"



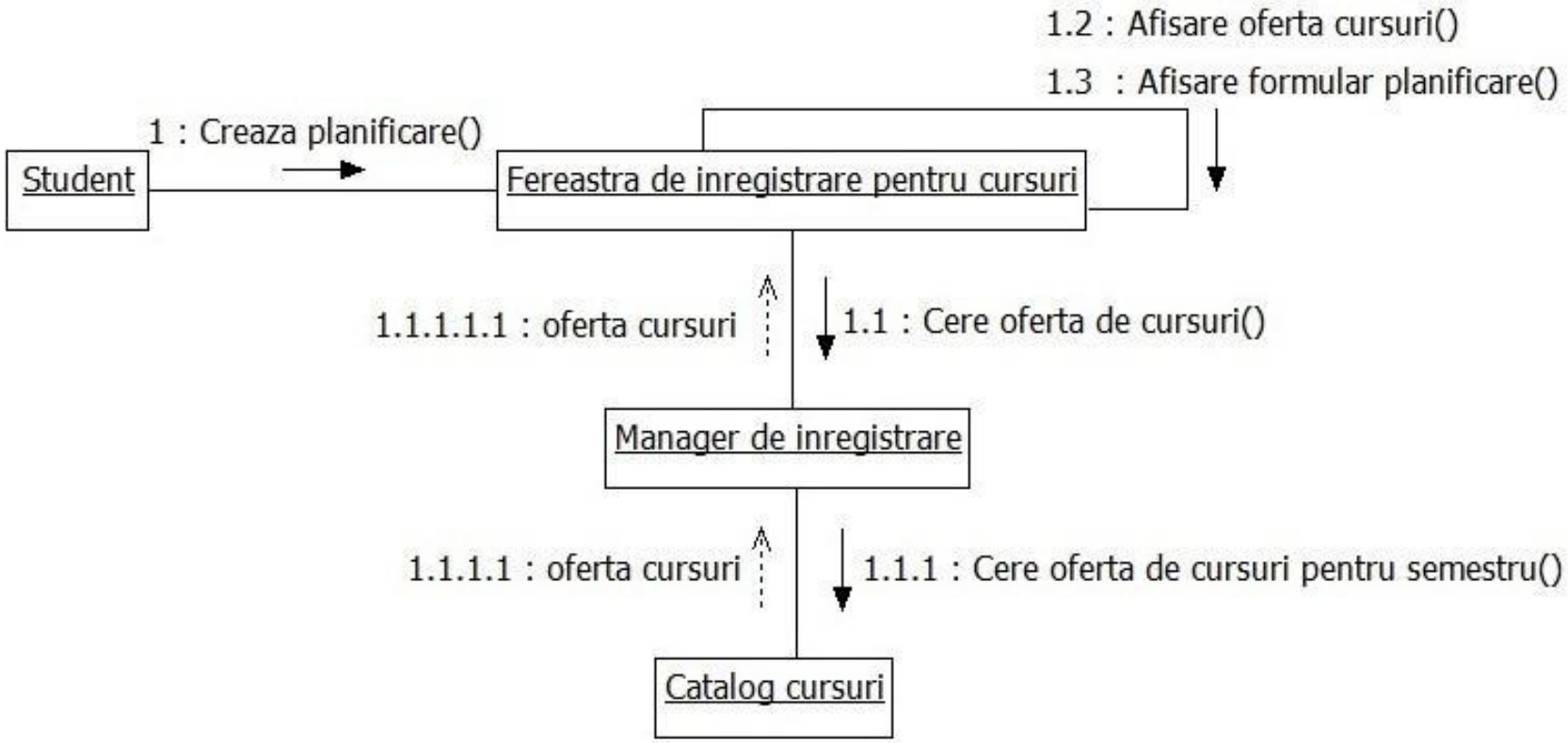
Intr-o diagrama de colaborare se pot folosi si *mesaje imbricate*, pentru a indica faptul ca un mesaj este trimis pentru tratarea unui alt mesaj.

Diagrame de colaborare (4)

- Mesaje imbricate -

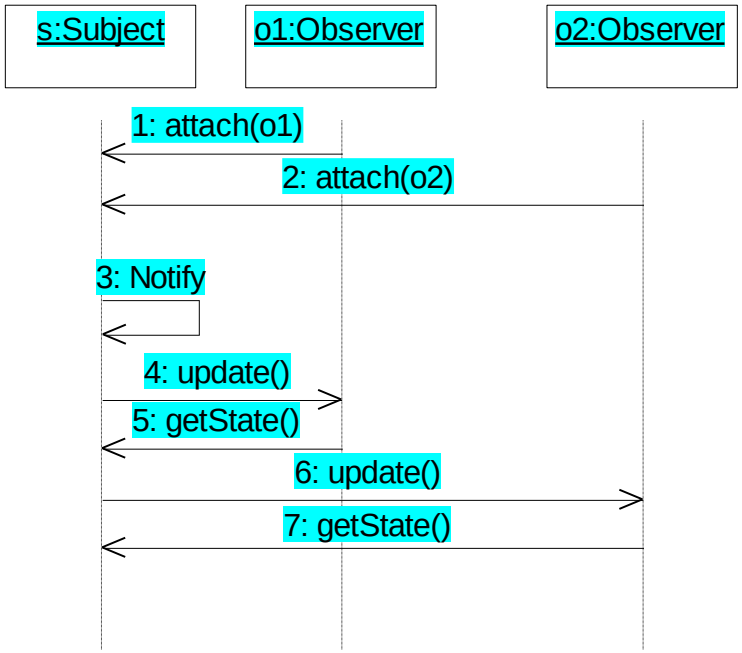
Mesaj imbricat: mesaj trimis in cursul procesarii unui alt mesaj.

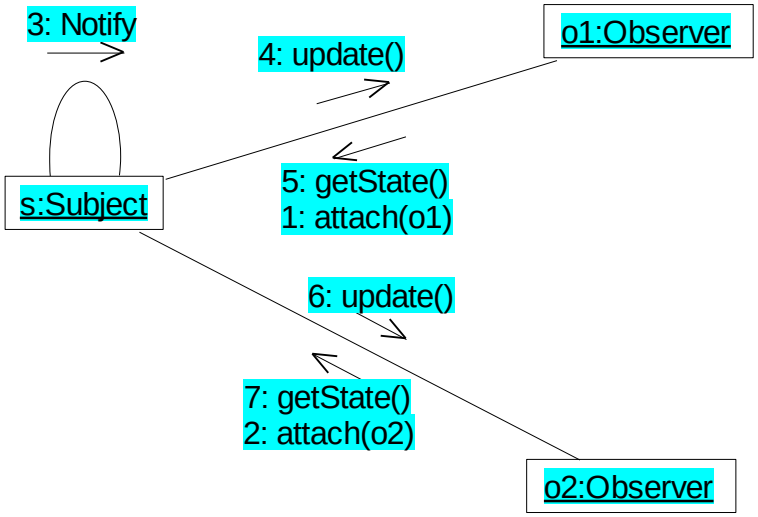
Exemplu: La executia operatiei “Creaza planificare()”, ca urmare a mesajului 1, se apeleaza operatia “Cere oferta de cursuri()” (mesajul 1.1), care apeleaza “Cere oferta de cursuri pentru semestru: (mesajul 1.1.1).



Echivalenta: diagrame de secventa – diagrame de colaborare

Sablonul de proiectare SUBIECT - OBSERVATOR





NOTĂ: Numerotarea mesajelor in diagrama de secventa nu este necesara!

Diagrame de interactiune in UML 2

Diagramele de interactiune in UML 2 sunt:

- Diagrame de secventa
- Diagrame de comunicare: diagramele 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

UML
Clase si diagrame de clase - 1

Prof. univ. dr. ing. Florica Moldoveanu

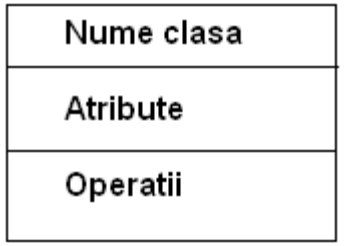
Curs *Ingineria programelor* – UPB, Automatică şi Calculatoare
2020-2021

Reprezentare obiecte si clase(1)

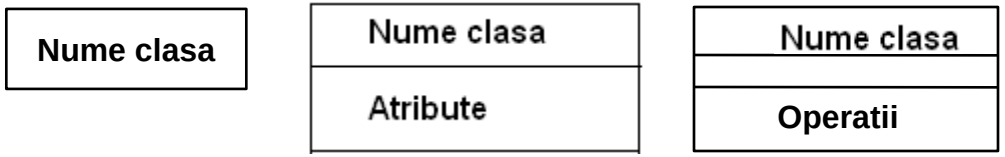
CLASA

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

Reprezentarea completa a unei clase: un dreptunghi cu 3 compartimente

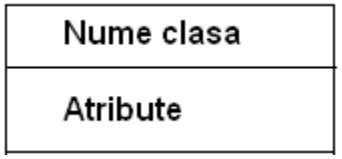


Compartimentul atributelor si cel al operatiilor pot lipsi:

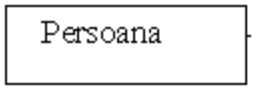


Reprezentare obiecte si clase (2)

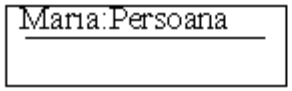
- **Clasele definite in etapa de „Analiza a cerintelor” contin, de regula, numai numele clasei si atributele (fara specificarea tipului fiecarui atribut).**
- **Ele se numesc clase conceptuale:**

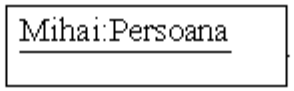


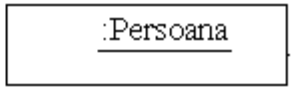
Clasa Persoana



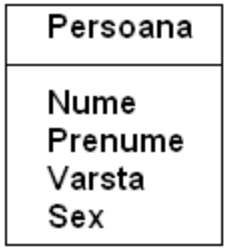
Obiecte din clasa Persoana

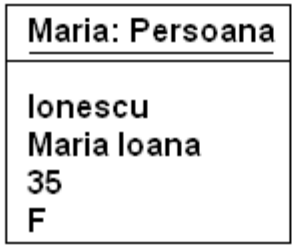






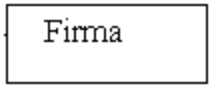
orice persoana



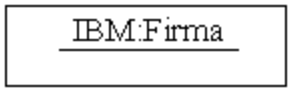


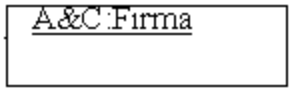
Un obiect este o instanta a unei clase.

Clasa Firma



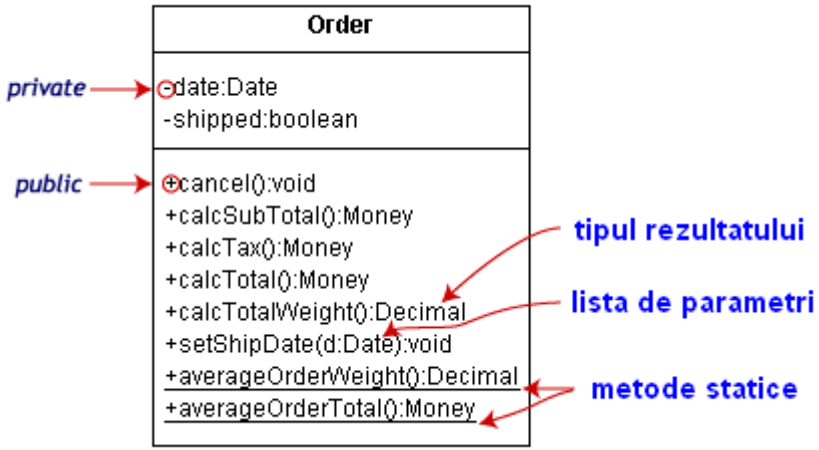
Obiecte din clasa Firma





Reprezentare obiecte si clase(3)

- **Reprezentarea detaliata a unei clase este construita in etapa de proiectare de detaliu.**
- Ea precizeaza in plus faţă de numele operatiilor si atributelor:
  - vizibilitatea informatiilor din clasa (+: public, -: private, #: protected)
  - tipul atributelor
  - lista de parametri a fiecarei operatii si tipul parametrilor.



Reprezentarea detaliata permite generarea automata a codului clasei.

Diagrame de clase

- O diagrama de clase reda un set de clase şi relaţiile dintre ele.

Exista 2 tipuri de relatii între clase:

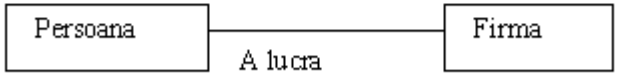
- asociere
- generalizare

legaturi: relatii intre obiecte

Legaturile dintre obiecte sunt instanțe ale unei relații de asociere dintre clasele lor.

Relatia de asociere dintre clase

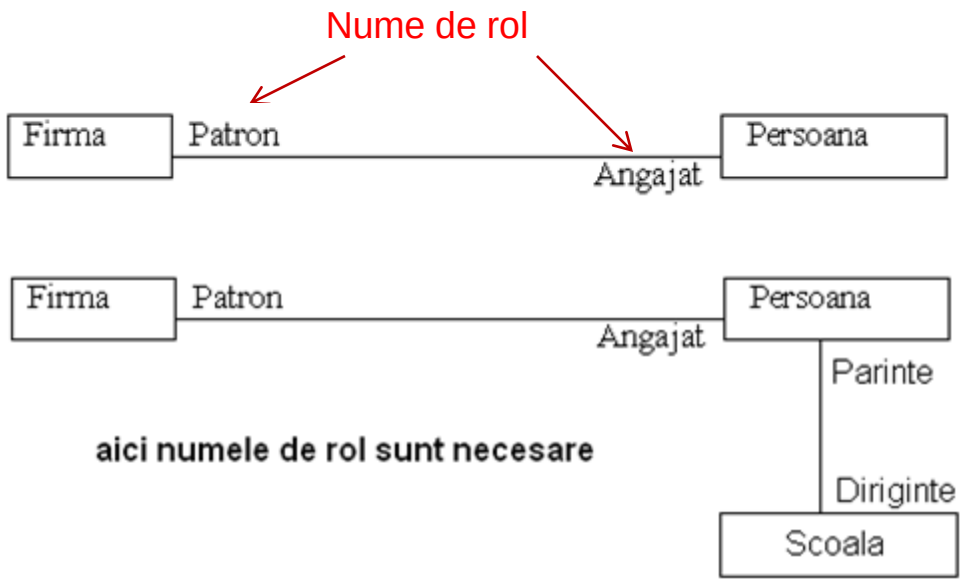
- o abstractizare a unui set de legaturi dintre obiectele claselor



asociere: relatie intre clase

Relatia de asociere (1)

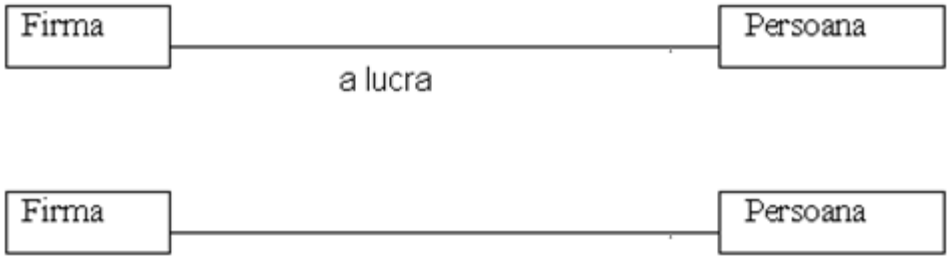
Extremitatilor unei asocieri li se pot atasa nume de ROL.



Rol: felul in care obiectele unei clase "vad" obiectele unei alte clase intr-o relatie de asociere.

Numele de rol clarifica rolurile pe care obiectele unei clase le joaca in asocierile cu diferite clase.

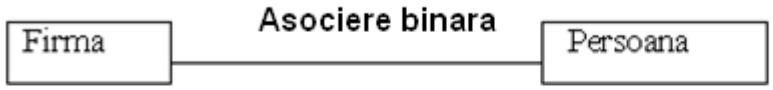
Asocierile pot avea nume

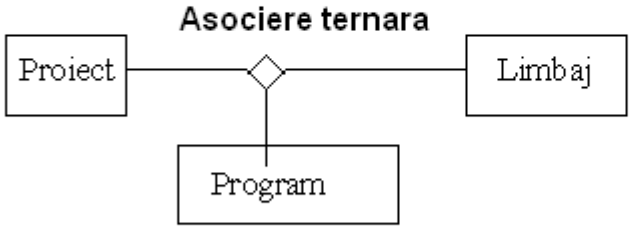


Numele de asociere este optional.

Relatia de asociere (2)

Aritatea asocierilor

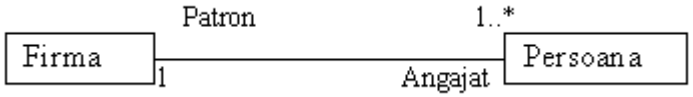




Orice asociere N-ara (cu mai mult de 2 capete) poate fi reprezentata numai in acest fel.

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

Multiplicitatea asocierilor: numarul de obiecte ale unei clase care pot fi legate unui obiect al celeilalte clase, la un moment dat.



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

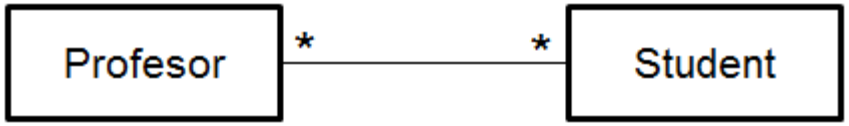
Un obiect al clasei firma va avea legaturi cu 1 sau mai multe obiecte ale clasei Persoana la un moment dat.

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

Relatia de asociere (3)

Obiectele claselor între care există o relatie de asociere au existență proprie şi sunt independente unele de altele. Ele pot fi create si distruse independent.

Exemplu :



La un moment dat, un profesor poate avea legaturi cu (zero sau) mai multi studenti si un student poate avea legaturi cu (zero sau) mai multi profesori.

Intre obiectele claselor asociate nu exista o relatie de apartenenta.

Legaturile dintre obiectele claselor asociate au un caracter temporar. Un student poate avea legaturi cu profesori diferiti la momente de timp diferite.

Relatia de asociere se implementeaza prin pointeri sau referinte. Obiectele claselor asociate între care exista legaturi la un moment dat « se cunosc » prin referinte.

Relatia de asociere (4)

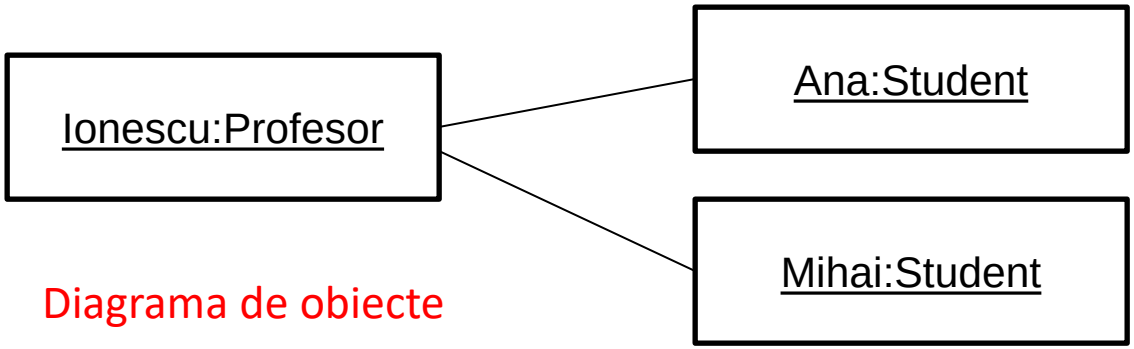


Diagrama de obiecte

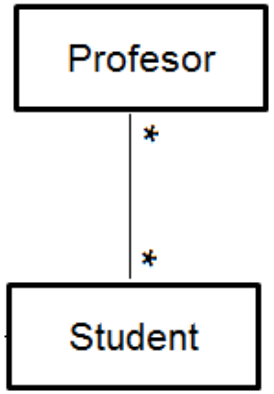


Diagrama de clase

Obiectele Ana si Mihai contin o referinta la obiectul Ionescu

iar obiectul Ionescu contine referinte catre obiectele Ana si Mihai.

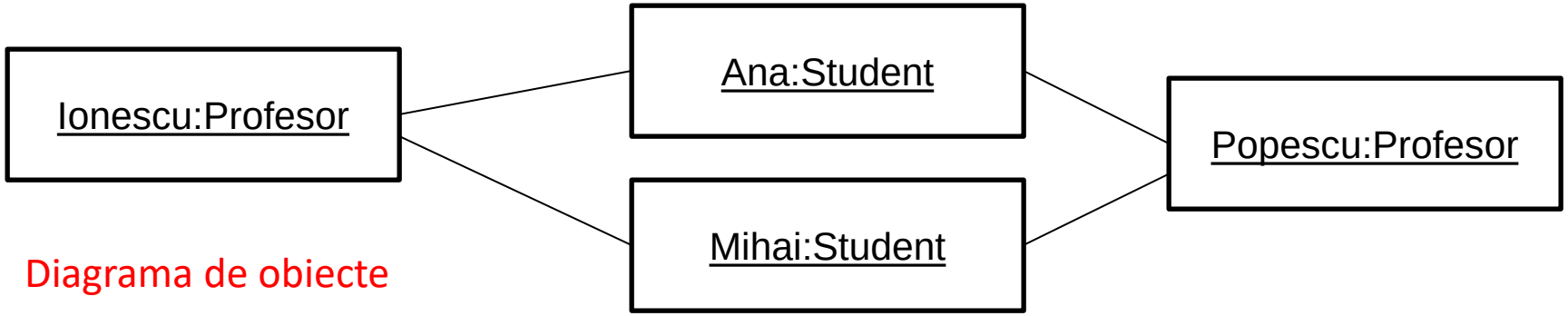


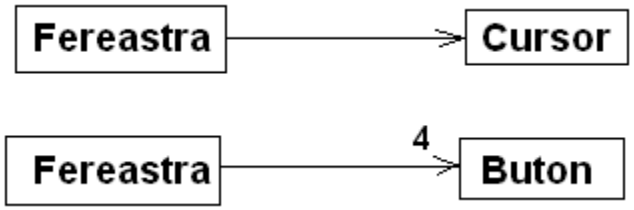
Diagrama de obiecte

O instanta a diagramei de clase la un moment de timp diferit

Relatia de asociere (5)

Asociere unidirectionala: navigabilitatea

O asociere unidirectionala este navigabila intr-un singur sens:



- Se reprezinta printr-o sageata la capatul "navigabil" al asocierii.
- Obiectul de la capatul navigabil este accesibil unui obiect de la cealalta extremitate; invers NU.

Implicit (fara sageată) asocierea este considerata bidirectionala.

Relatia de asociere (6)

Asociere unidirectionala

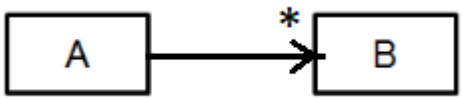
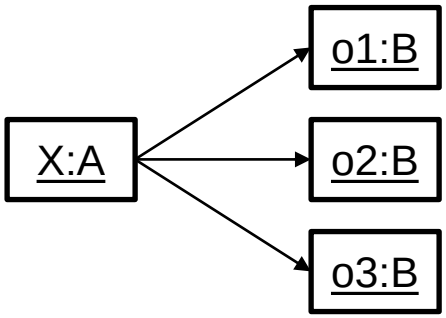


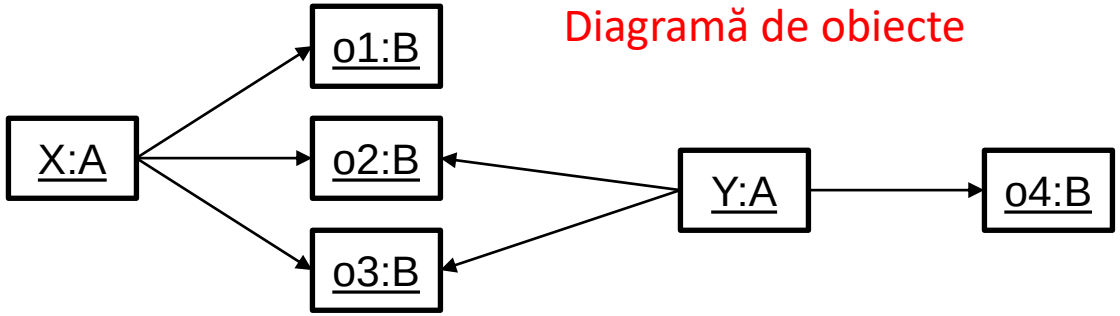
Diagrama de clase



Diagramă de obiecte

Obiectul X contine referinte la obiectele o1, o2, o3, care nu contin referinte la X.

Diagramă de obiecte



Obiectul Y contine referinte la obiectele o2, o3 si o4.

Obiectele o1,o2,o3,o4 “nu cunosc” obiectele X si Y.

Relatia de asociere (7)

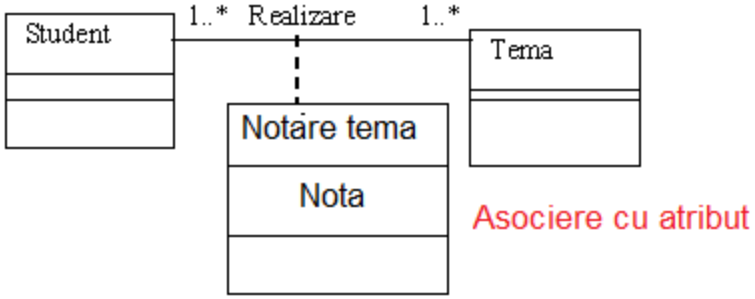
Clasa asociere

O clasa asociere permite reprezentarea atributelor unei asocieri.

Nota nu este un atribut al temei!

Un student trebuie sa realizeze mai multe teme si pentru fiecare tema primeste o nota.

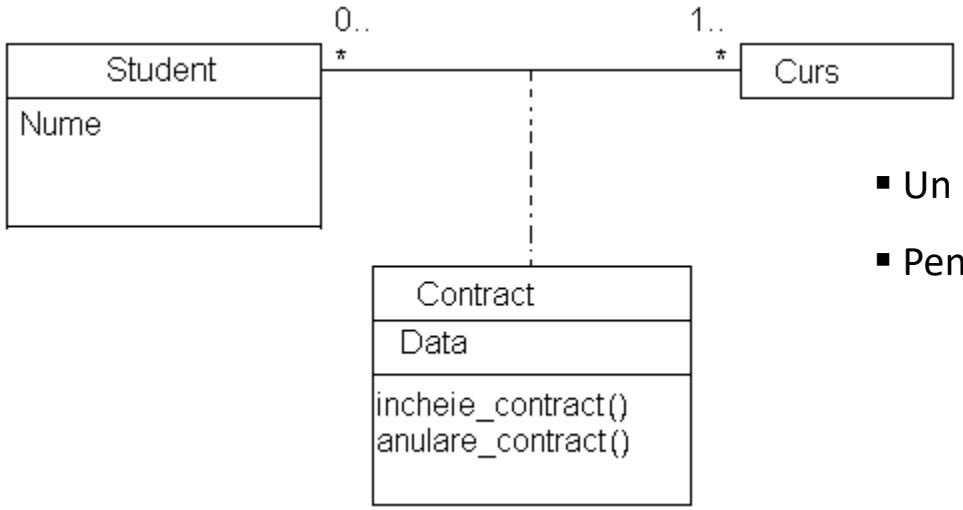
Nu toti studentii primesc aceeasi nota pentru aceeasi tema.



- Nota este un atribut al relatiei dintre clasele “Student” şi “Tema”.
- Se reprezinta ca atribut al clasei asociere, “Notare tema”.

Relatia de asociere (8)

Clasa asociere



- Un student participa la unul sau mai multe cursuri
- Pentru fiecare curs incheie un contract.

Exercitiu:

- O firma are mai multi angajati si mai multe departamente.
- Fiecare departament participa la mai multe proiecte in acelasi timp.
- Fiecare angajat lucreaza la mai multe proiecte in acelasi timp si este platit in functie de numarul de ore lucrate in fiecare luna pentru fiecare proiect. La un proiect lucreaza mai multi angajati.

Sa se reprezinte printr-o diagrama de clase clasele de obiecte din domeniul problemei si relatiile dintre ele.

UML
Diagrame de clase -2

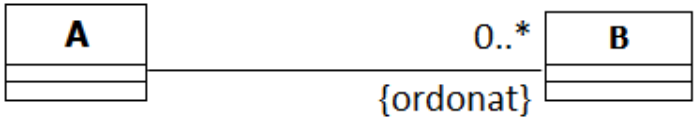
Prof. univ. dr. ing. Florica Moldoveanu

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

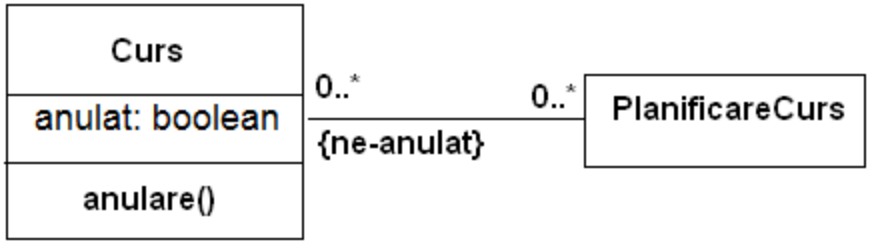
Relatia de asociere - continuare

Asociere constrânsă

O constrângere este o conditie care trebuie sa fie îndeplinita pe toata durata de existență a obiectelor claselor asociate.

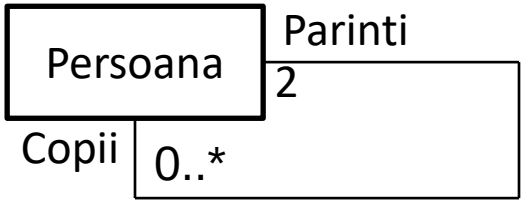


- Obiectele clasei B legate la un obiect din clasa A trebuie sa fie ordonate in orice moment.



- Se poate crea o legatura intre un obiect PlanificareCurs si un obiect Curs, numai daca valoarea atributului **anulat** este *ne-anulat*.

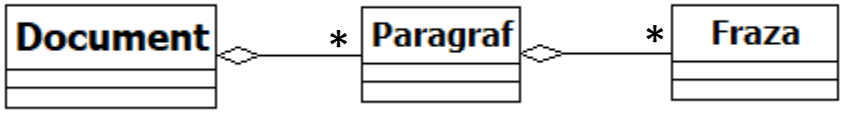
Asociere reflexivă: abstractizarea unor legaturi intre obiecte din aceeasi clasa



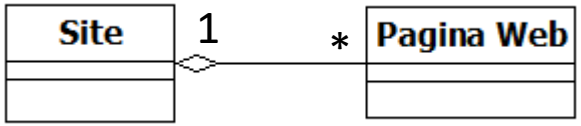
- Numirea rolurilor este în acest caz esentiala pentru claritatea diagramei.

Asocierea prin agregare (1)

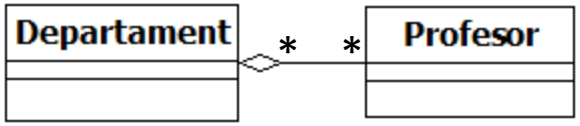
- Agregarea este o formă particulară de asociere, care exprimă un **cuplaj temporar de tipul "compus-componente", parte-întreg, parte-din.**
- Obiectele au propria lor existenta dar între agregat si componente exista o relatie de apartenenta la un moment dat.



O fraza F poate fi scoasa dintr-un paragraf si folosita in alt paragraf

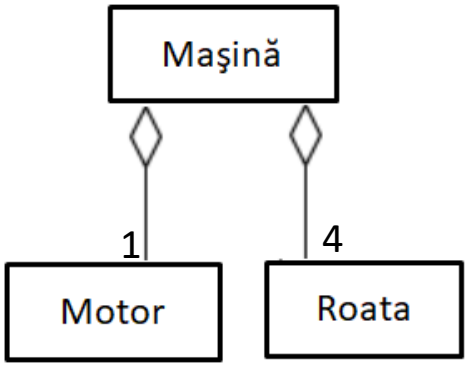


O pagina web, P, poate fi scoasa de pe un site, S, pentru a fi folosita în alt site. Site-ul S nu dispare dupa scoaterea paginii P. Distrugerea site-ului S nu afecteaza paginile pe care le avea.



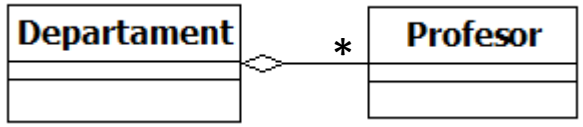
Un profesor, X, face parte dintr-un departament D, la un moment dat. La un alt moment de timp, X poate face parte din alt departament, din mai multe departamente sau din nici unul. Departamentul continua sa existe dupa plecarea unui profesor. Daca un departament dispare, profesorii care fac parte din departament se pot angaja in alte departamente.

Asocierea prin agregare (2)

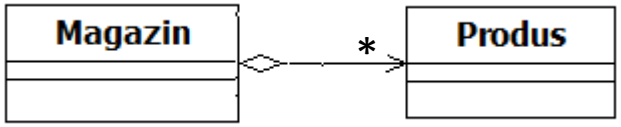


- O maşină are un motor si 4 roti.
- Atat motorul cât şi rotile pot fi inlocuite
- Distrugerea masinii nu are ca efect distrugerea motorului si a rotilor pe care le-a avut.

Asocierea prin agregare poate fi bidirectionala sau unidirectionala:



- Departamentul cunoaste profesorii sai.
- Un profesor stie din ce department face parte.



- Un magazin cunoaste produsele pe care le contine.
- Produsele nu cunosc magazinul în care se afla.

Agregarea mai este numita "Shared aggregation". Se caracterizeaza prin urmatoarele reguli:

- Agregatul si componentele au propria lor existenta, sunt create si distruse independent.
- Componentele unui agregat nu dispar atunci cand este distrus agregatul.
- O componenta a unui agregat poate face parte din mai multe agregate.

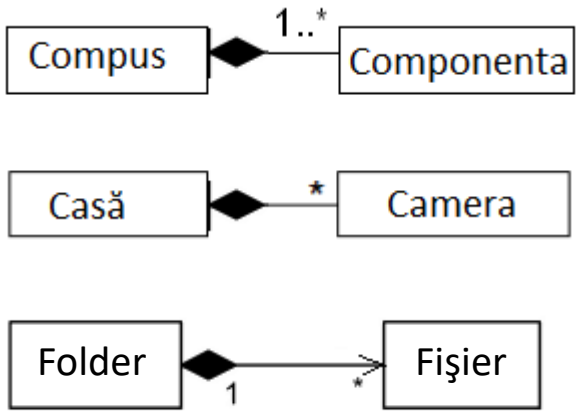
Este o relatie de tip compus-componente slabă.

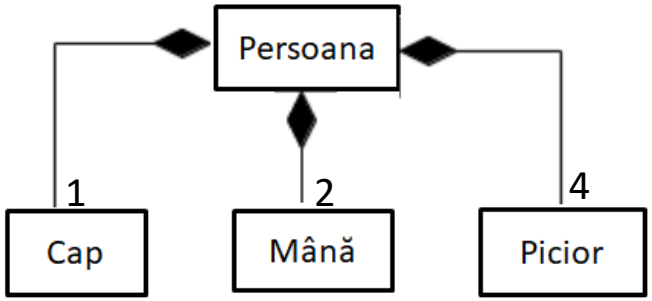
Asocierea prin compunere

Compunerea – numită şi "Composite aggregation"

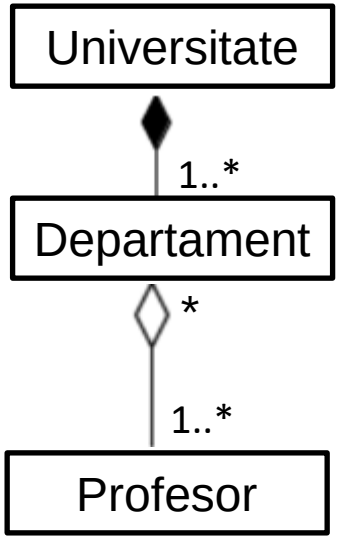
Este o agregare prin conţinere fizică, ce se caracterizeaza prin urmatoarele reguli:

- O componenta poate fi inclusa intr-un singur agregat la un moment dat
- Componentele nu pot exista de sine stătător, ele fac întotdeauna parte dintr-un agregat.
- Agregatul este responsabil de crearea și distrugerea componentelor sale, fie în mod direct, fie indirect (prin intermediul altor clase)
- Distrugerea agregatului are ca efect distrugerea tuturor componentelor sale.

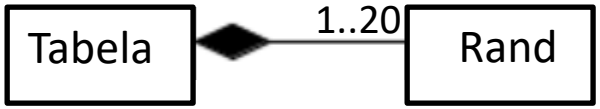




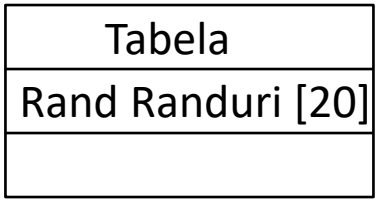
Diferente Compunere - Agregare



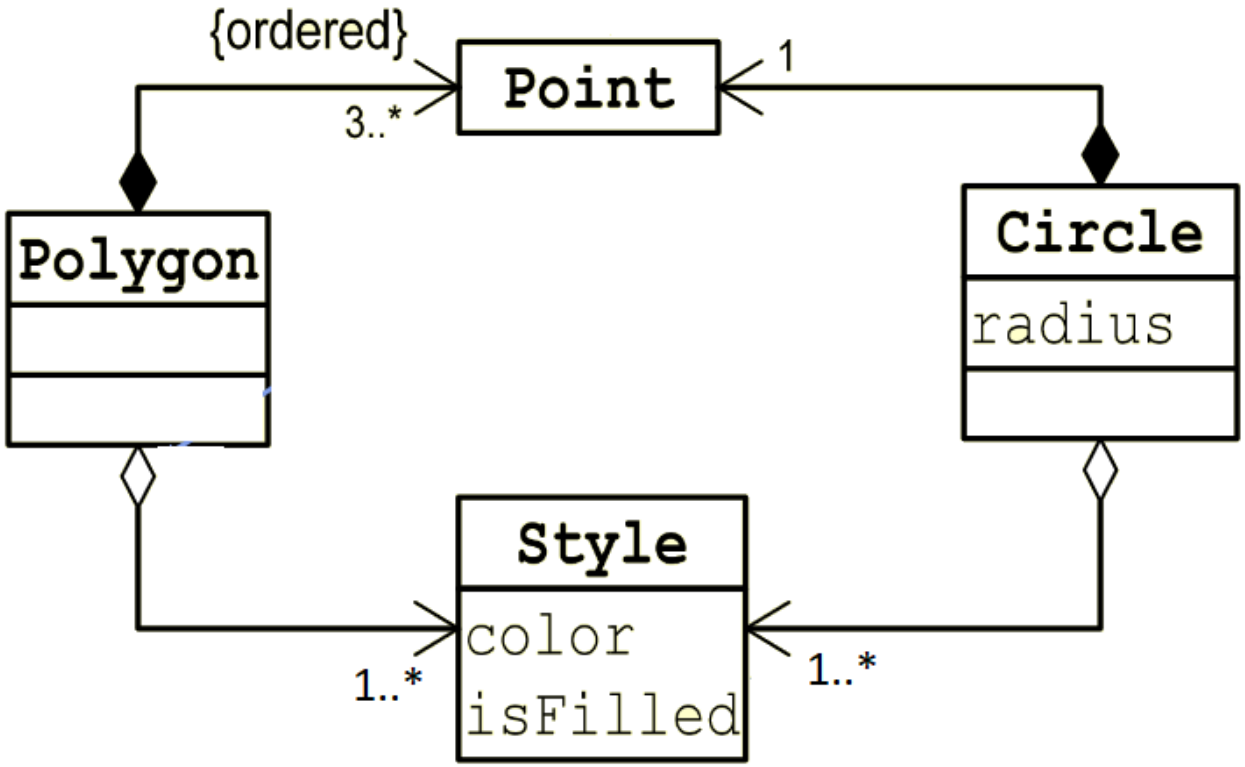
- O universitate este alcatuita din departamente (Calculatoare, Matematica, etc.)
- Fiecare department are mai multi pofesori.
- Daca universitatea se închide, departamentele dispar, dar profesorii din departamente continua sa existe.
- Un professor poate lucra in mai multe departamente
- Un department poate face parte dintr-o singura universitate



Echivalente



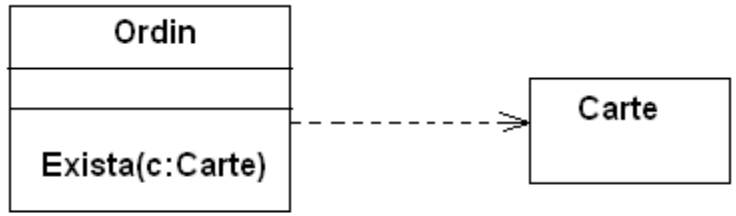
Diferente Compunere - Agregare



Relatia de dependenţă (1)

- Este cea mai slaba relatie între doua clase.
- Exprima de obicei o relatie temporara între obiectele a doua clase:

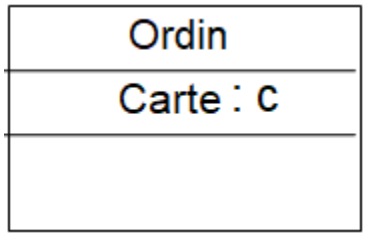
  obiectele clasei dependente interactioneaza pentru scurt timp cu obiectele clasei tinta.
- De ex., o clasa A depinde de o clasa B, daca o metoda a clasei A are un parametru de tip B.

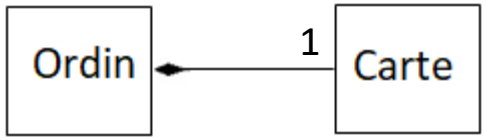


Client: elementul dependent Furnizor: elementul independent

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.

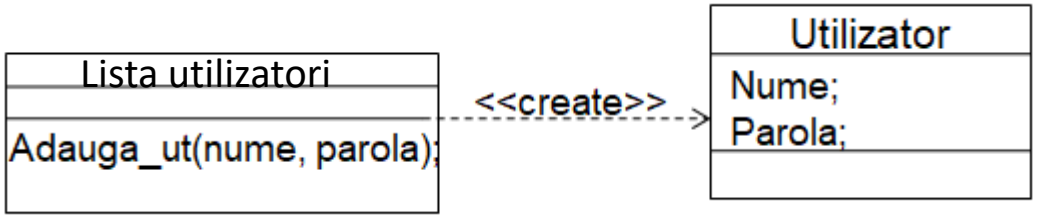




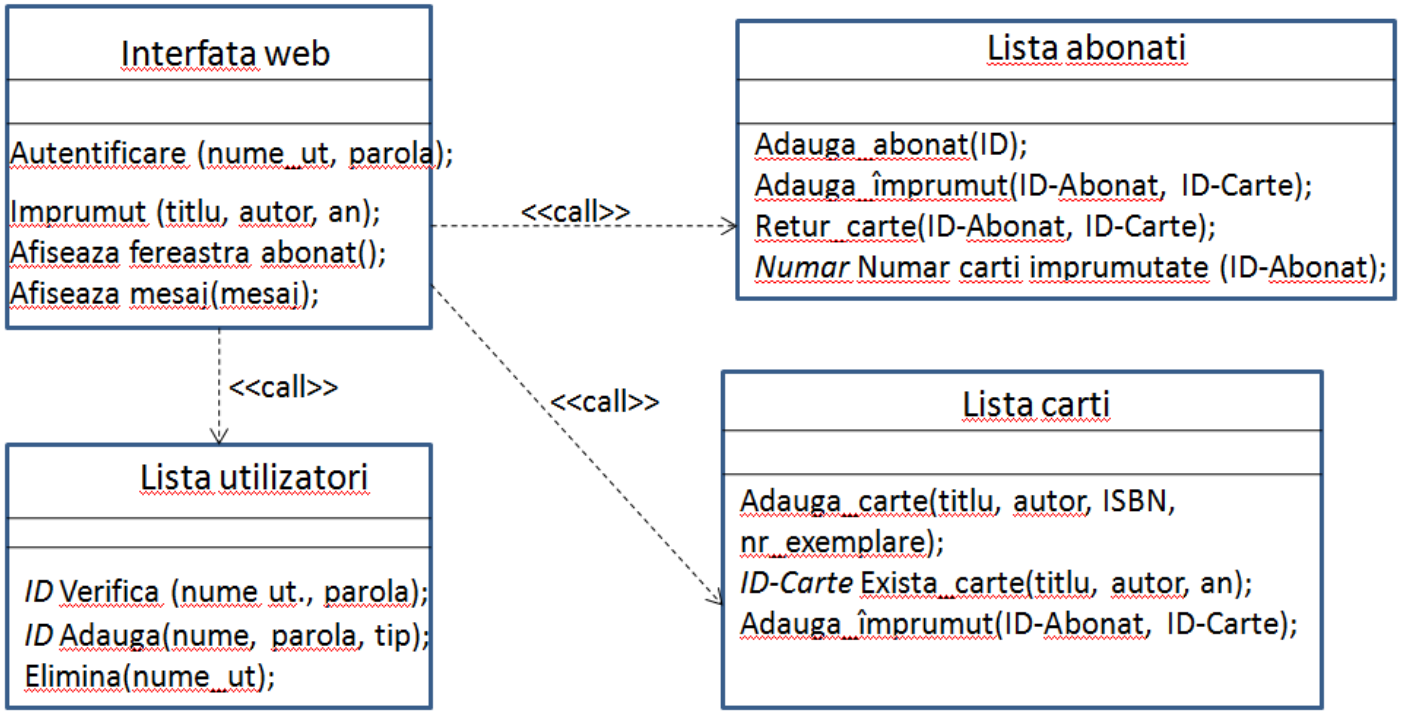
Un obiect Ordin contine un obiect Carte

Relatia de dependenţă (2)

Relația de dependență poate fi adnotată cu un stereotip care oferă informații despre natura relatiei. Exemple:



Obiectele clasei **Utilizator** sunt create de un obiect al clasei **Lista utilizatori** la apelul metodei Adauga_ut.



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

De la diagrama de secventa la diagrama de clase

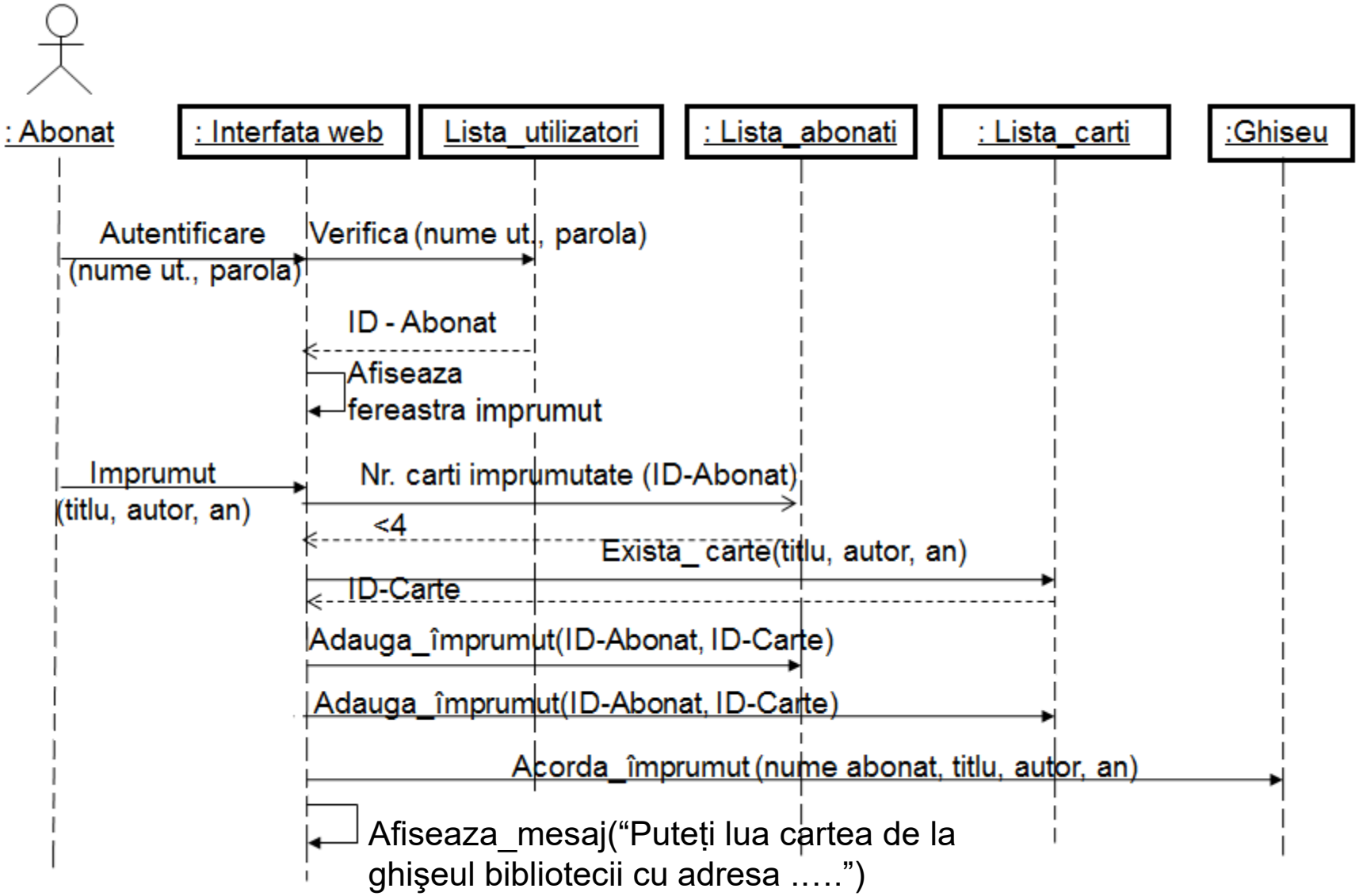
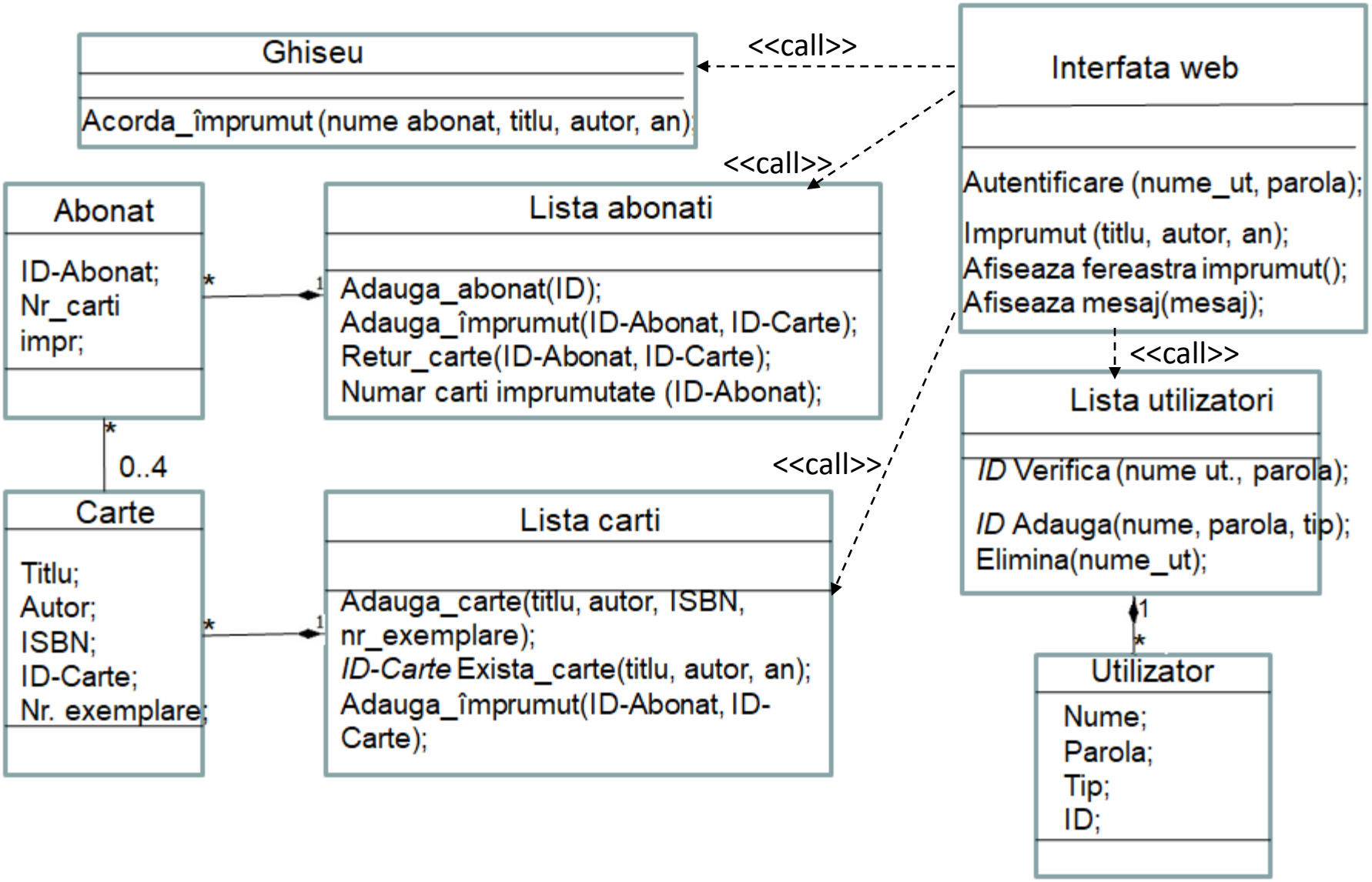


Diagrama de clase conceptuale a SGCB



Relatia de generalizare (1)

- Se bazeaza pe notiunile de: clasificare, generalizare, specializare si extindere.

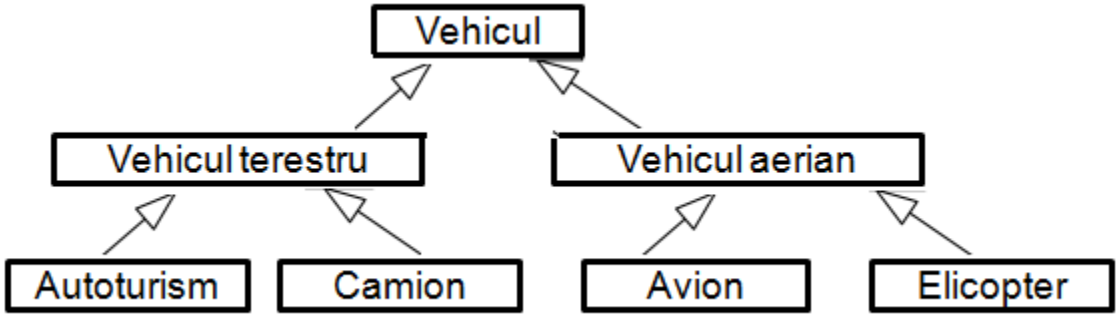
Generalizarea: factorizarea elementelor comune (atribute, operatii si constrangeri) ale unui ansamblu de clase într-o clasa mai generala, numită superclasă.

➢Relatia de generalizare din UML este mai abstracta decat relatia de mostenire din limbajele POO. Ea este mai adecvata etapei de analiza (există şi intre cazuri de utilizare!).

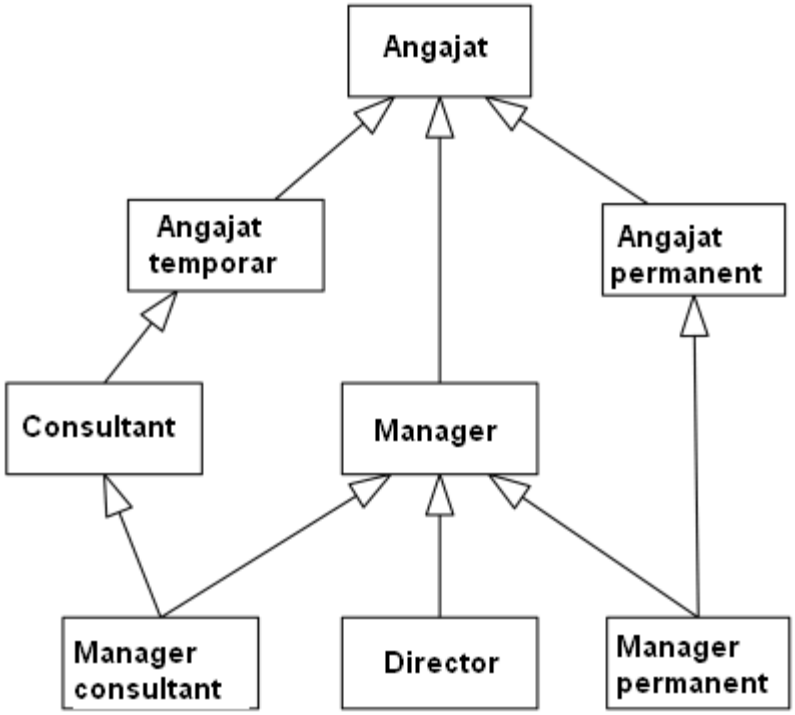
➢Prin clasificare si generalizare, universul problemei este divizat în parti independente care grupeaza obiecte prin afinitate. Modificarea unei parti antreneaza un minimum de modificari ale celorlalte → **arborele de mostenire**: **modificarea unui subarbore nu afecteaza ceilalti subarbori care au aceeasi radacina cu subarborele modificat.**

Specializare si extindere: capturarea particularitatilor unui ansamblu de obiecte, nereprezentate complet prin clasele existente intr-o structura ierarhica.

Relatia de generalizare (2)



Arborele de mostenire: fiecare clasa poate avea o singura super-clasa.

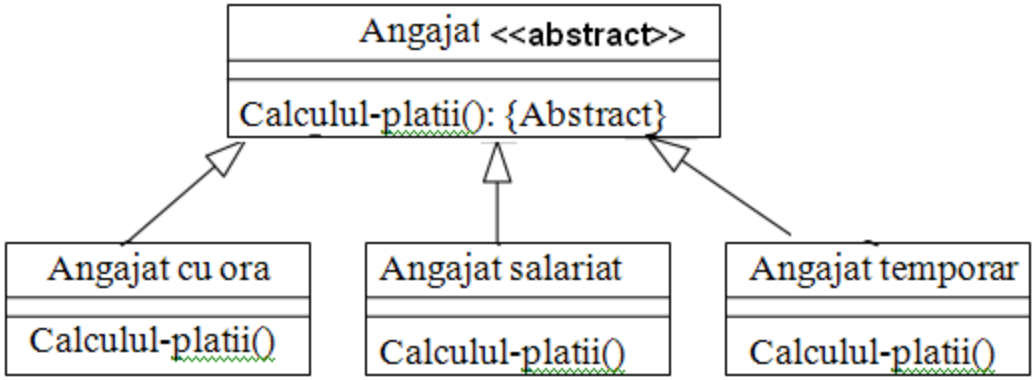


Graful de mostenire: clasele pot avea mai multe super-clase

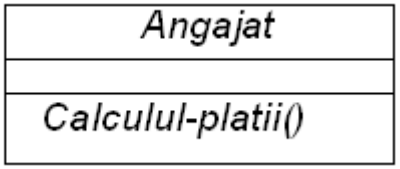
Relatia de generalizare (3)

Clasa abstracta

- Clasa care nu este instantiabila, dar ale carei clase descendente pot produce instante.
- Clasele abstracte înglobeaza mecanisme generale, ce pot fi particularizate prin specializare si extindere, în clasele descendente.
- O clasa este desemnata ca abstracta prin **stereotipul <<abstract>>** sau folosind caractere italice pentru numele său.
- O operatie abstracta este desemnata prin **constrangerea {Abstract}** sau folosind caractere italice pentru numele său.



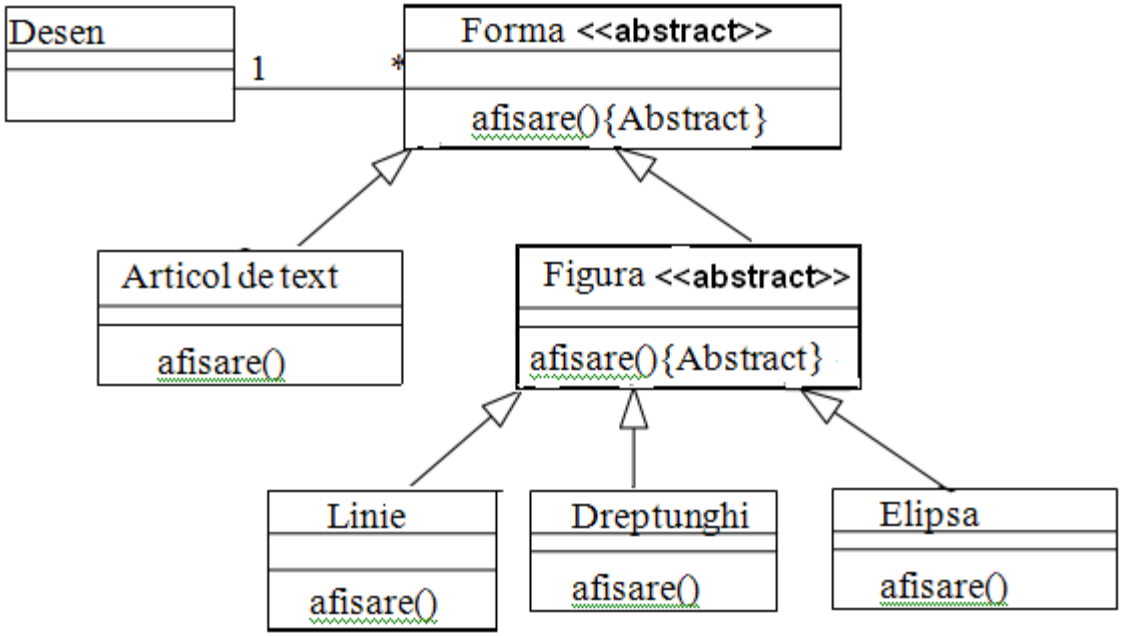
sau



Relatia de generalizare (4)

Polimorfismul

- Desemneaza proprietatea unui element de a lua mai multe forme.
- In Informatica, termenul desemneaza un concept al teoriei tipurilor, conform caruia un nume de obiect poate desemna instante ale unor clase diferite dar provenite din aceeasi arborescenta.
- Interactiunile dintre obiecte pot fi descrise folosind protocolul de comunicatie definit în super-clasele lor. Polimorfismul operatiilor desemneaza posibilitatea ca o operatie a unei superclase să declanseze operatii diferite ca raspuns la acelasi mesaj.

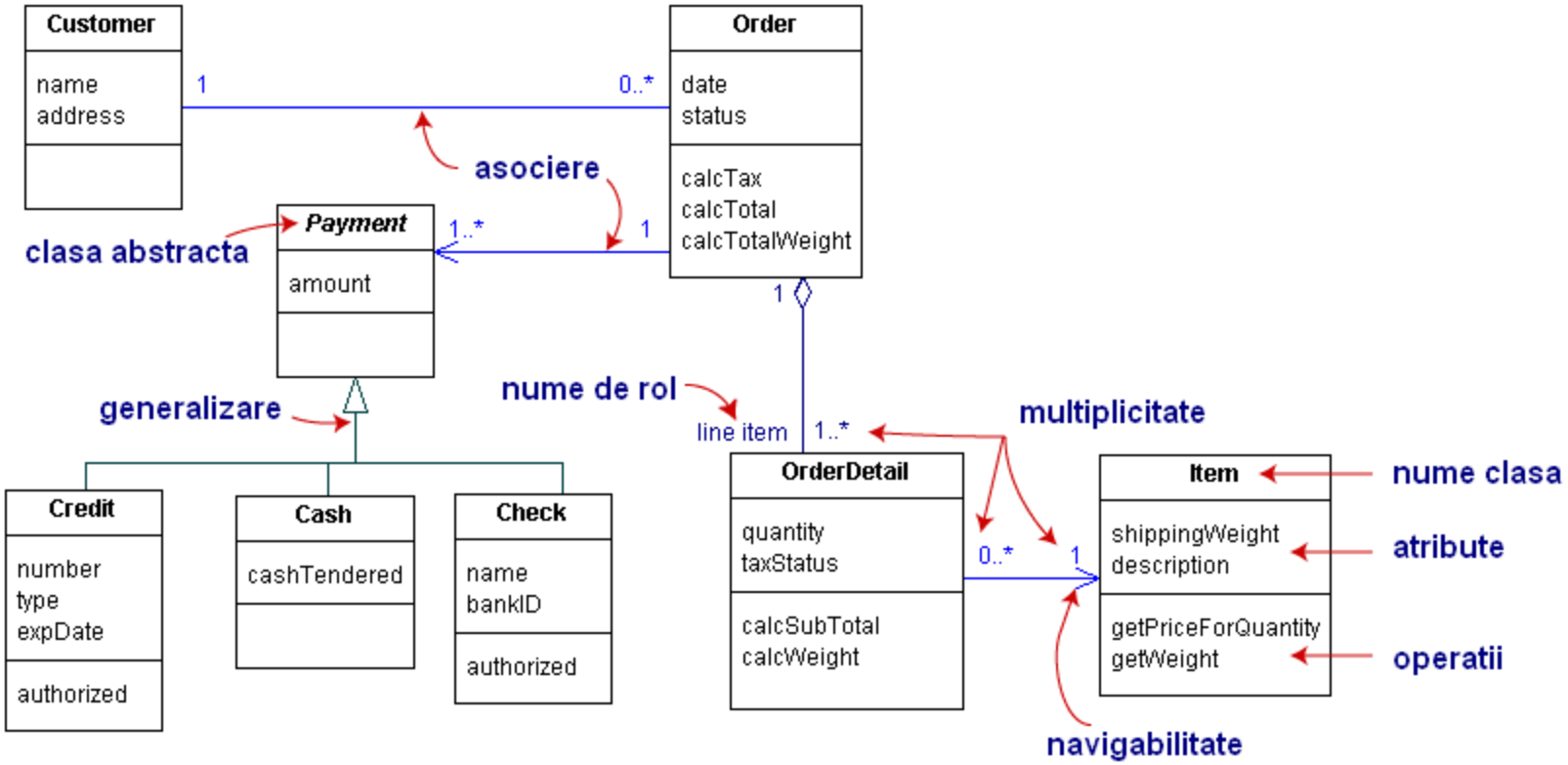


Forma * f;

f: pointer la un obiect Articol de text, Linie, Dreptunghi sau Elipsa

f-> afisare()
va fi executata operatia specifica obiectului punctat de f

Diagrama de clase conceptuale
- clase si relatii intre clase -



Diagrame de clase: utilizari

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

- Clasele corespund entitatilor (conceptelor / obiectelor) 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 (de proiectare) al sistemului.
- De regula, nu se definesc tipurile atributelor si nici cele ale parametrilor operatiilor.

2) In proiectarea de detaliu si implementare

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

Diagrame de obiecte conceptuale (1)

- O diagrama de obiecte este o instanță a unei diagrame de clase.
- Obiectele: instante ale claselor
- Legaturile dintre obiecte: instante ale relatiilor dintre clase

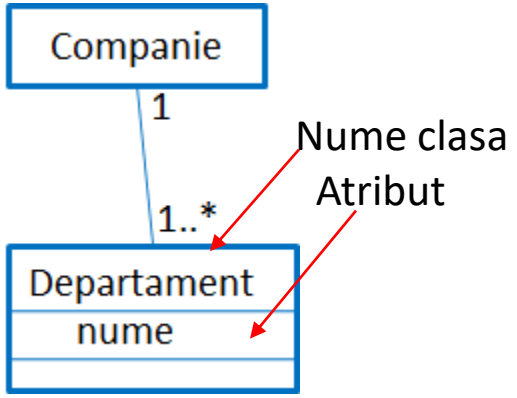
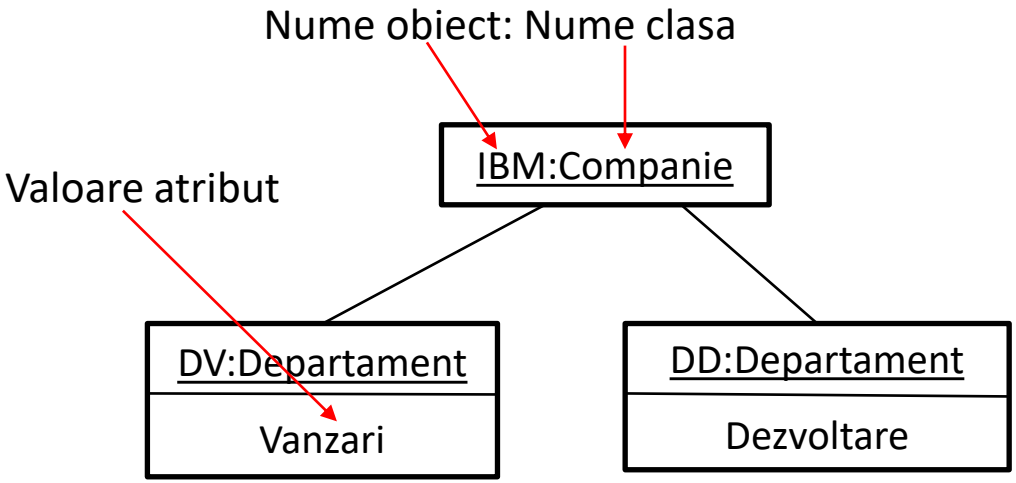


Diagrama de clase

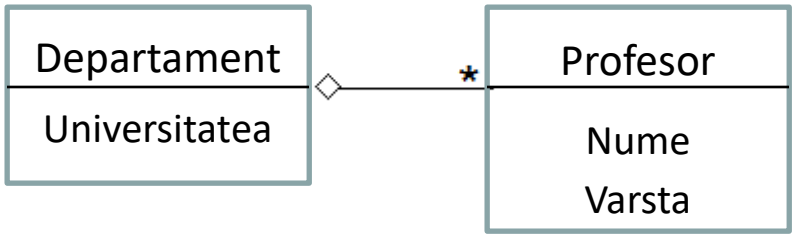


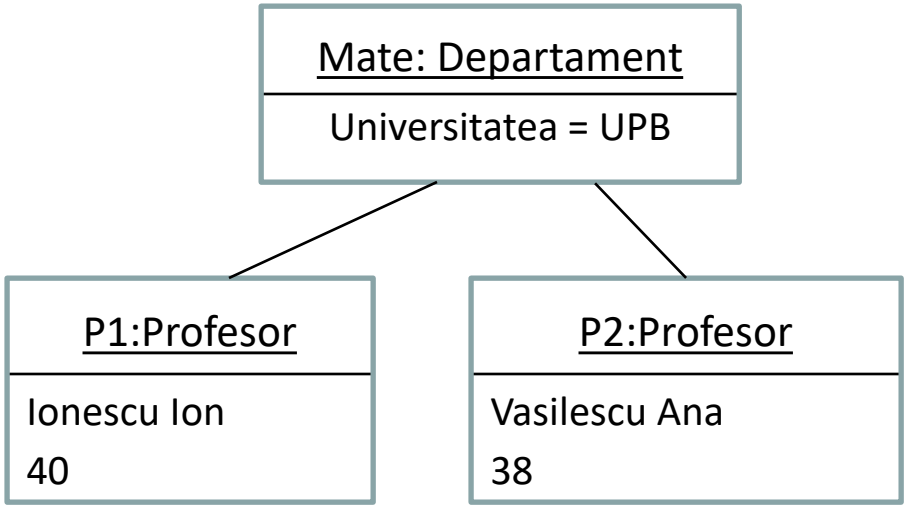
Diagramă de obiecte

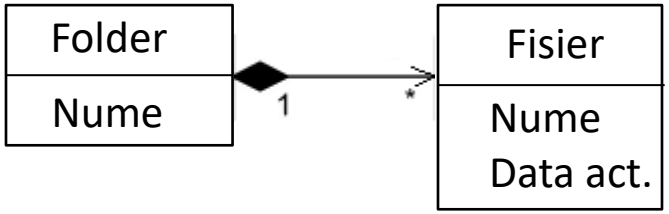
- Diagramele de obiecte conceptuale se folosesc pentru:

- a ilustra d.p.d.v. structural starile mai importante la executia unui sistem.
- a reda legaturile dintre obiecte la diferite momente ale executiei sistemului

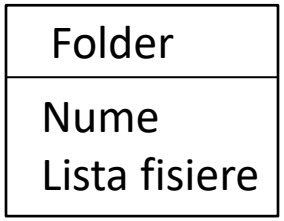
Diagrame de obiecte conceptuale (2)

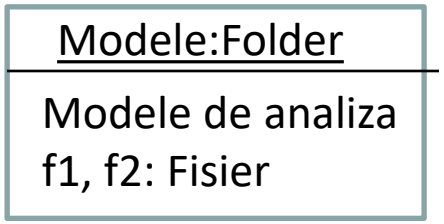


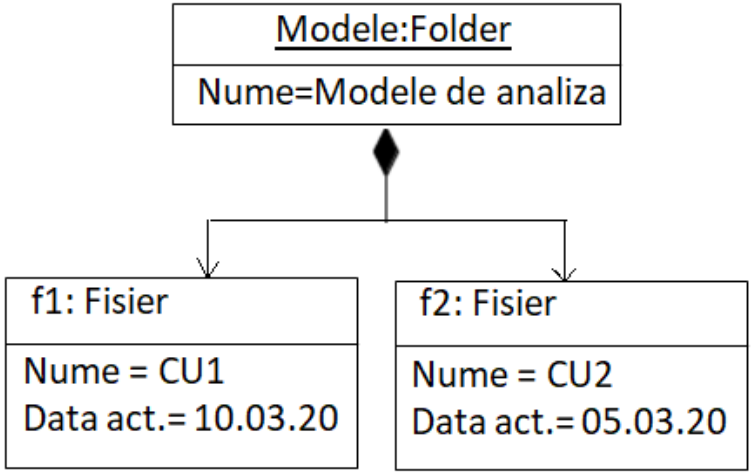




echivalent







Obiectul Modele contine obiectele f1, f2.

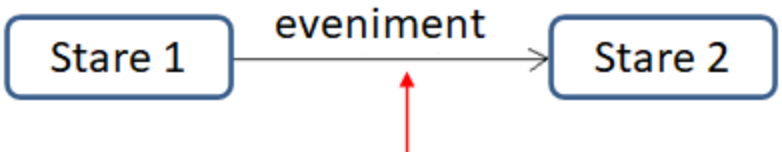
Diagrame de stari (State machine diagrams)

Prof. univ. dr. ing. Florica Moldoveanu

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

Diagrame de stari - utilizari

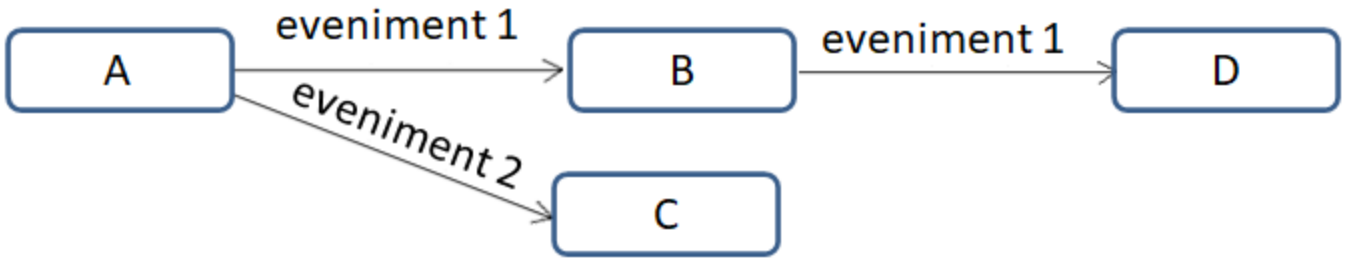
- Se folosesc pentru a reda aspecte dinamice ale unui sistem.
- O diagrama UML de stări redă stările unei entităti într-o perioadă de timp şi modul în care entitatea răspunde la diferite evenimente, prin tranziții dintr-o stare în alta.



Tranzitia din starea 1 in starea 2 datorata evenimentului

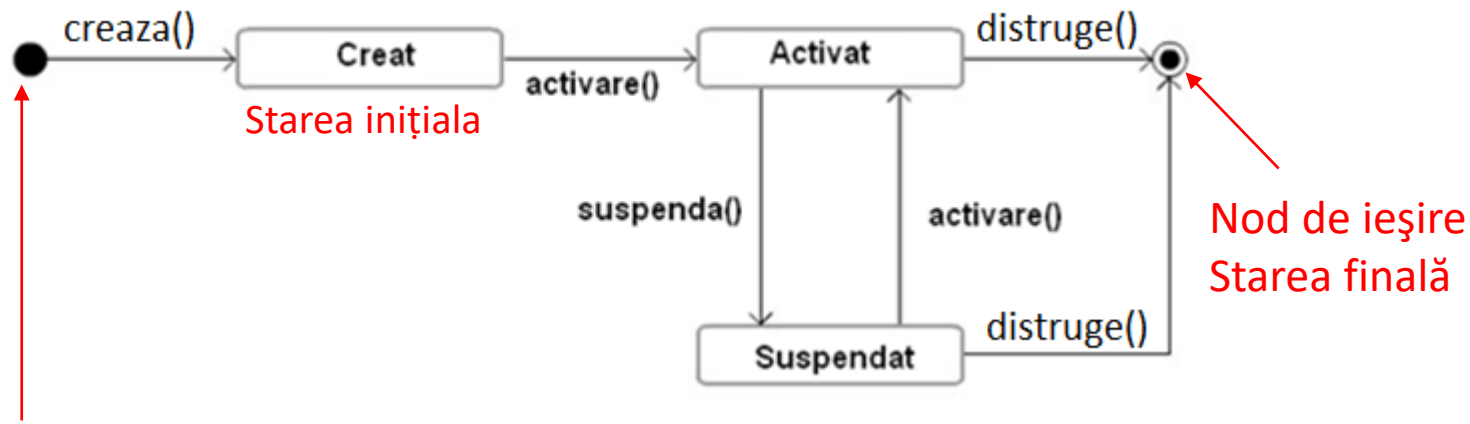
- **Principala utilizare**: documentarea claselor de obiecte cu un comportament reactiv.
- O diagramă de stări redă comportarea unui obiect dependentă de starea sa curentă.

Un obiect răspunde diferit la un eveniment în funcție de starea sa curentă.



Stari si tranzitii (1)

Exemplu: stările unui obiect sale şi tranzitiile determinate de apelul operatiilor (clasei) sale.

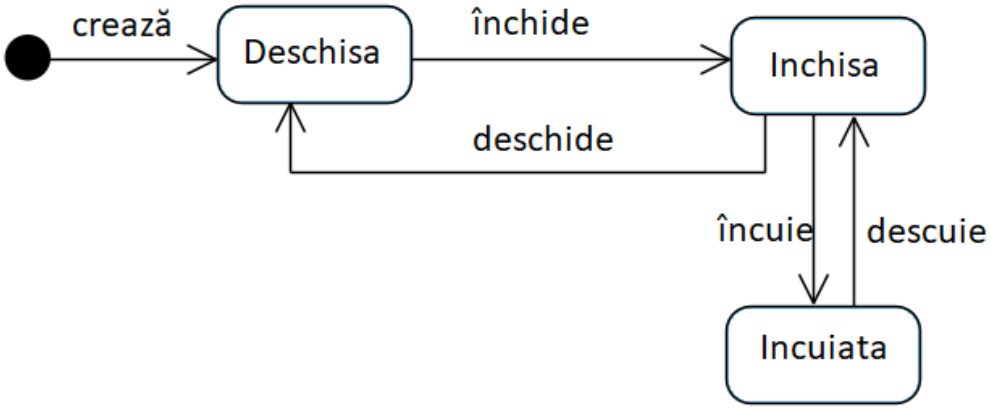


Nod de intrare (pseudo-starea initiala), din care se trece automat în starea inițială

❖ **Nodul de ieşire poate lipsi dintr-o diagramă de stări.**

Exemplu: stările unei uşi

pe parcursul existenței sale.

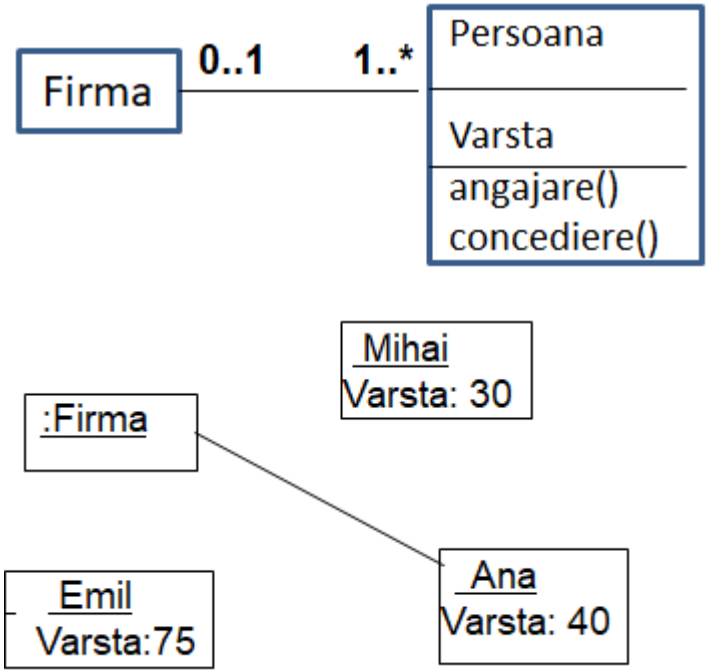


Stari si tranzitii (2)

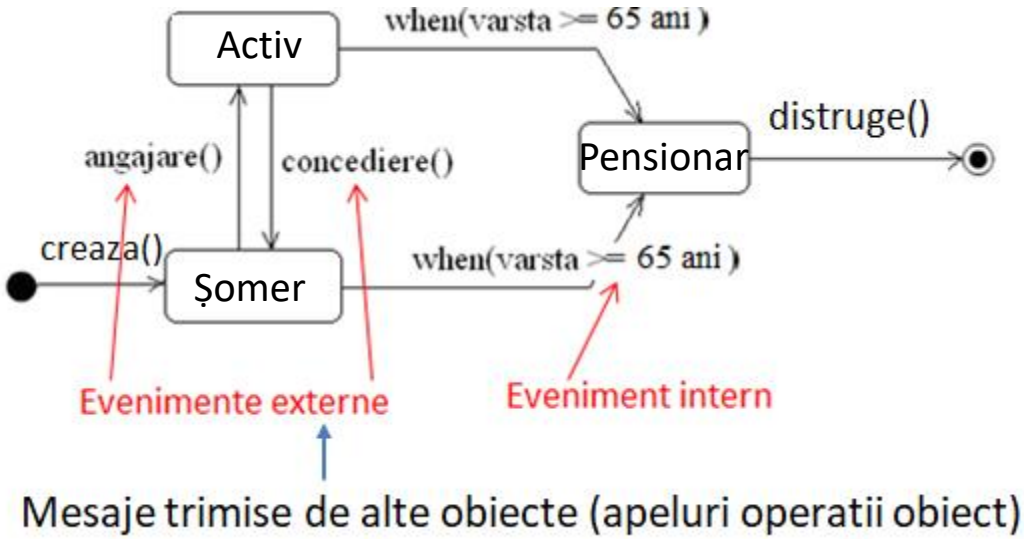
- Starea curenta a unui obiect este determinata de valorile atributelor sale, inclusiv de legaturile existente intre obiect si alte obiecte.

De exemplu, starea curenta a unei persoane poate fi: în activitate, în somaj sau la pensie.

Este determinata de varsta persoanei si de prezența unei legaturi catre o firma:



Modelarea vieții unui obiect Persoana



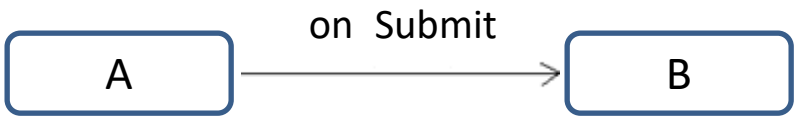
Evenimente (1)

Un eveniment poate fi:

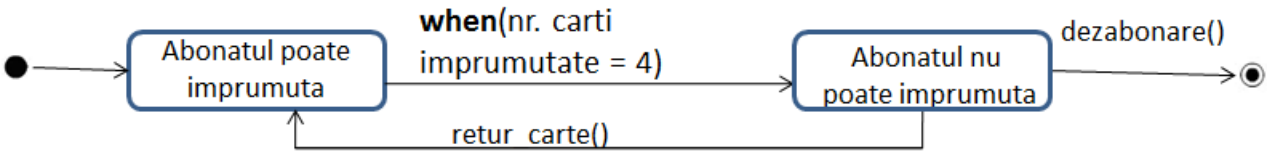
- Receptionarea unui semnal, cum ar fi o exceptie, o notificare, un eveniment generat de interactiunea cu utilizatorul.
- Receptionarea unui apel, adica invocarea unei operatii a clasei obiectului.
- Recunoasterea unei conditii in mediul extern sau in obiectul insusi:
  - conditie predefinita, care este indeplinita la un moment dat: "*change event*"
    - ***when****(expresie booleana).*
  - trecerea unei perioade de timp desemnate: "*elapsed-time event*"
    - ***after****(expresie al carei rezultat este o perioada de timp).*

- Un eveniment se reprezinta printr-o eticheta amplasata deasupra tranzitiei pe care o declanseaza.

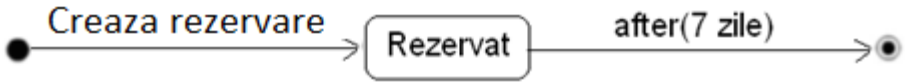
Evenimente (2)



Eveniment generat de interactiunea cu utilizatorul



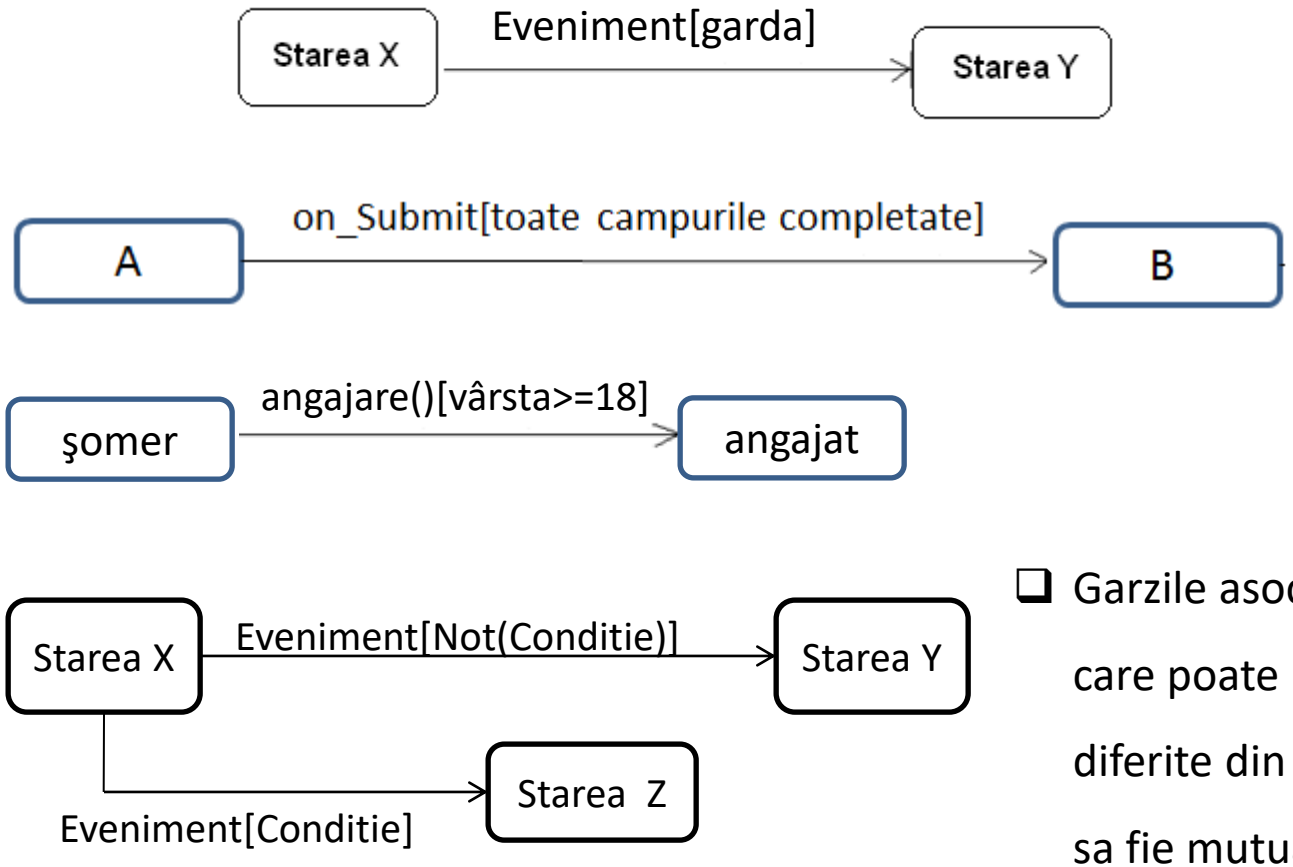
"change event"



"elapsed-time event"

Tranzitii conditionate

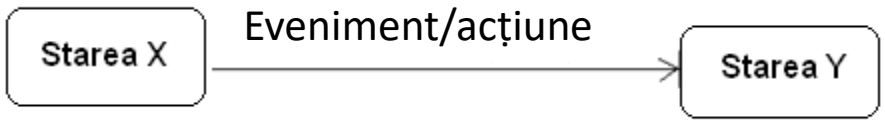
- Tranzitiile pot fi controlate prin garzi. O garda este o expresie booleana care valideaza declansarea unei tranzitii in cazul aparitiei unui eveniment.



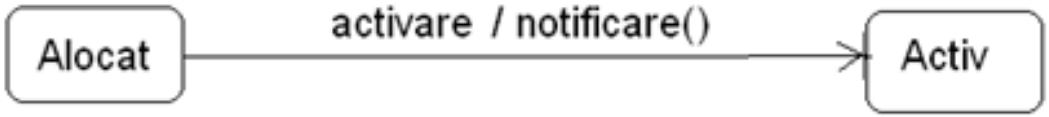
- Garzile asociate unui eveniment care poate produce tranzitii diferite din aceeasi stare trebuie sa fie mutual exclusive.

Actiuni si activitati(1)

- Operatiile unei clase apar in diagramele de stari ca **actiuni si activitati.**
- O actiune este considerata ca instantanee, adica are un timp de executie neglijabil in raport cu dinamica sistemului, şi nu poate fi întrerupta de un eveniment.
- In diagramele de stari, actiunile sunt atasate tranzitiilor:

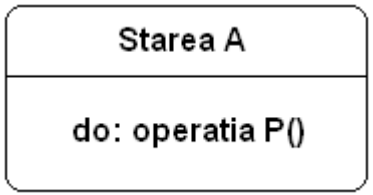


- O actiune defineste modul in care obiectul care receptioneaza evenimentul trebuie sa raspunda la eveniment. Timpul de executie al unei actiuni este nesemnificativ.
- Exemple de actiuni: apelul unei operatii a clasei, crearea sau distrugerea unui alt obiect, trimiterea unui semnal catre un alt obiect.



Actiuni si activitati(2)

- O activitate este o operatie care necesita un anumit timp de executie. Este asociata unei stari.
- Anumite activitati sunt ciclice, ca afisarea unei imagini pe ecranul unui monitor sau ca soneria telefonului care persista pana cand un eveniment o intrerupe declansand o tranzitie.
- Alte activitati sunt secventiale, cu o durata finita, ca de exemplu executia unui calcul.
- Activitatile sunt indicate prin cuvantul cheie "do":



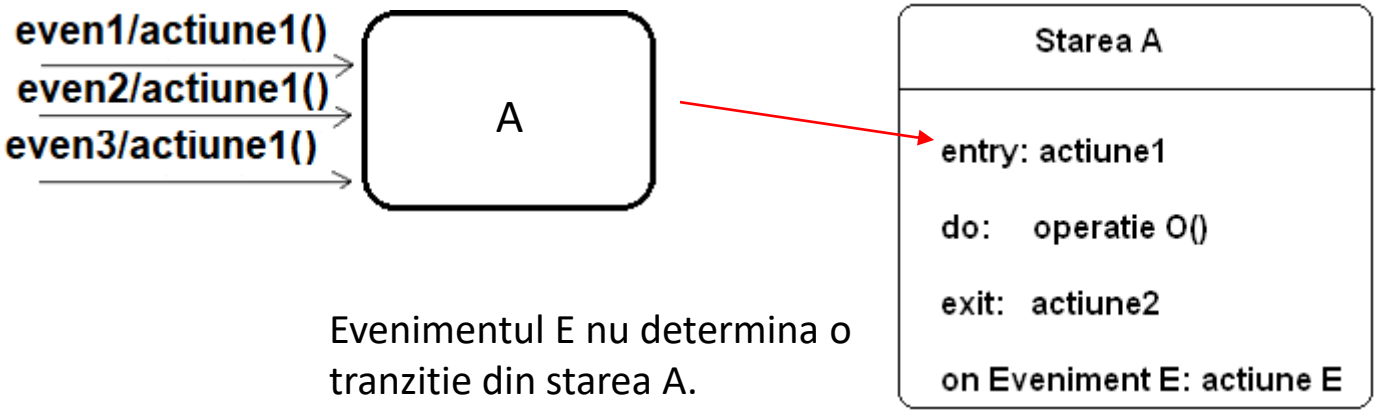
In starea A se execută operația P

- O activitate poate fi întreruptă oricand, atunci cand are loc un eveniment care determina o tranzitie de iesire din starea respectiva.

Actiuni si activitati(3)

- Mai multe evenimente pot determina tranzitia in aceeasi stare.
- Fiecare eveniment poate declansa o anumita actiune.
- Atunci cand toate evenimentele care conduc in aceeasi stare declanseaza aceeasi actiune, actiunea poate fi modelata ca ***actiune de intrare*** in starea respectiva.
- Iesirea dintr-o stare poate fi determinata de mai multe evenimente.
- Atunci cand toate evenimentele care declanseaza tranzitii din aceeasi stare specifica o aceeasi actiune, actiunea poate fi modelata ca o ***actiune de iesire*** din starea respectiva.

Actiuni de intrare/iesire si actiuni interne.

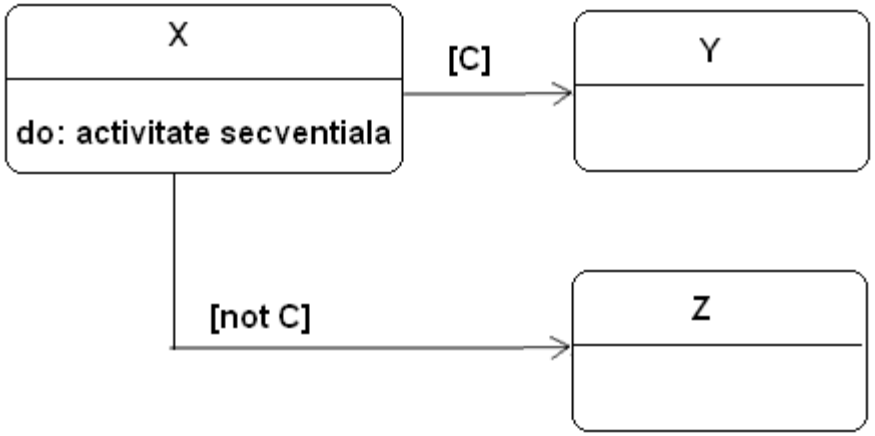


Evenimentul E nu determina o tranzitie din starea A.

Tranzitii automate

- La terminarea unei activitati secventiale are loc o tranzitie automata din starea în care s-a executat.
- O tranzitie care nu este marcata printr-un eveniment este numita tranzitie automata.

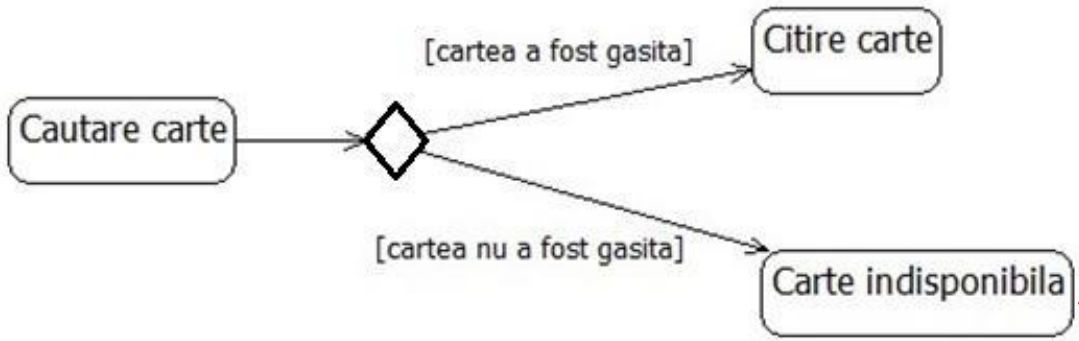
- Tranzitiile automate pot fi controlate prin garzi:



Pseudostări (1)

Alegerea dinamică (Choice)

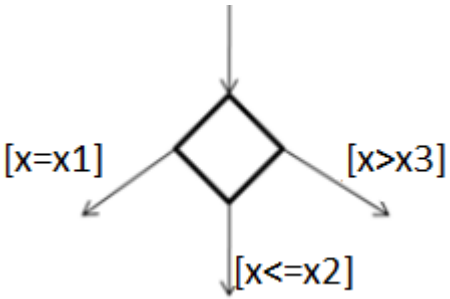
❖ Pseudostare în care se efectuează o ramificare condiţionată dinamică.



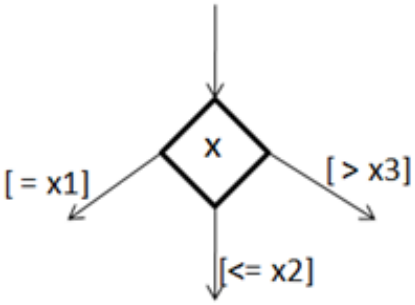
Stare în care utilizatorul poate primi un mesaj sau i se dă opțiuni de continuare, ca de ex. căutarea unei alte cărți.

- Se evaluează gărzile asociate tranzițiilor de ieșire din pseudostare și se execută tranziția a cărei gardă este adevărată.
- Cel putin una dintre gărzi trebuie sa fie adevărată.
- Dacă mai mult de o gardă este adevărată, se alege pentru execuție, în mod arbitrar, una dintre tranziții.

Pseudostări (2)



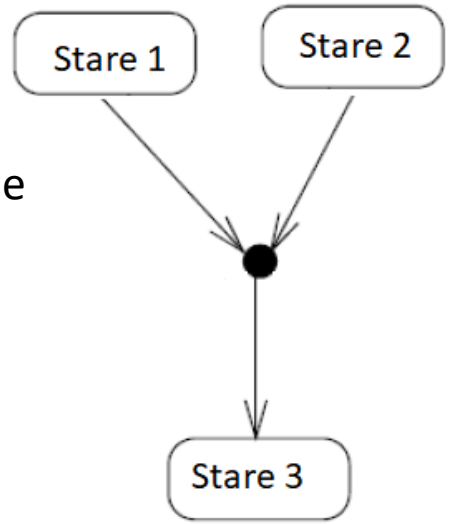
Reprezentare simplificată



Toate garzile sunt expresii binare cu acelaşi operand în stânga.

Joncţiune (Junction)

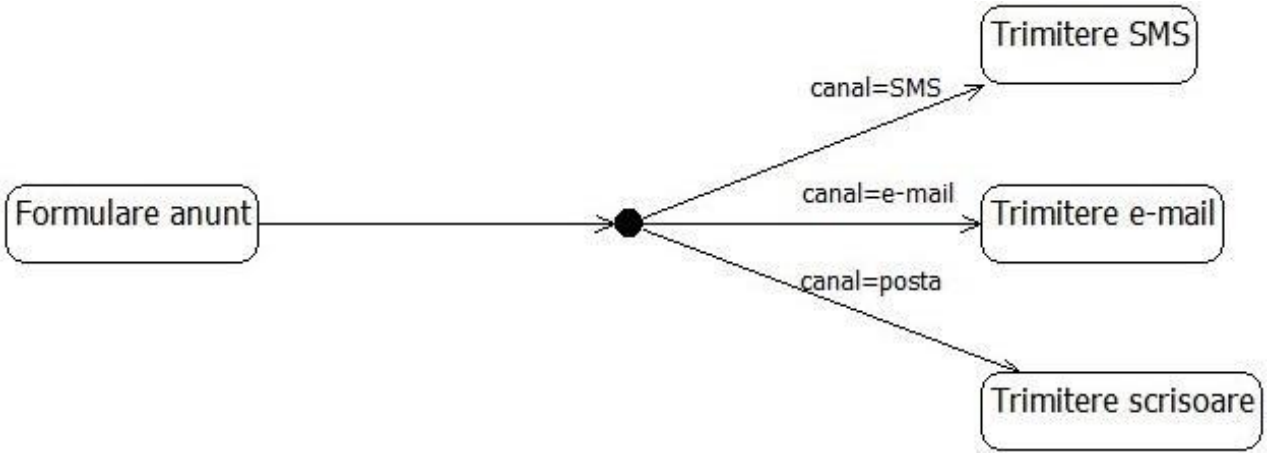
- Se poate folosi pentru a combina mai multe tranzitii de intrare într-o singură tranziție de iesire.



Pseudostări (3)

Joncţiune –ramificare conditionată statică

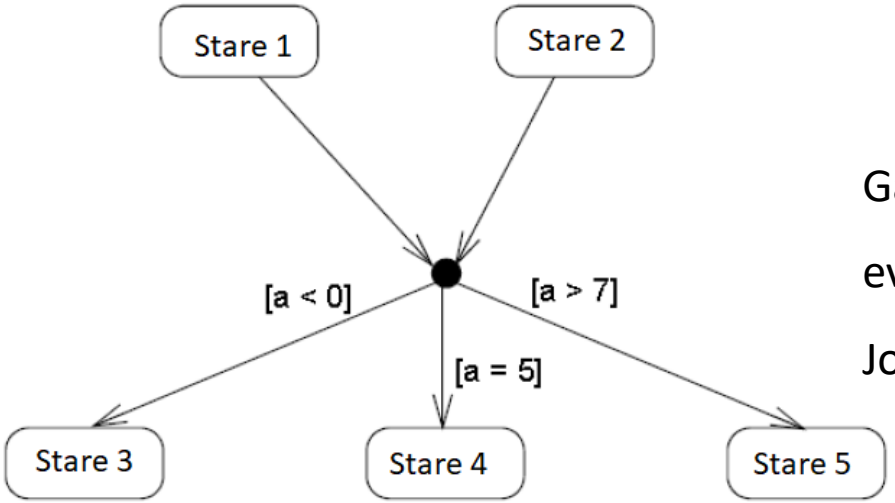
- Pseudostarea se poate folosi pentru a ramifica o tranzitie de intrare în mai multe tranzitii de iesire cu gărzi (constrangeri) asociate diferite.



- Garzile asociate tranzitiilor de iesire sunt evaluate inainte de intrarea in pseudostarea Jonctiune: pseudostarea mai este numita “ramificare conditionata statica”

Pseudostări (4)

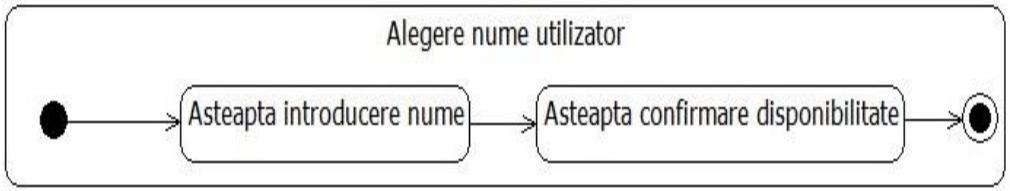
Joncţiune – continuare



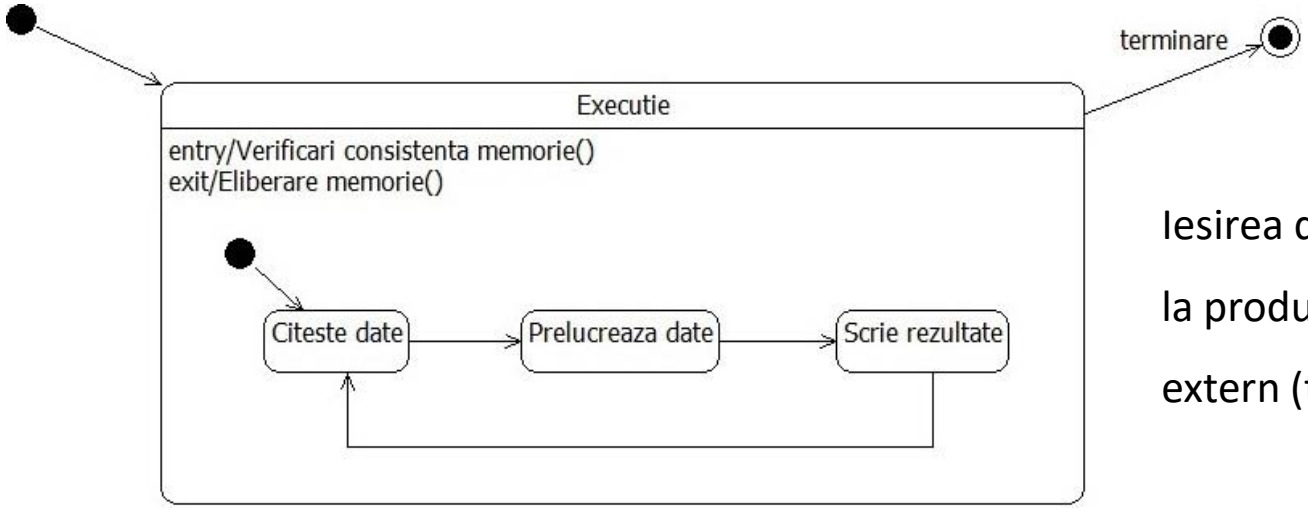
Garzile asociate tranzitiilor de iesire sunt evaluate inainte de intrarea in pseudostarea Jonctiune

Stari compuse (1)

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

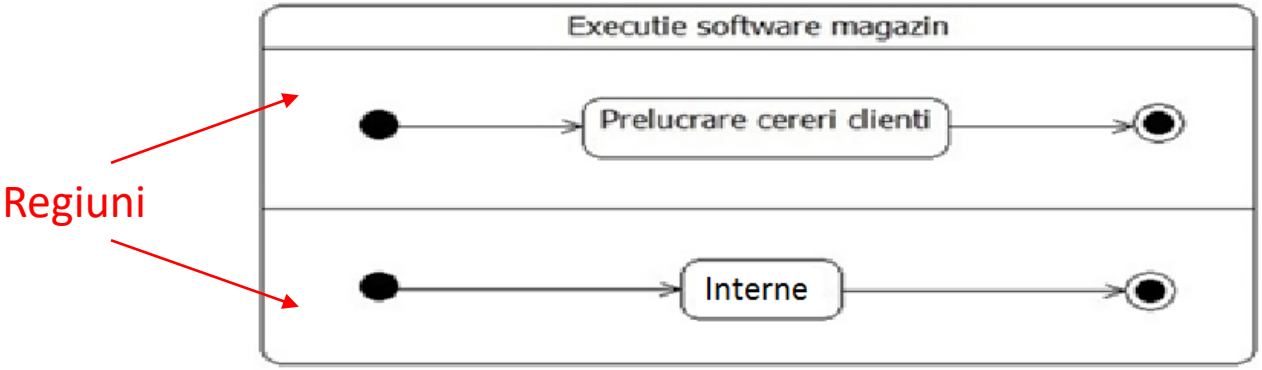


Stare compusa cu 2 substari secventiale



Iesirea din starea Executie are loc la producerea unui eveniment extern (terminare)

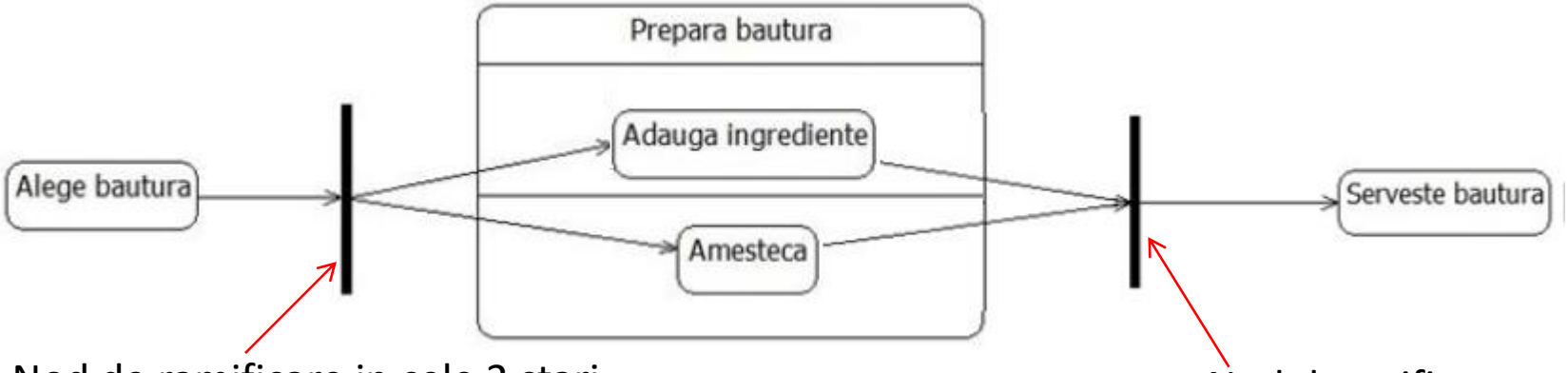
Stari compuse (2)



Stare compusa cu 2 substari concurente(ortogonale).

- Fiecare regiune contine un set de stari si tranzitii care se excuta secvential.
- Regiunile unei stări compuse ortogonale se execută în paralel.
- Regiunile descriu stari si tranzitii ale unor fire de executie paralele.

Stari compuse (3)



Nod de ramificare in cele 2 stari ortogonale.

Nod de unificare

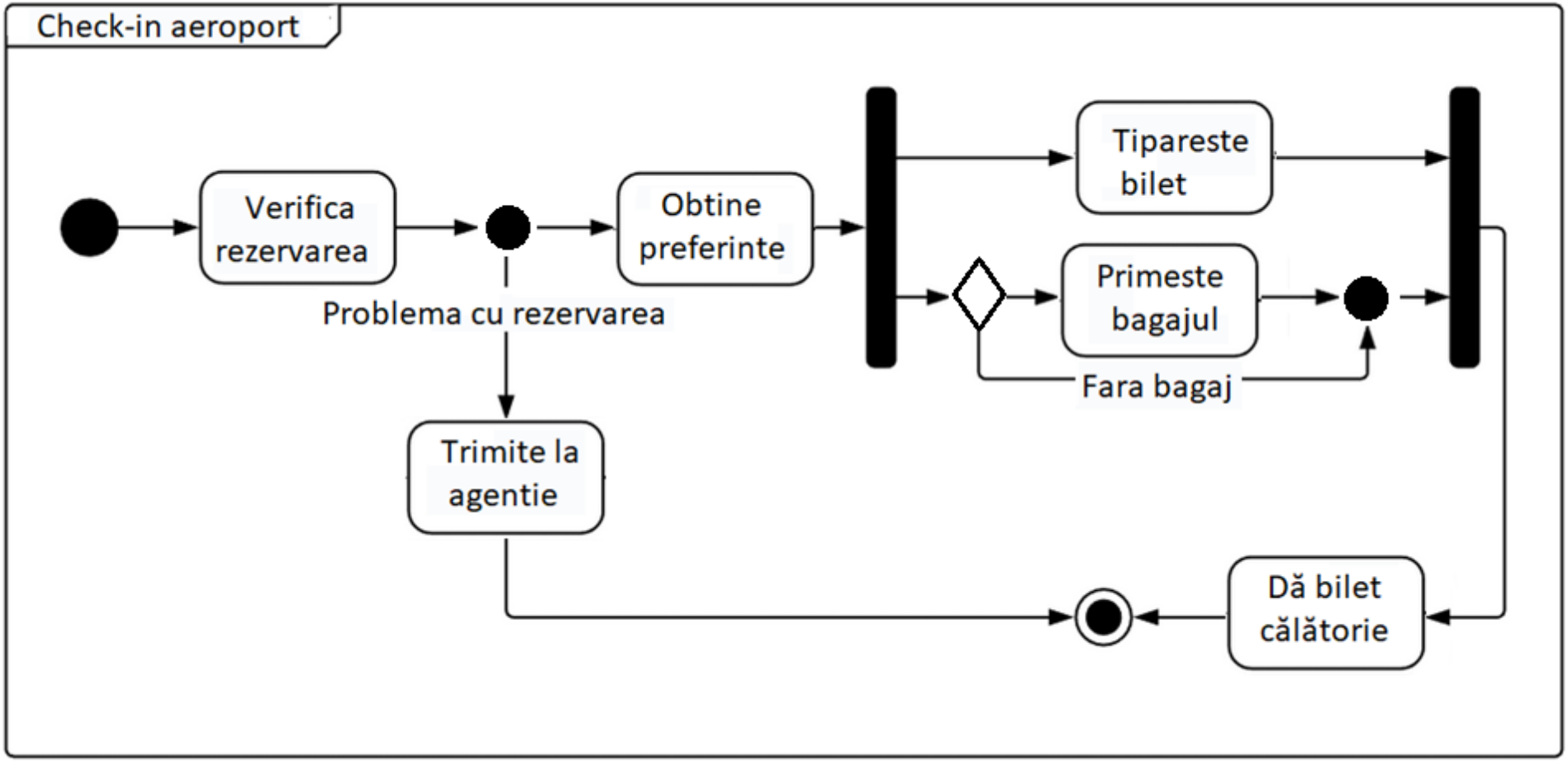
Ramificarea (Fork)

- ❖ Pseudostare care descompune tranziția de intrare în două sau mai multe tranziții de ieșire în stări ortogonale ale unei stări compuse.
- ▪ Tranzițiile de ieșire dintr-o pseudostare de tip *ramificare* nu pot avea asociate gărzi sau evenimente.

Unificarea (Join)

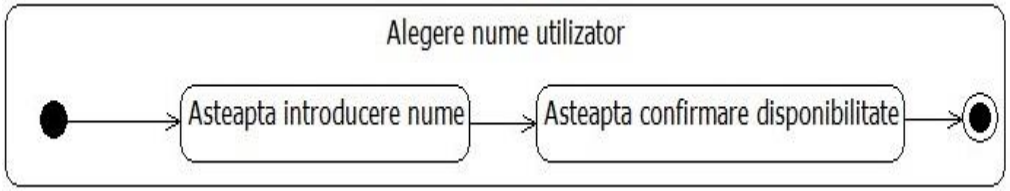
- ❖ Pseudostare care unifica tranzitii din stari ortogonale.
- ▪ Tranzitiile de intrare intr-o pseudostare Join nu pot avea asociate garzi sau evenimente.

Stari compuse si pseudostari - exemplu



Reprezentare ierarhica

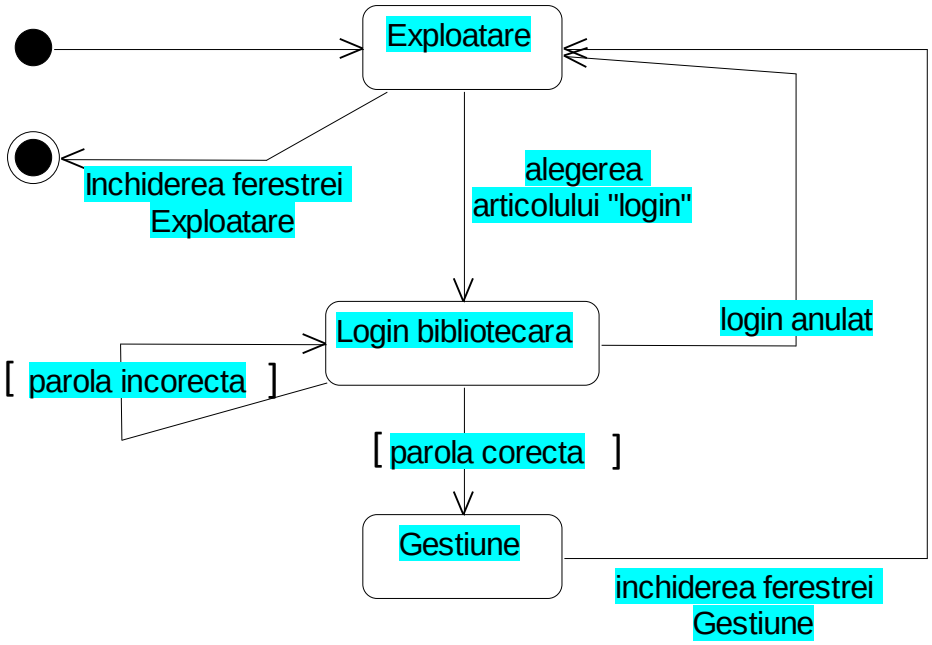
- Mai multe stari pot fi abstractizate intr-o singura stare, care corespunde unei reprezentari de nivel ierarhic mai inalt.
- O stare compusa corespunde unui nivel de abstractizare mai coborat.
- De exemplu, cele 2 reprezentari corespund la 2 niveluri de abstractizare diferite



- 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.

Concluzii

- Diagramele de stari se folosesc pentru modelarea comportamentala a **entităților reactive**→ pot fi identificate analizand diagramele de interactiune.
- Permit o documentare mai buna a claselor complexe.
- Pot fi folosite în alegerea cazurilor de test.
- Pot fi folosite si pentru redarea comportamentului in timp al unui sistem sau pentru modelarea comportamentala a interfetei utilizator. Exemplu:



Lecturi suplimentare

1. Alin Moldoveanu, Florica Moldoveanu, Maria-Iuliana Dascălu, Anca Ioniţă, Oana-Maria Ferche, Victor Asavei, Anca Morar, UML practic, Ed. MatrixRom, 2014

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

3. https://warren2lynch.medium.com/state-diagram-comprehensive-guide-with-examples-e08b6d1c70fe

4. https://sparxsystems.com/resources/tutorials/uml2/state-diagram.html

5. https://www.lucidchart.com/pages/uml-state-machine-diagram

Proiectarea arhitecturala -1 (Proiectarea de sistem)

Prof. univ. dr. ing. Florica Moldoveanu

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

Proiectarea arhitecturala

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

  - Definirea obiectivelor proiectarii

  - Definirea subsistemelor software ale viitorului sistem

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

De la modelul de analiza → la modelul de proiectare

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

Modelul de proiectare arhitecturala cuprinde:

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

Obiectivele proiectarii

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

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

Criterii de proiectare (1)

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

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

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

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

Criterii de proiectare (2)

- **Criterii de performanţă**

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

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

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

Cadere = functionare neconforma cu specificatia.

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

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

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

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

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

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

Criterii de proiectare (3)

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

— Extensibilitatea: usurinta de a adauga functionalitati sistemului

— Usurinta de modificare a sistemului

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

— Portabilitatea: usurinta de a porta sistemul pe diferite platforme

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

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

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

- **Criterii ale utilizatorului final**

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

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

Criterii de proiectare (4)

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

— Costul dezvoltarii: costul dezvoltarii sistemului initial

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

— Costul actualizarii (upgrade) sistemului: costul modificarii sistemului existent pentru a obtine sistemul actualizat, asigurandu-se compatibilitatea cu sistemul inlocuit (backward compatibility)

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

— Costul administrarii: costul necesar administrarii sistemului

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

Stabilirea si prioritizarea criteriilor de proiectare (1)

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

Performanță <–> cost dezvoltare

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

Stabilirea si prioritizarea criteriilor de proiectare (2)

Fiabilitatea <-> cost dezvoltare

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

Stabilirea si prioritizarea criteriilor de proiectare (3)

Cost dezvoltare <–> cost mentenanță

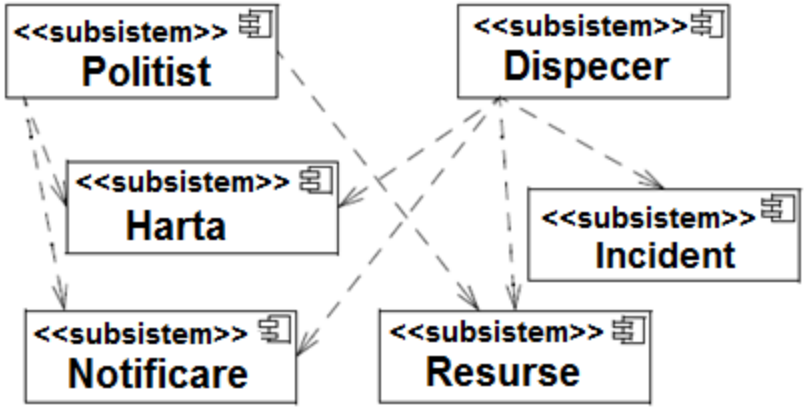
Obiectivul prioritar	Criteriul de proiectare primar	Criteriul de proiectare secundar
Dezvoltarea sistemului cu încadrarea cheltuielilor în costul de dezvoltare prevazut	**Costul de dezvoltare** Planificarea activitatilor de dezvoltare (proiectare, testare) cu un set de resurse umane, a.î. sa nu se depaseasca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

Proiectarea arhitecturii software (1)

- Arhitectura software este rezultatul unui **proces iterativ** de **descompunere a cerintelor functionale si alocarea lor unor subsisteme ale viitorului sistem**.
- Scopul descompunerii: reducerea complexitatii (“divide et impera”).
- Un **subsistem:**
  - Contine un set de clase corelate, care realizeaza o functie/un set de functii.
  - Se doreste sa fie o parte substituibila a sistemului, cu interfețe bine definite, care încapsulează comportamentul claselor pe care le conţine.
  - Trebuie sa fie descris a.î. sa poata fi dezvoltat de o singură persoană sau de o echipă de dezvoltare, în mod independent.
- Descompunand sistemul în subsisteme relativ independente:
  - Echipe independente le pot dezvolta cu un minimum de comunicare
  - Mentenanta sistemului este mult uşurata.
- In cazul sistemelor complexe, descompunerea se aplica în mod recursiv, descompunând subsistemele în subsisteme mai simple.

Proiectarea arhitecturii software (2)

Reprezentarea arhitecturii software printr-o diagrama de componente UML

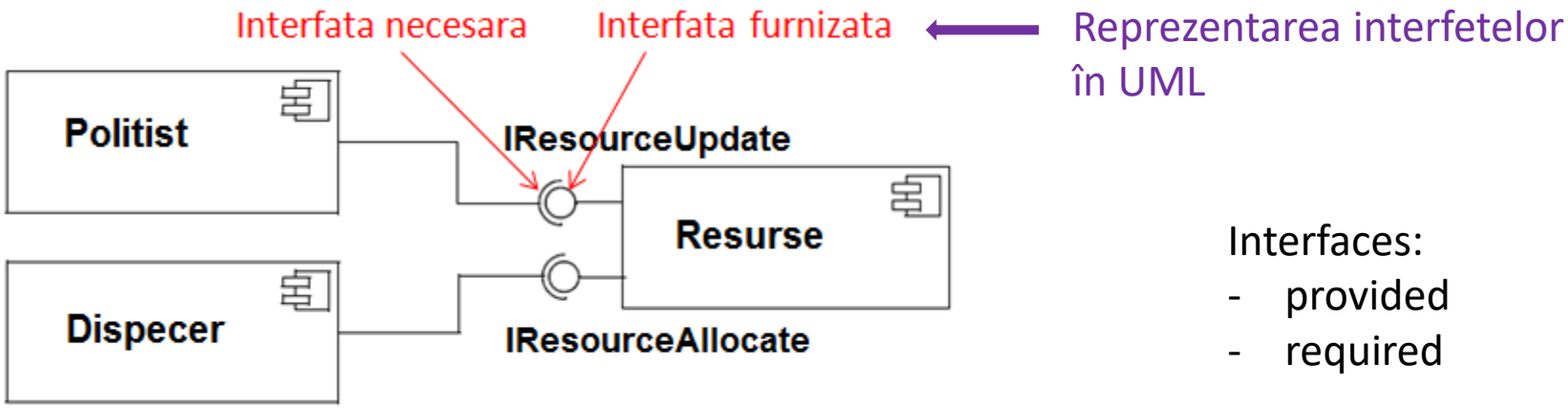


- Intre subsistemele rezultate din descompunerea cerintelor functionale se stabilesc relatii de dependenţă.
- Un subsistem A depinde de subsistemul B daca A foloseste operatii/servicii implementate de B.

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

Servicii şi interfeţe de subsisteme

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



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

Principii si strategii de descompunere în subsisteme

- **Subsistemele trebuie sa fie cat mai independente unul de altul**:
  - O modificare a unui subsistem trebuie sa aibă influenta minima asupra altor subsisteme.
  - O schimbare mica a cerintelor nu trebuie sa conduca la modificari majore ale arhitecturii software (sa afecteze cat mai putin relatiile dintre subsisteme).
  - Efectul unei conditii de cădere trebuie sa fie izolat în subsistemul care a generat-o.
  - Un subsistem trebuie sa poata fi înteles ca o entitate de sine-statatoare.
- **Subsistemele trebuie sa "ascunda" continutul lor** (*information hiding*): se separa interfaţa/interfetele de implementare.

❖ Strategii de descompunere in subsisteme:

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

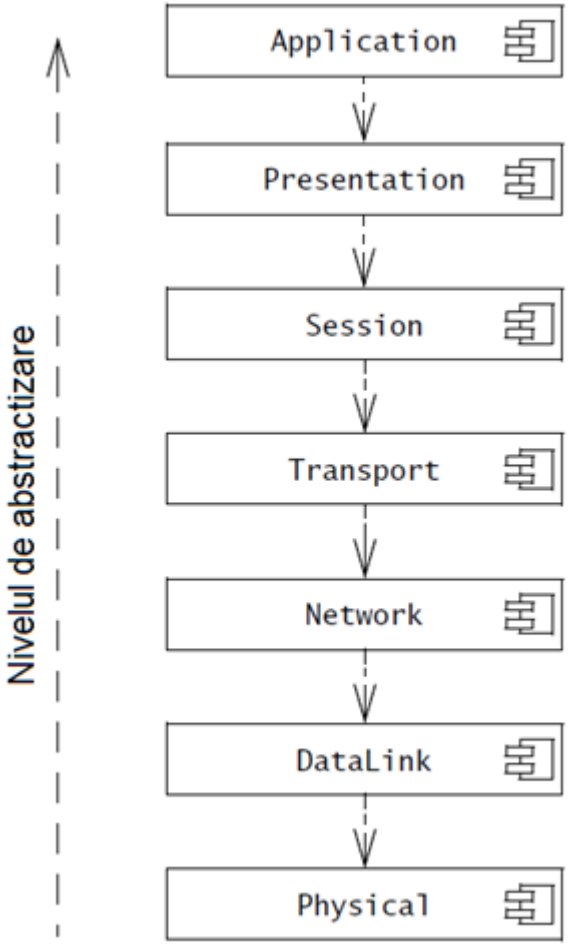
Descompunere ierarhică

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

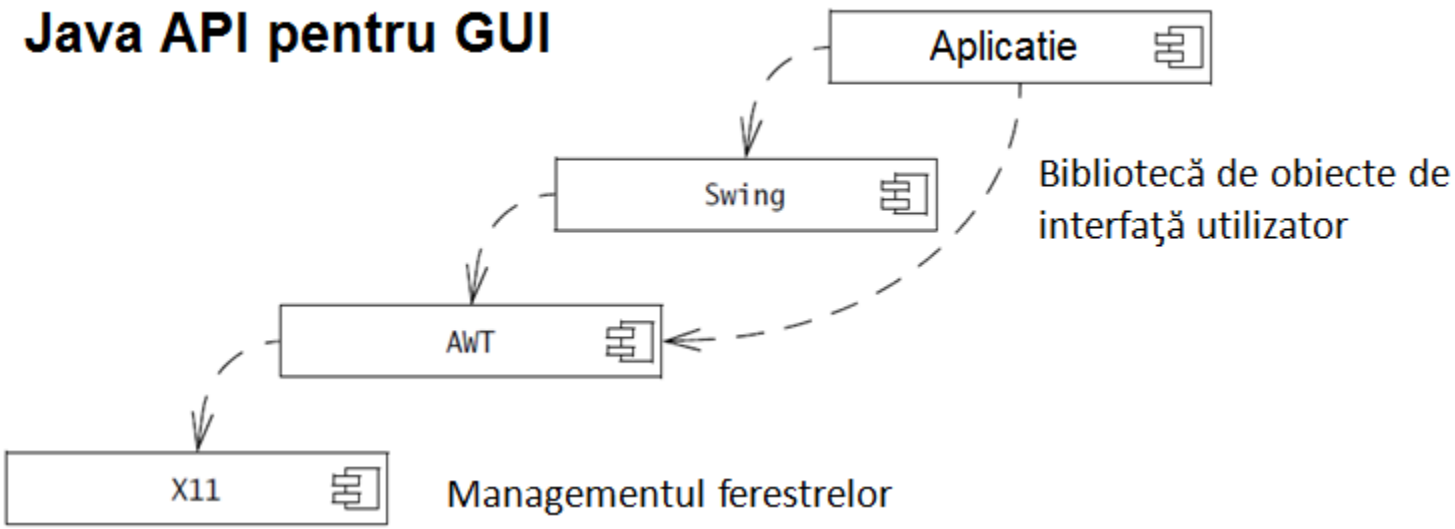
- **In mod uzual, descompunerea ierarhica are 3-5 niveluri.**

Arhitectură ierarhică închisă

Modelul OSI
(Open Systems Interconnection)



Arhitectură ierarhică deschisă



Avantajele unei arhitecturi ierarhice închise:

- cuplare slaba între subsisteme

- subsistemele pot fi integrate si testate incremental

Dezavantaje:

- fiecare nivel introduce un overhead de viteza si memorie → scaderea vitezei si cresterea necesarului de memorie.

- adaugarea de noi functionalitati pe un nivel intermediar este dificila, daca nu a fost prevazuta.

Partiţionare

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

Exemplu: sistemul de bord al unui autovehicul poate fi descompus in:

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

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

- **Mai intai** se face o **partitionare in subsisteme de nivel inalt**, care ofera functionalitati specifice sau se execută pe noduri hardware diferite.
- **Fiecare subsistem de nivel inalt este descompus ierarhic**, daca se justifica, in niveluri din ce in ce mai coborâte, până ce se obțin subsisteme suficient de simple pt a fi dezvoltate de o singura persoana/echipa, in mod independent.

Cuplare şi coeziune

- **Cuplarea a doua subsisteme** este data de numarul de dependenţe dintre ele. Daca sunt slab cuplate ele sunt relativ independente, modificarea unuia va avea impact redus asupra celuilalt
- **In proiectarea de sistem se doreste ca subsistemele sa fie slab cuplate**. Aceasta minimizează impactul pe care caderile la executie sau schimbarile viitoare într-un subsistem îl are asupra celorlalte subsisteme.
- **Coeziunea unui subsistem** este determinata de numarul de dependenţe dintre elementele sale.
- **O proprietate dorita a proiectarii de sistem este obtinerea de subsisteme cu coeziune interna mare**.
- Un subsistem cu o coeziune interna slaba poate fi descompus in alte subsisteme, mai coezive →poate conduce la cresterea cuplarii, caci creste numarul de interfeţe.
- **In general, trebuie facut un compromis între coeziune si cuplare**. O euristica buna este ca pe fiecare nivel de abstractizare sa nu fie mai mult de 7+/- 2 subsisteme. De asemenea, numarul de niveluri ierarhice sa fie de 7+/-2. In general 3 niveluri sunt suficiente.

Stiluri arhitecturale

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

Stiluri arhitecturale care vor fi tratate in cadrul cursului:

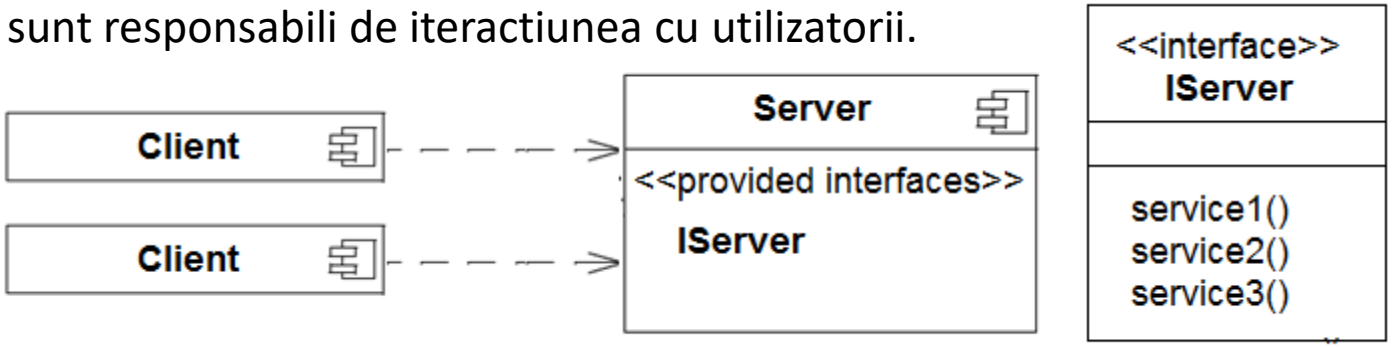
- Client - Server
- Peer - to – peer
- Arhitecturi ierarhice
    - Three - tier
    - Four - tier
- Arhitecturi centrate pe date
    - Repository
- Arhitecturi centrate pe fluxul datelor
    - Pipe and filter
- Arhitecturi orientate pe interactiune
    - Model - View - Controller (MVC)
- Arhitecturi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

Arhitectura *Client- Server* (1)

❖ Unul sau mai multe subsisteme, care alcatuiesc **server-ul,** furnizeaza servicii altor subsisteme, numite **clienti.**

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

❖ Clientii sunt responsabili de iteractiunea cu utilizatorii.



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

Arhitectura *Client- Server* (2)

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

Clientii:

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

Serverul:

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

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

Arhitectura *Client- Server* (3)

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

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

Arhitectura *Peer-to-peer*

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

Arhitecturi centrate pe date: *Repository*

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

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

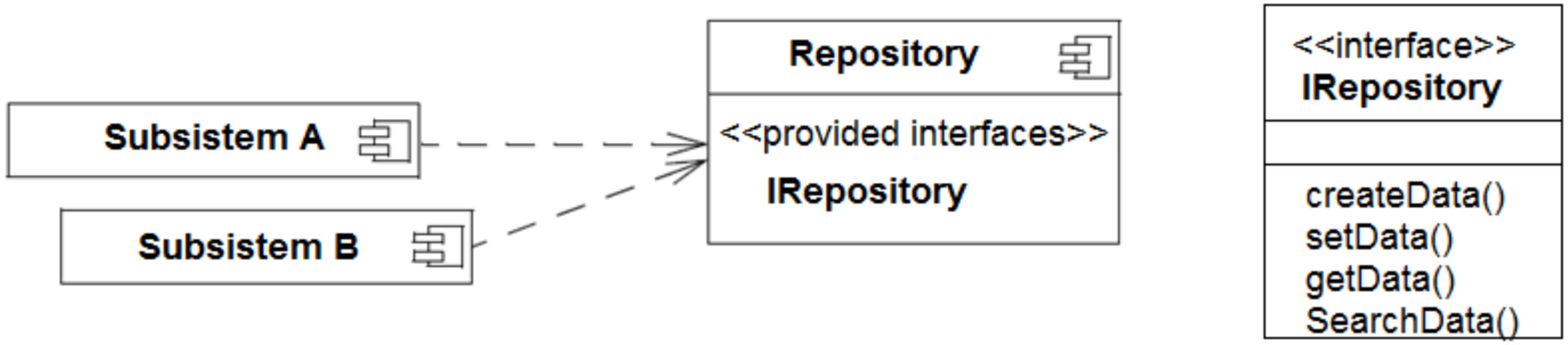
Arhitectura are 2 tipuri de subsisteme:

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

Fluxul controlului diferentiaza 2 tipuri de arhitecturi centrate pe date:

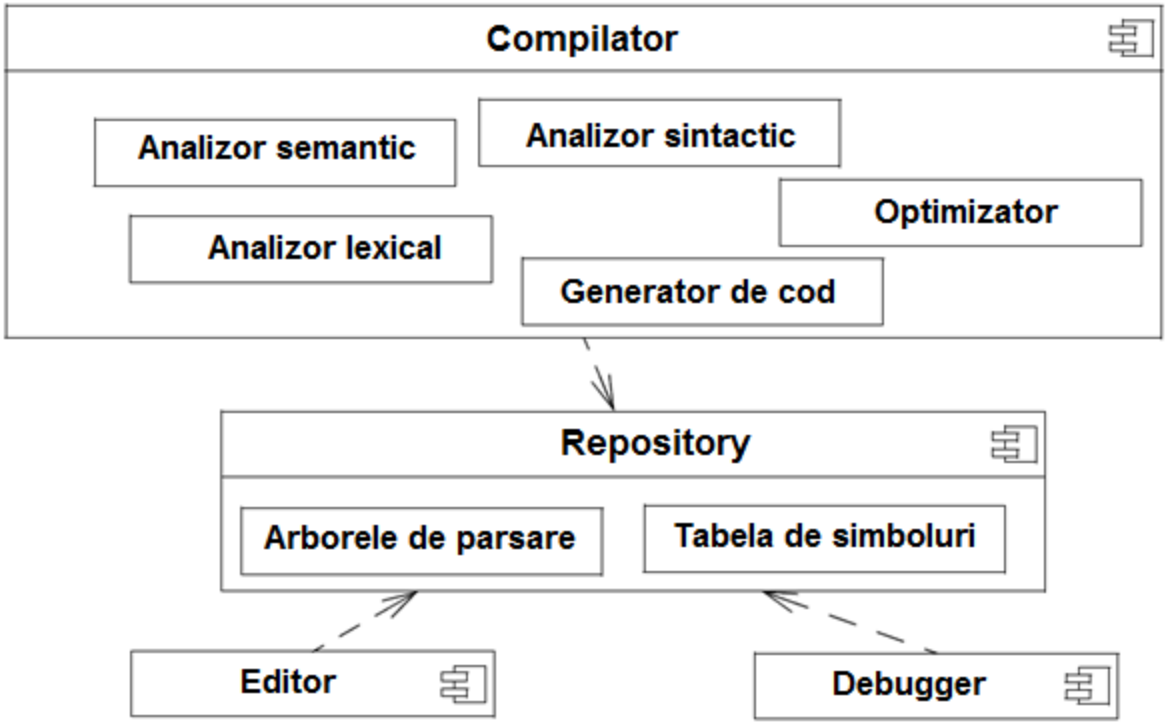
- Arhitecturi de tip “Repository pasiv”
- Arhitecturi de tip “Repository activ”.

Repository pasiv (1)



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

Repository pasiv (2)



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

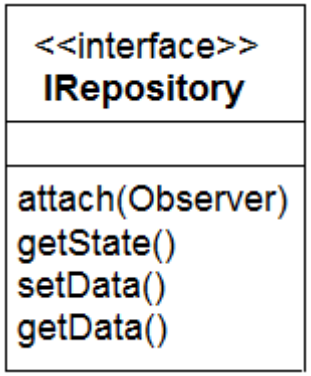
- Componenta Repository trebuie sa asigure serializarea accesurilor concurente.

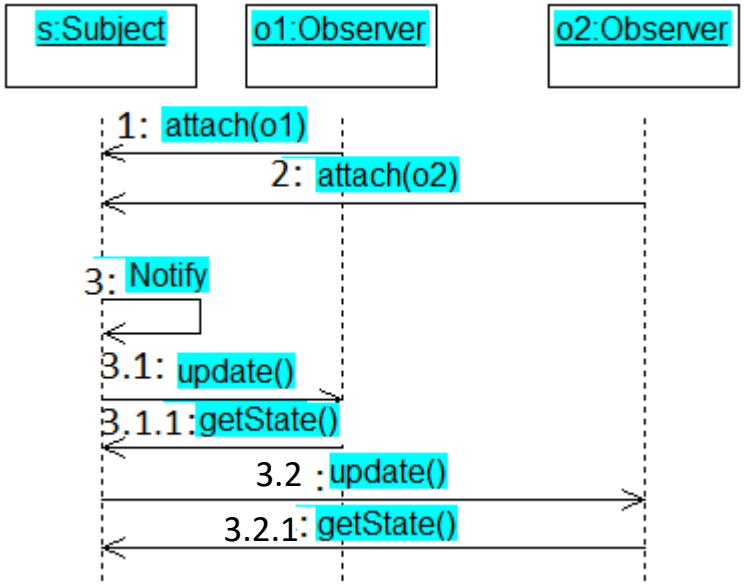
❖ **Avantajele arhitecturii**:
- Asigura integritatea datelor, operatii de backup si restaurare a datelor
- Asigura scalabilitate – pot exista oricâți accesori
- Reduce overhead-ul produs de transmisia datelor între componentele software

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

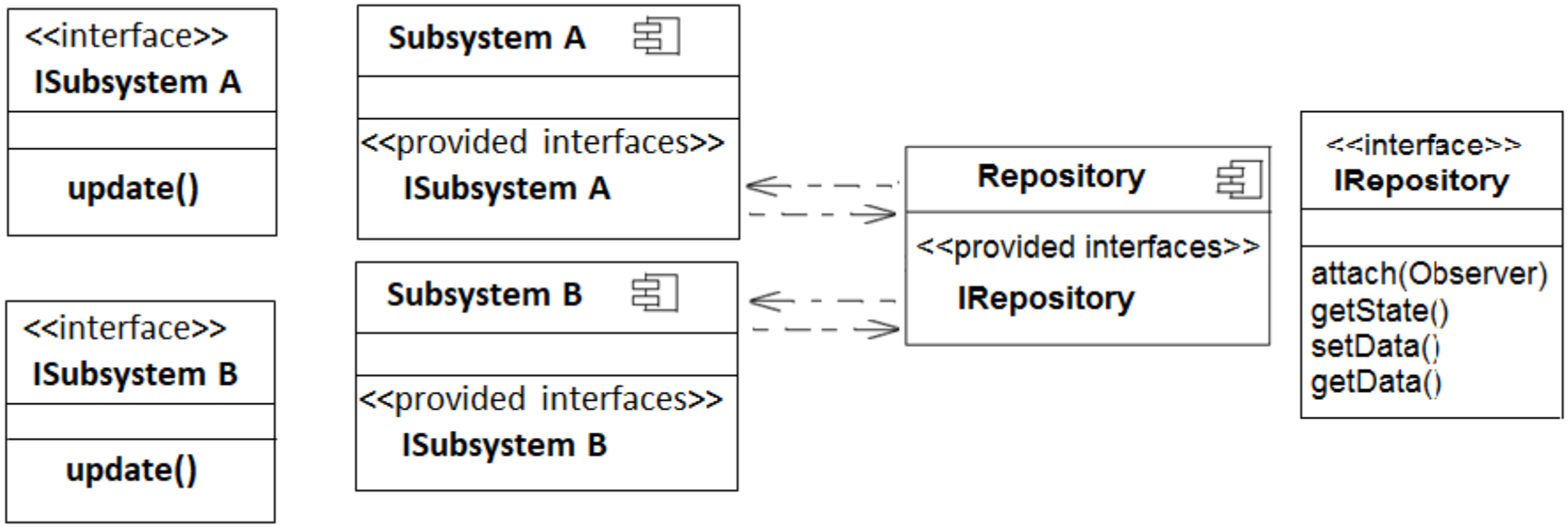
Repository activ (1)

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





Repository activ (2)



Repository activ (3)

Avantajele arhitecturii:

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

Dezavantaje:

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

Proiectarea arhitecturala -2

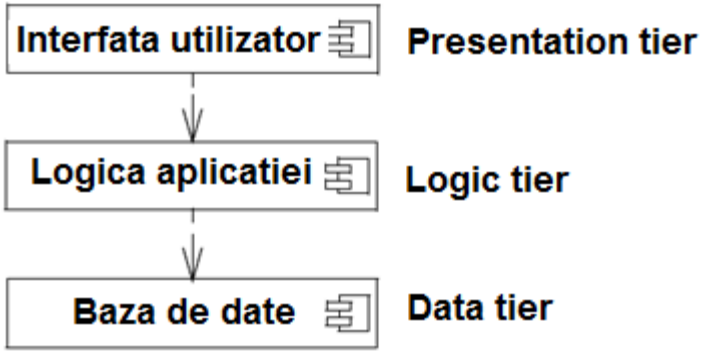
Prof. univ. dr. ing. Florica Moldoveanu

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

Arhitectura *Three-tier (1)*

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

- ***Nivelul interfaţă utilizator (Presentation tier)*** – *responsabil cu* 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 *stocare*
- ***Nivelul stocare (Storage/ Data tier)*** – sistemul de gestiune a bazei de date – asigura stocarea si regasirea obiectelor persistente.



Arhitectura *Three-tier (2)*

- **Cele 3 niveluri sunt de regula alocate pe noduri hardware distincte.**

- Arhitectura este des folosita în sistemele bazate pe Web:

  - Web Browers implementează nivelul Interfață utilizator

  - Web server – trateaza cererile provenite de la web browser: implementeaza nivelul aplicatie

  - Sistemul de gestiune al bazei de date – asigura gestiunea datelor persistente

- **Avantajele arhitecturii**:

- *Nivelul stocare*, analog unui subsistem *Repository*, **poate fi partajat de diferite aplicatii** care utilizeaza aceeasi baza de date.

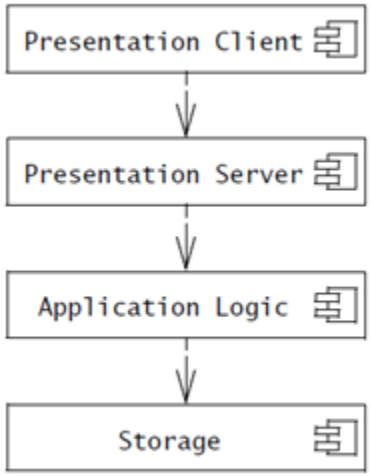
- Separarea nivelului *interfata* de nivelul *aplicatie* permite **existenta mai multor interfețe utilizator** pentru subsistemul/subsistemele care implementeaza logica aplicatiei.

- **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 variantă a arhitecturii Three-tier în care nivelul *Interfata utilizator* este descompus în:

- nivelul *Prezentare Client* (Presentation Client Layer) – operatii minimale de interactiune
- nivelul *Prezentare Server* (Presentation Server Layer) – implementeaza logica prezentarii



Pe calculatoarele utilizatorilor - Web client/browser

Pe unul sau mai multe servere - Presentation logic

▪ **Avantaje**:

- Cele 2 niveluri Prezentare pot fi implementate in limbaje de programare diferite si modificate independent.
- Nivelul *Presentation Client* poate oferi o gama larga de interfete utilizator (pentru diferite tipuri de dispozitive) care pot partaja elemente ale nivelului *Presentation server*
- Mai scalabila decat arhitectura în 3 niveluri.
- Mai usor de testat

Arhitectura *Four-tier (2)*

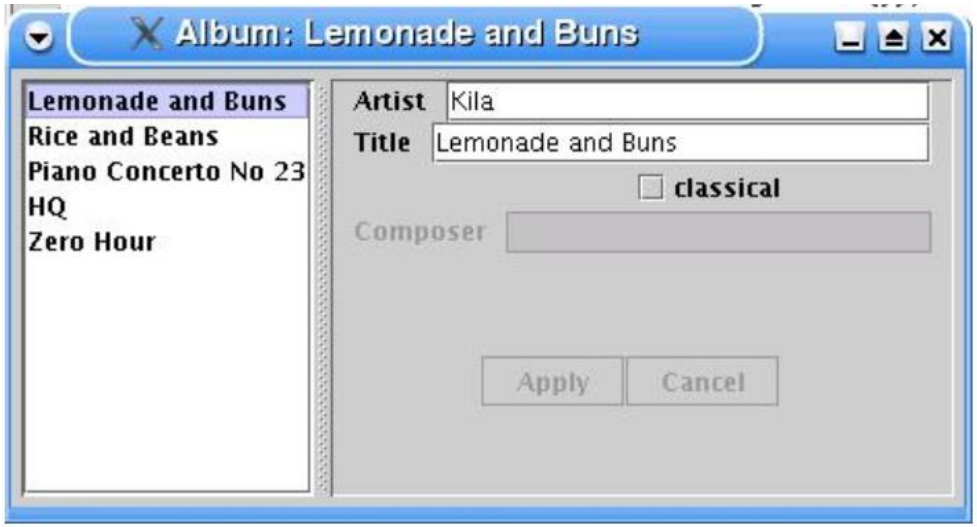
Logica prezentarii - exemplu:

• Alegerea din lista din stanga ferestrei determina datele care se vor afisa in partea dreapta a ferestrei şi titlul ferestrei.

• Câmpul Composer poate fi completat numai dupa ce s-a bifat check-boxul “classical”.

• Butoanele Apply si Cancel pot fi folosite numai dupa ce s-au completat toate campurile.

❖ Toate verificarile legate de afişările din partea dreaptă a ferestrei şi posibilitatile de interactiune pot fi efectuare de nivelul “Presentation logic”.



https://martinfowler.com/eaaDev/OrganizingPresentations.html

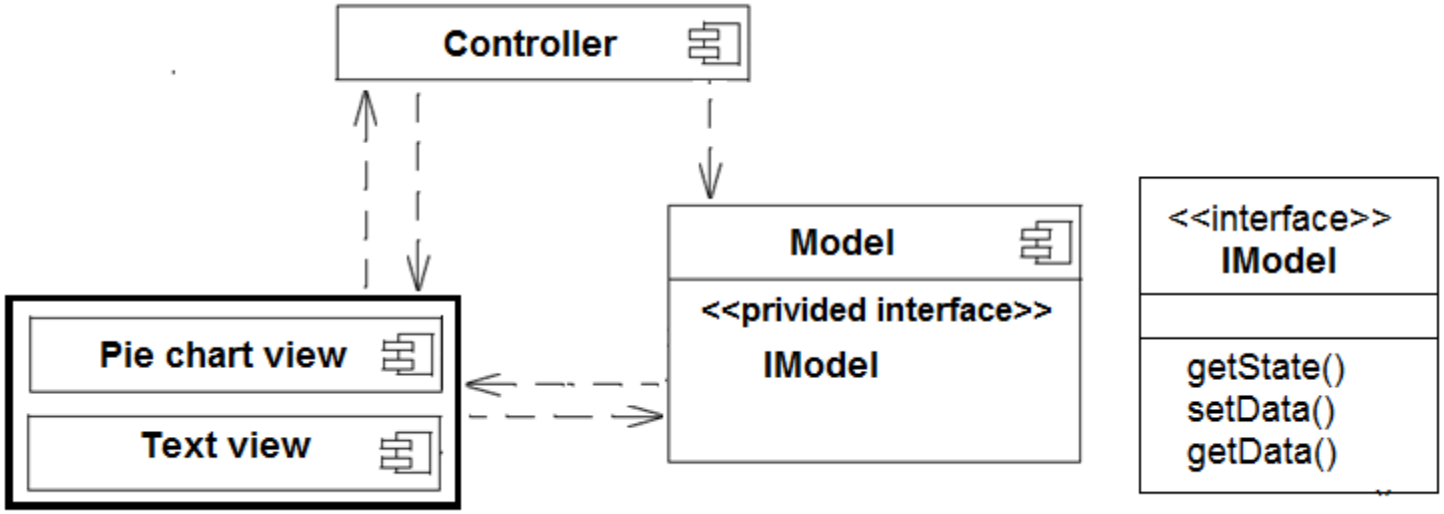
Arhitecturi orientate pe interactiune
Model-View-Controller (1)

❖ **Scopul**: separarea interfetei cu utilizatorul de datele şi logica aplicației.

Principalul stil arhitectual în această categorie: Model-View-Controller (MVC).

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

- Subsistemul **Model**: conține o reprezentare a datelor specifice aplicatiei, împreună cu:
  - Operațiile de acces la date
  - Operațiile de actualizare a datelor, conform interactiunii cu utilizatorul şi logicii aplicatiei



Model-View-Controller (2)

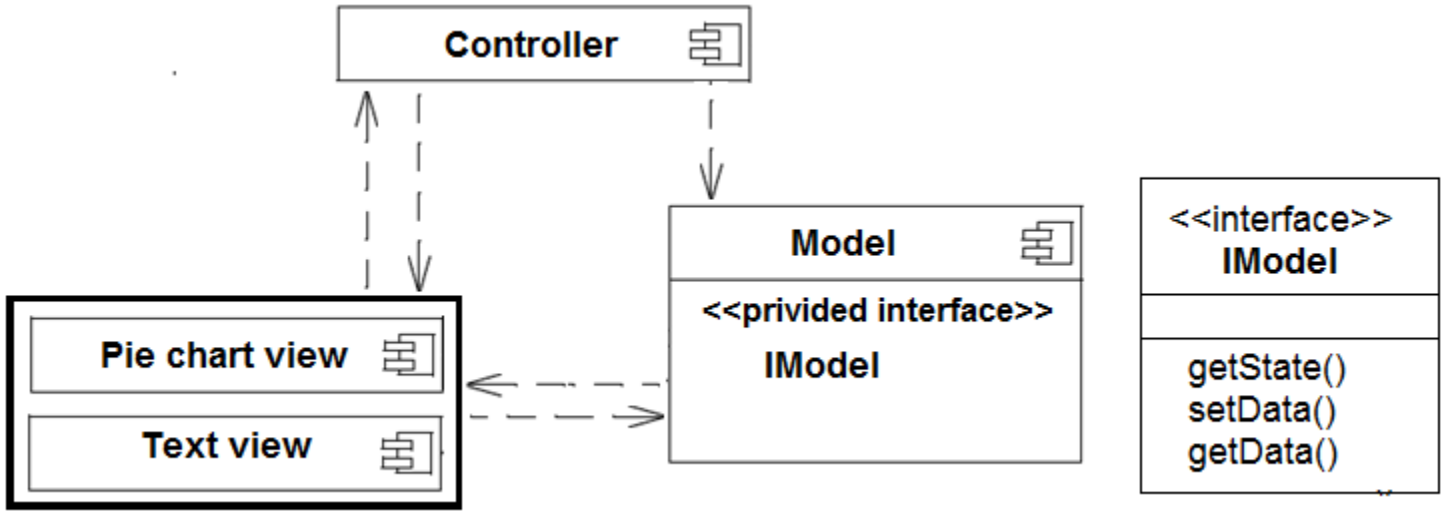
- Subsisteme **View**. Un subsistem View:

  - este asociat cu o fereastră ecran prin care utilizatorul interactioneaza cu sistemul

  - prezintă o vedere a modelului în fereastra asociată

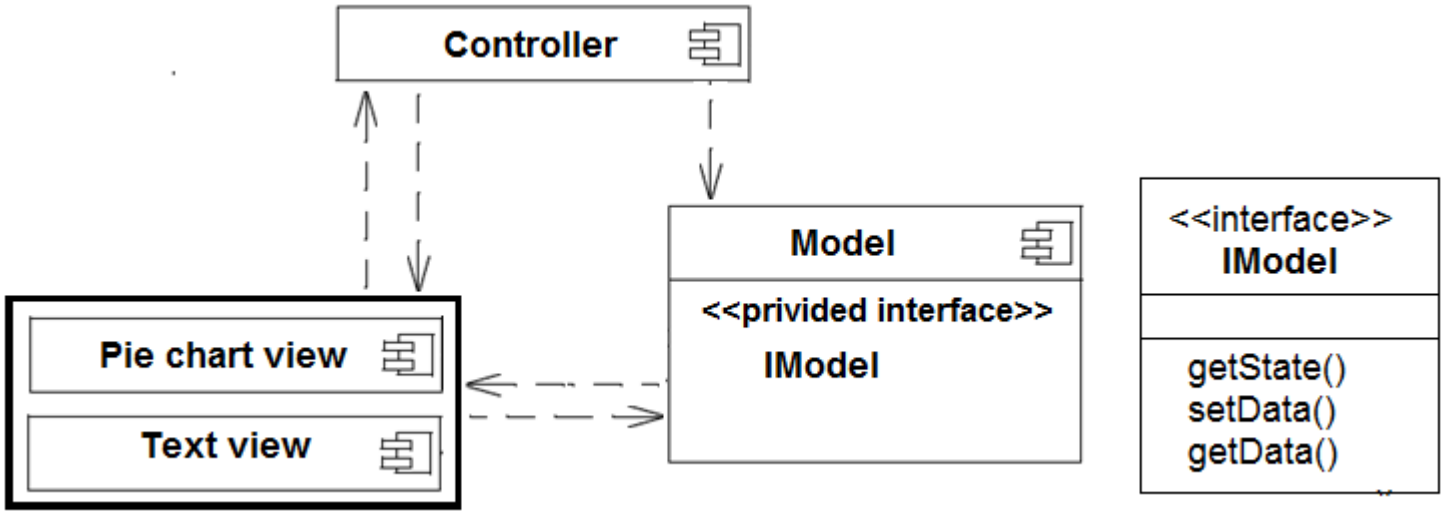
  - transmite **Controller**-ului mesaje eveniment generate de interactiunea cu utilizatorul

  - actualizează prezentarea datelor aplicației în fereastră, la orice modificare a lor



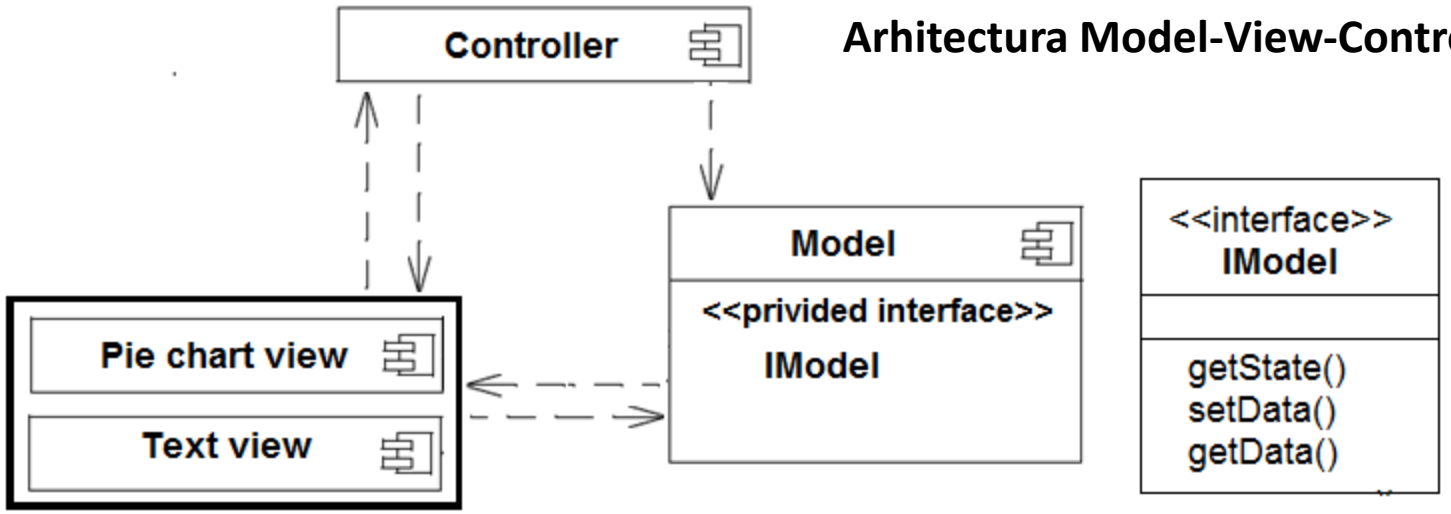
Model-View-Controller (3)

- Subsistemul **Controller**: procesează evenimentele generate de interacțiunea cu utilizatorul şi transmite mesaje la:
  - **Model**, pentru accesarea datelor sau modificarea datelor în conformitate cu interactiunea
  - **View**, pentru actualizarea vederii prezentate în fereastră.



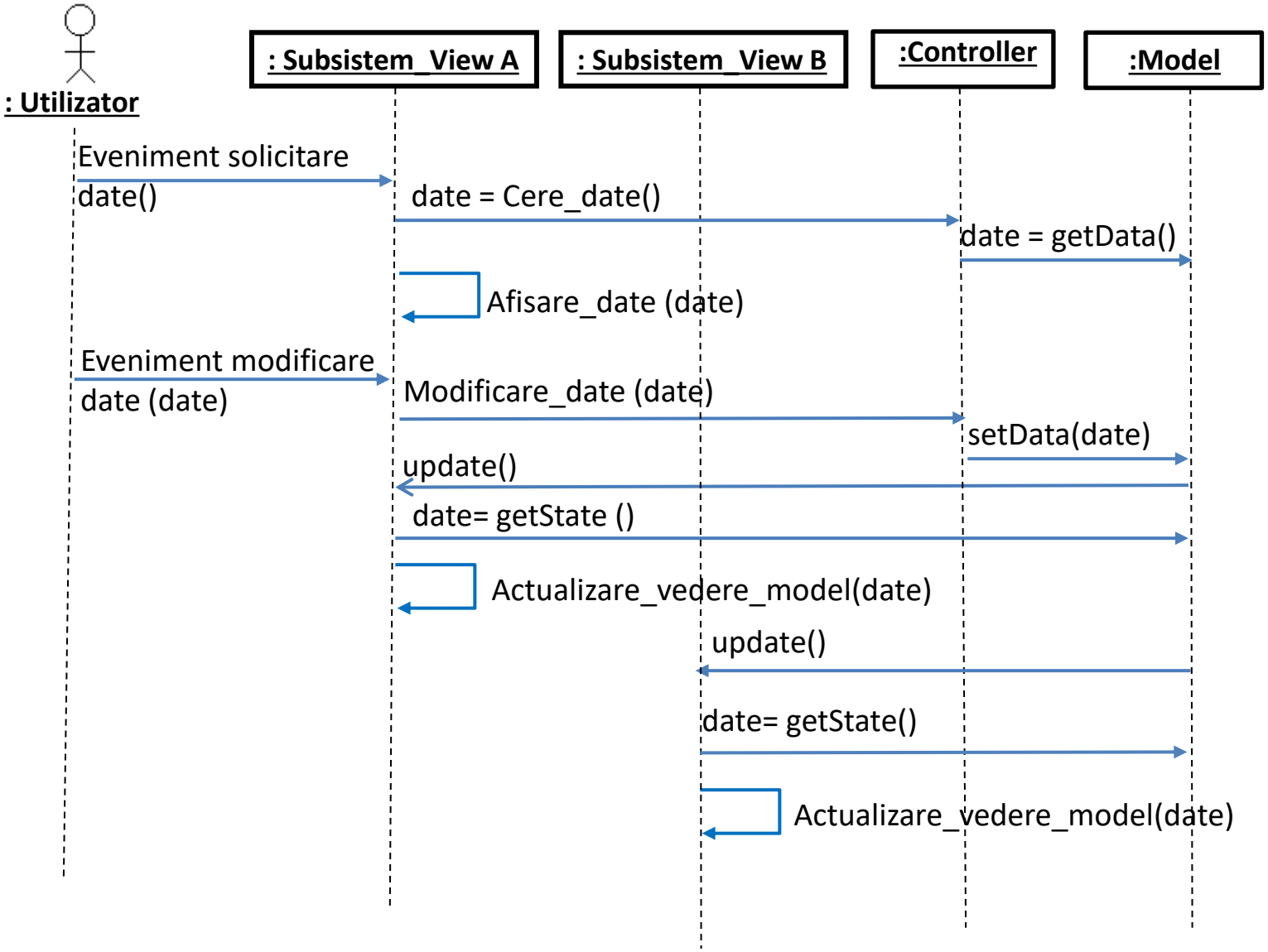
Model-View-Controller (4)

- MVC este un caz particular de *Repository activ*, unde subsistemul **Model** implementeaza depozitul (structura de date centrala) iar subsistemul **Controller** dictează fluxul controlului.



Arhitectura Model-View-Controller

Model-View-Controller (5)
- scenariu de utilizare -



Model-View-Controller (6)

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

❖ **Principalele avantaje ale arhitecturii:**

- ofera flexibilitate în adaptarea sistemului: interfeţele utilizator (View şi Controller) sunt mult mai supuse schimbarilor decat modelul (datele specifice aplicatiei).
- extensia simpla a sistemului pentru noi tipuri de clienti (componente View)
- uşurează mentenanța codului
- componentele sistemului pot fi dezvoltate şi testate separat
- adecvată aplicațiilor web

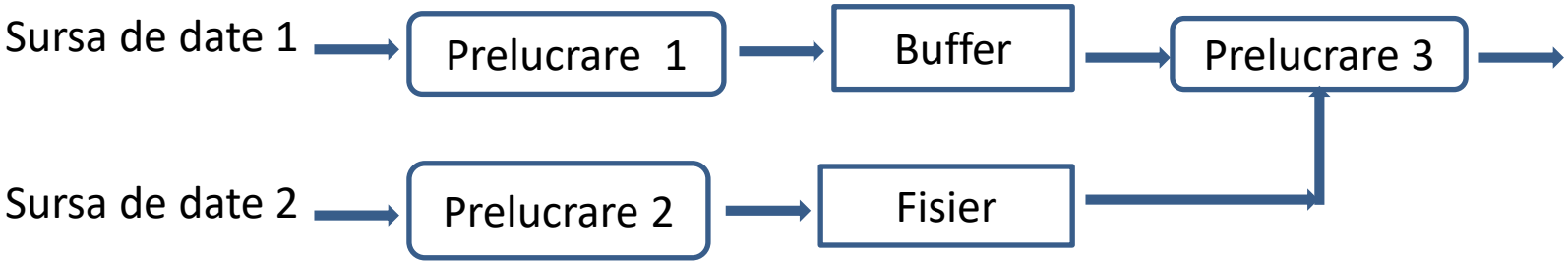
❖ **Dezavantaj**: poate conduce la cod suplimentar şi creşterea complexității aplicației atunci cand modelul de date şi interacțiunea cu utilizatorul sunt simple.

Arhitecturi centrate pe fluxul datelor: *Pipeline*

- **Fluxul global al controlului** este structurat în mai multe **etape de transformare a datelor** de intrare, pentru a produce anumite rezultate.
- **Fiecare etapă de transformare este o prelucrare simplă si este alocată unui subsistem**.
- **Comunicarea dintre subsistemele de transformare a datelor** se face prin **conectori** ca: buffer, coadă de mesaje, ş.a.
- Arhitecturi tipice bazate pe fluxul datelor:
  - Prelucrarea secvențială – pipeline secvential (classic)
  - Pipe and filter – pipeline ne-secvential

Pipeline secvential

- **Execuția unei etape de transformare a datelor se poate iniția numai după ce s-a terminat executia etapei precedente**: Prelucrare 3 se execută după Prelucrare1 **sau** după Prelucrare 2.
- Comunicarea între subsistemele de transformare se face prin bufere sau fişiere temporare.



❖ **Avantaje:**

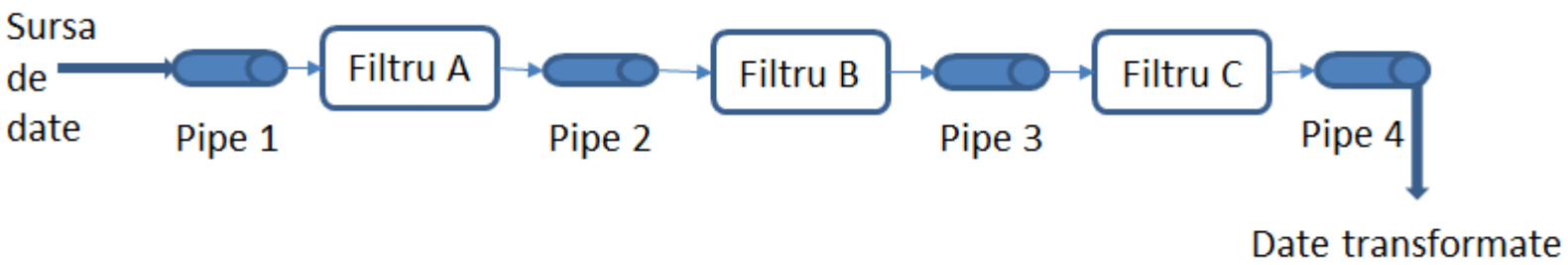
- Divizarea în subsisteme este simplă: fiecare subsistem poate fi un program independent care primeste date de intrare şi produce date de iesire.
- Subsistemele de transformare a datelor pot fi reutilizate.

❖ **Dezavantaje**

- Nu furnizează suport pentru procesarea paralelă şi interactivă.
- Latenţă în procesare şi throughput coborât.

Arhitectura *Pipe and filter* (1)

- **Scopul arhitecturii**: descompunerea unei prelucrări complexe în prelucrări simple separate, care transformă datele incremental şi pot fi executate în paralel.
- Arhitectura este alcatuită din:
  - O sursă de date
  - Mai multe etape de procesare, numite “filtre”, fiecare executnd o prelucrare simplă
  - Conectori, numiti “pipes”, prin care sunt transferate datele între filtre
    - Filtrele si conectorii formeaza un « pipeline »
  - Un receptor al datelor prelucrate în pipeline

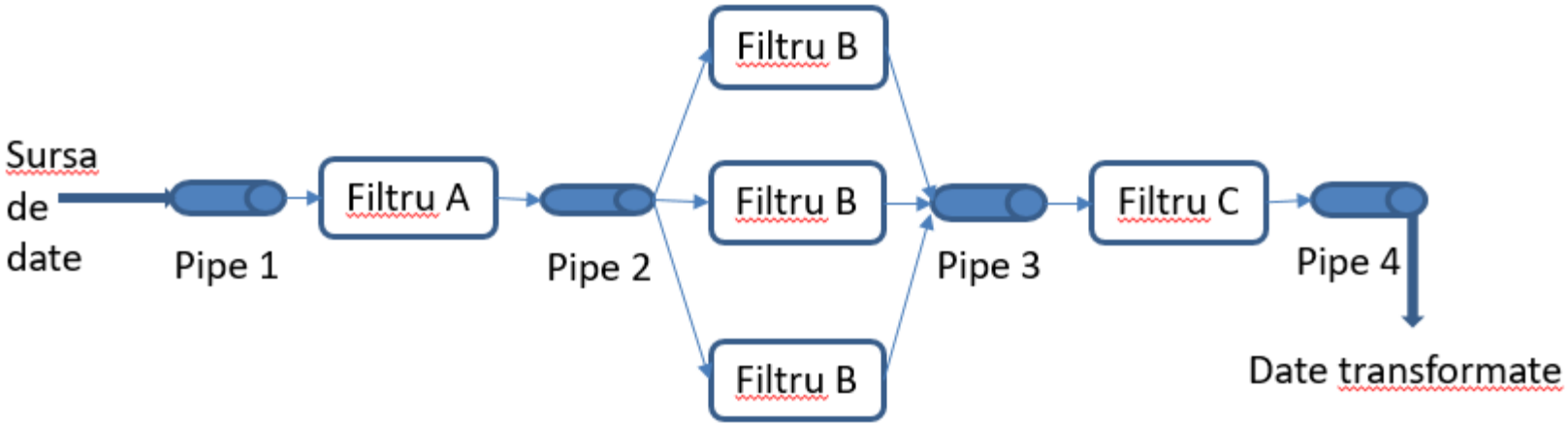


Arhitectura *Pipe and filter* (2)

- In general, intrările şi ieşirile filtrelor sunt structurate ca fluxuri de date (streams), iar conectorii sunt implementati ca bufere FIFO.
- **Avantajele structurarii datelor în fluxuri:**
  - Fiecare filtru îsi începe executia imediat ce s-au primit date în conectorul său de intrare şi se execută atâta timp cât exista date în conectorul respectiv.
  - Un filtru îşi poate începe executia chiar daca filtrul anterior în secventa de procesare, care produce fluxul său de intrare, nu şi-a terminat executia.
  - Filtrele din pipeline pot fi executate în paralel.
- Filtrele se pot executa pe resurse de calcul diferite: filtrele intensiv computationale pot rula pe hardware de înalta performanta, altele pe hardware obisnuit.
- Filtrele pot fi executate pe calculatoare aflate în locatii geografice diferite, în apropierea resurselor de care au nevoie.

Arhitectura *Pipe and filter* (3)

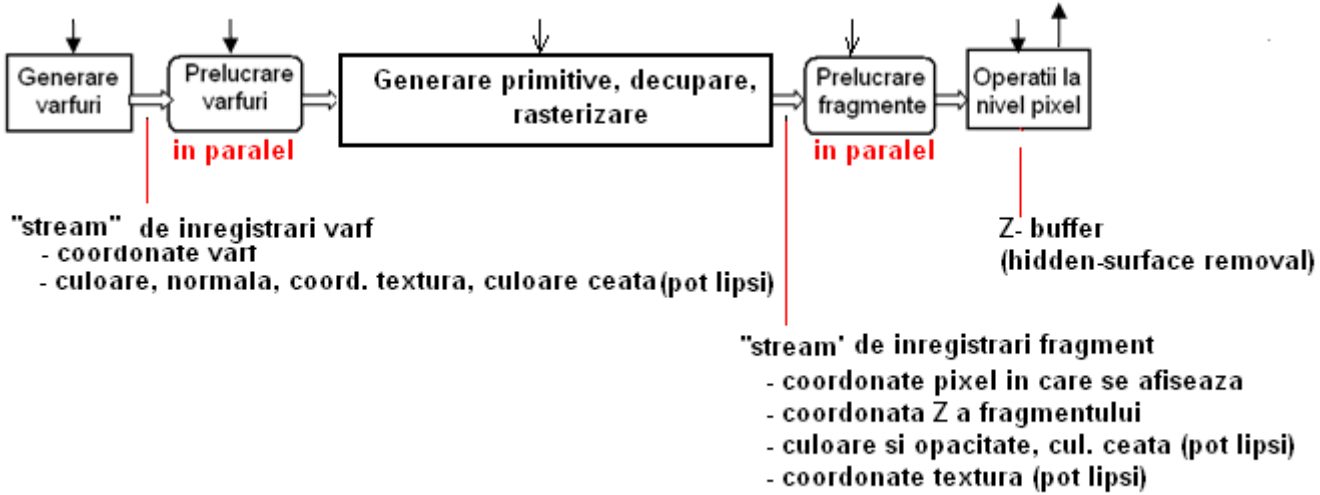
- Mai multe instanțe ale unui filtru pot fi executate în paralel pe aceeasi maşina sau pe maşini diferite.



- Cele trei instante ale filtrului B se executa în paralel pentru date diferite provenite din acelasi conector de intrare.
- Datele produse sunt memorate în 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 în 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** şi asigură o rată înaltă de execuție a tascurilor de prelucrare a unor volume mari de date.
- Permite **reutilizarea:** filtrele implementează prelucrări simple, care pot fi reutilizate în diferite aplicații.
- **Mentenanța** sistemului este mai simplă:
  - pot fi efectuate mai uşor modificări
  - filtrele sunt slab cuplate
- Oferă **flexibilitate în proiectarea sistemului**, fiind posibile atat executia secventială cat şi cea paralelă a filtrelor.
- **Toleranta la caderi**: dacă o instanță a unui filtru cade sau masina pe care se execută devine nedisponibilă, prelucrarea aflată în curs pe acea instanță poate fi alocată altei instanțe.

Arhitectura *Pipe and filter* (6)

❖ **Dezavantajele arhitecturii**

- Flexibilitatea crescută a arhitecturii poate introduce complexitate, mai ales atunci când filtrele dintr-un pipeline sunt distribuite pe diferite maşini.
- Este necesară o infrastructură care să asigure că datele transferate între filtre nu se pierd.
- Nu este adecvată interactiunilor.
- Poate introduce un overhead datorită transferului datelor între filtre.
- Pot sa apara probleme în cazul instanțelor de filtre executate în paralel, atunci cand una dintre instanțe cade: înainte de cădere a modificat o informație de stare sau deja a postat un mesaj pentru etapa următoare.

Arhitectura dirijata de evenimente (1)

- Arhitectură distribuită frecvent utilizată.
- Adecvată sistemelor care colectează date în timp real şi produc acțiuni ca răspuns la datele achiziționate.
- Alcătuită din componente care procesează evenimente asincron, în paralel.

Intr-un sistem bazat pe IOT exista 2 tipuri de elemente (“obiecte”):

- senzorii, care detecteaza producerea anumitor evenimente şi genereaza semnale electrice
- elemente de actionare (actuators), care produc acțiuni ca răspuns la evenimentele detectate de senzori.

Exemplu:

- un termometru amplasat într-un spațiu este un senzor care produce un eveniment (semnal electric) atunci cand temperatura depăşeste o anumită limită.
- rezultatul tratării evenimentului este comanda pornirii aparatului de aer condiționat

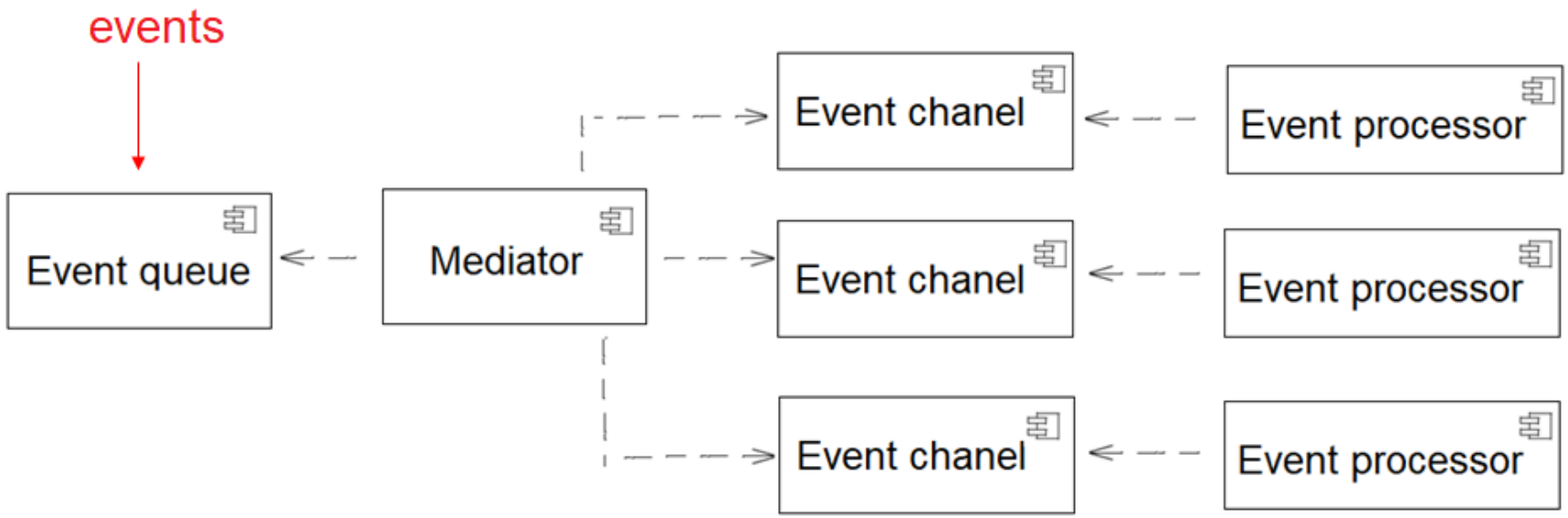
Arhitectura dirijata de evenimente (2)

Topologia Mediator

- 4 tipuri de componente: coada de evenimente, mediatorul, canale eveniment şi procesoare de evenimente.
- ***Mediatorul***

- *preia evenimente* din **coada de evenimente** şi descompune tratarea fiecarui eveniment de intrare în paşi de procesare care pot fi executați secvențial sau în paralel.

- genereaza *evenimente de procesare* care sunt *transmise în canale eveniment specifice*.



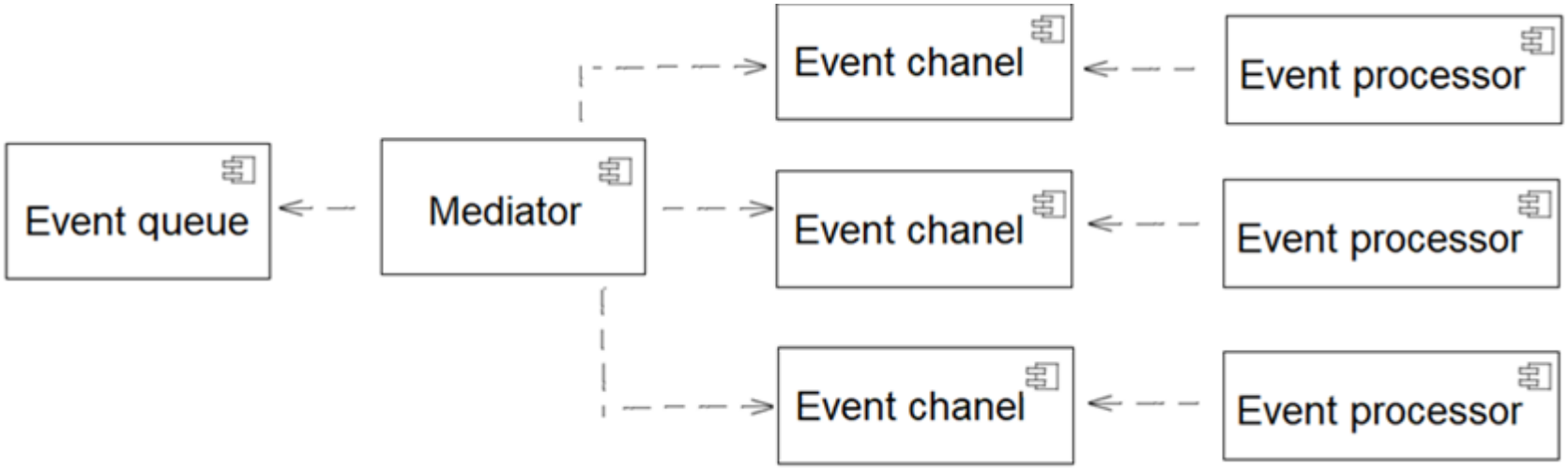
Arhitectura dirijata de evenimente (3)

Topologia Mediator

- Fiecare ***procesor de evenimente*** preia evenimentele dintr-un canal de evenimente şi efectuează prelucrări specifice, care implementează logica aplicatiei.
- ***Canalele de evenimente*** pot fi:

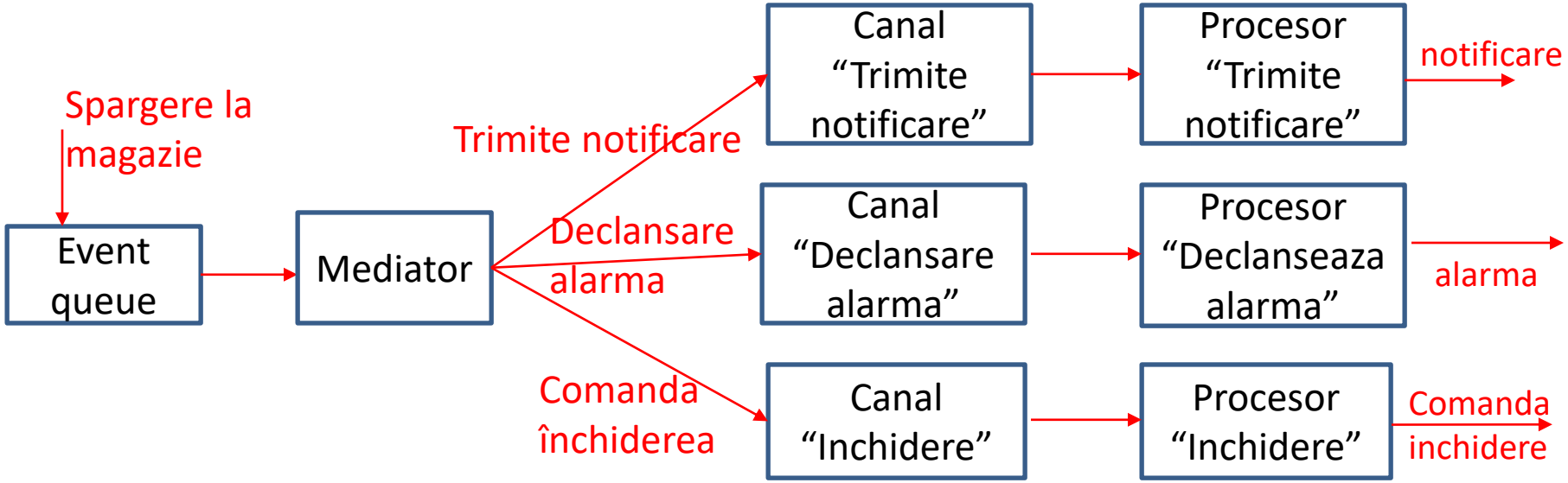
- de tip "coada de mesaje" – evenimentele dintr-un canal sunt preluate de un singur processor;

- de tip "*topics* and *subscriptions*" (un mesaj este transmis la mai mulți consumatori înregistrați pentru un anumit tip de mesaj) - evenimentele dintr-un canal sunt preluate şi tratate în paralel de mai multe procesoare de evenimente.



Arhitectura dirijata de evenimente (4)

Exemplu de prelucrare într-un sistem cu topologie Mediator:

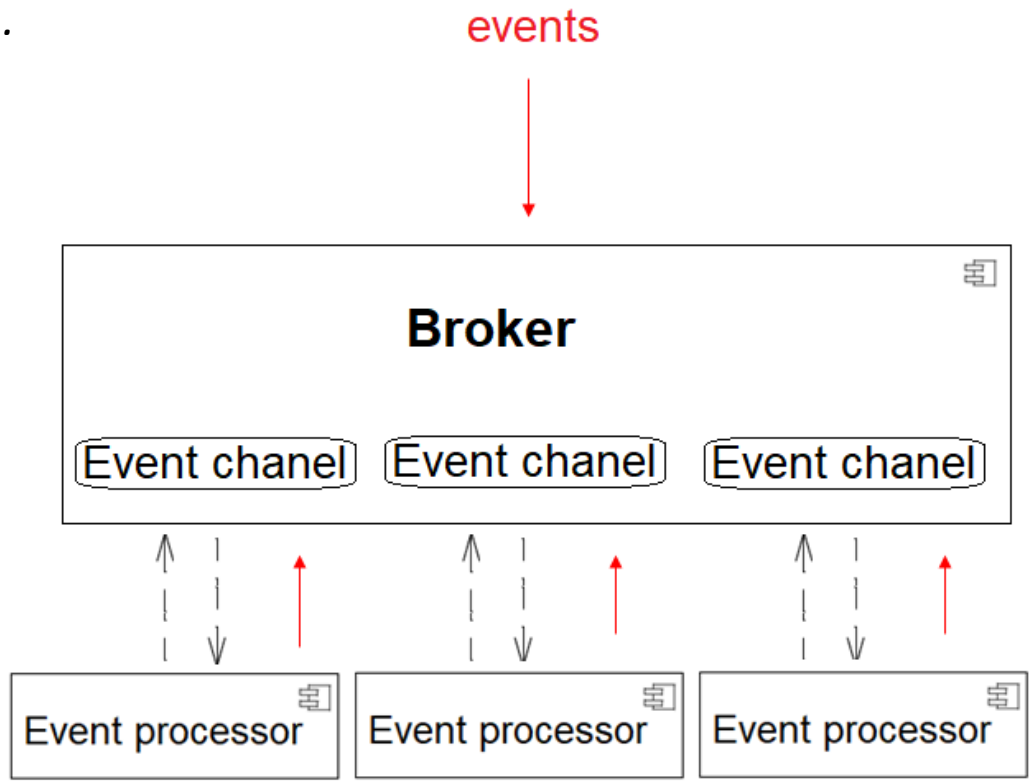


Arhitectura dirijata de evenimente (5)

Topologia Broker

- Broker-ul preia evenimentele şi le dirijează către procesoarele de evenimente prin canale de evenimente specifice.
- Canalele de evenimente sunt conținute în componenta Broker şi pot fi de tip “coada de mesaje” sau “*topics* and *subscriptions*” .

- Fiecare procesor de evenimente prelucrează un eveniment şi poate genera un nou eveniment pe care-l trimite la Broker într-un canal de evenimente specific.

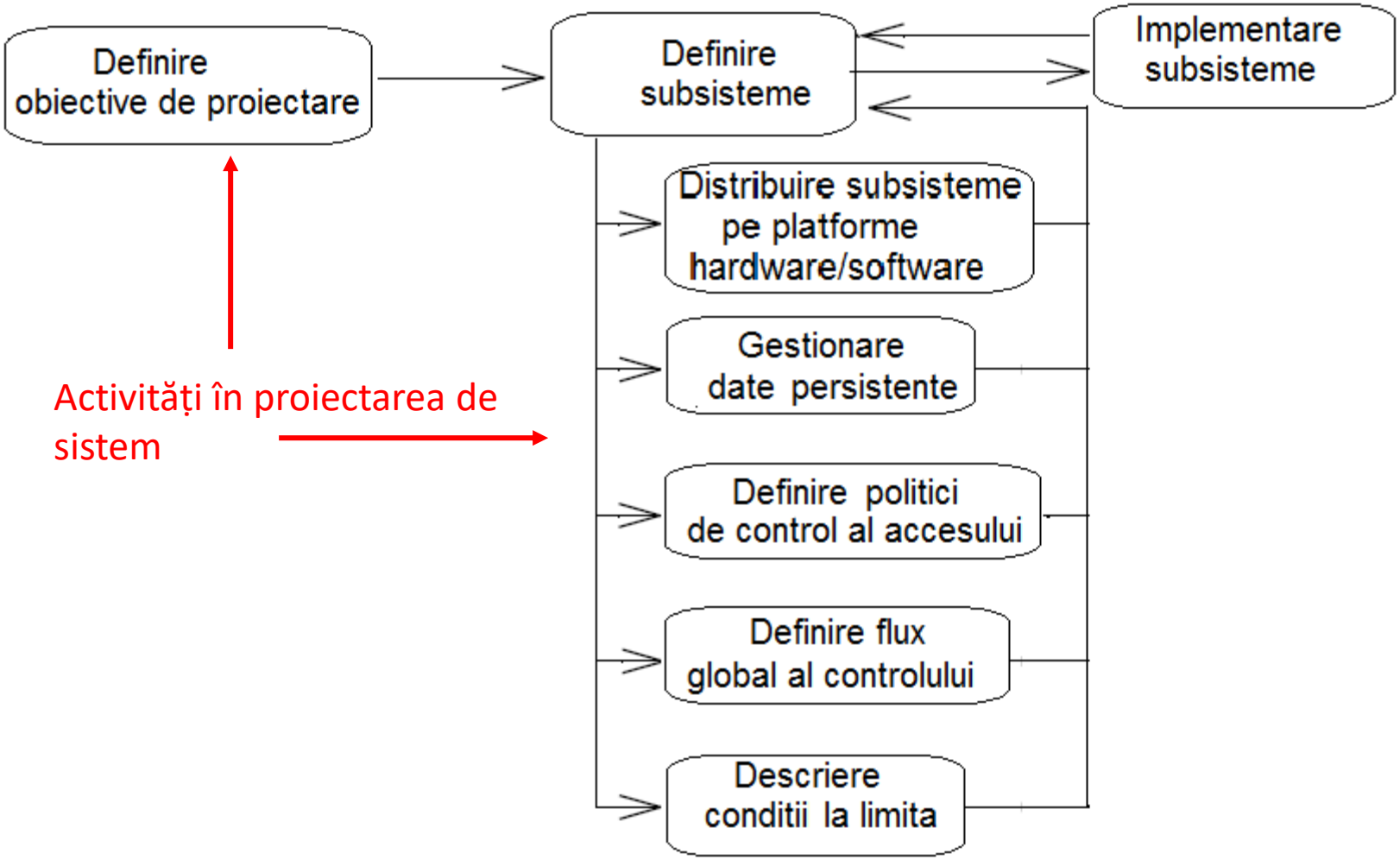


Arhitectura dirijata de evenimente (6) avantaje si dezavantaje

- **Complexă**: prelucrare asincronă, distribuită.
- Lipsa de atomicitate în procesarea unei tranzactii (eveniment): **dificil de divizat procesarea în pasi executati în paralel.**
- **Scalabilitate mare.**
- Adecvata atat pentru aplicatii mici cat si pentru aplicatii foarte mari.
- **Componente foarte decuplate.**
- Componentele ***Event processor*** pot fi modificate fara a afecta pe celelalte.

  In cazul topologiei *Mediator* schimbarea unei componente *Event processor* poate necesita o schimbare a componentei Mediator.
- **Testare dificila**: trebuie generate evenimente pt a simula natura asincrona a prelucrarilor.
- **Performanță înaltă** datorită prelucrărilor asincrone paralele care depăşesc costul transferului de mesaje prin cozi.
- Dezvoltarea poate fi complicată: natura asincronă, tratarea condițiilor de cădere, procesoare de evenimente care nu răspund, etc.

Definirea arhitecturii sistemului
- proces iterativ -



Rafinarea descompunerii în subsisteme

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

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

Distribuirea subsistemelor pe platforme hardware/software

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

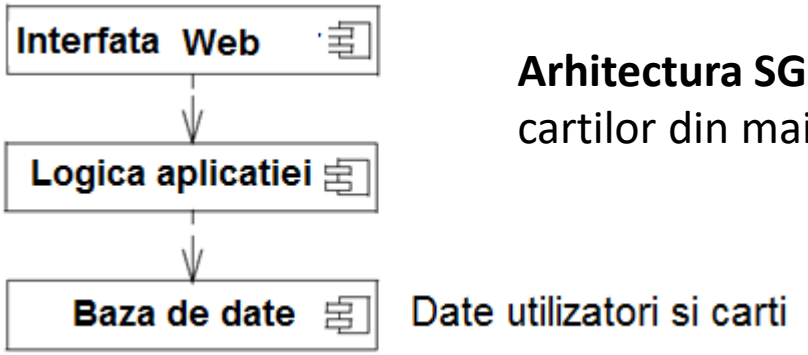
Identificarea şi gestionarea datelor persistente

Date persistente:

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

Mecanisme pentru asigurarea persistentei:

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



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

Gestionarea datelor persistente

- **Intrebari care influenteaza proiectarea unei baze de date**
  - Care este rata de cereri estimata?
  - Cat de des este accesata baza de date?
  - 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 în 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 între tabele.**

Controlul accesului

- Intr-un sistem multi-utilizator fiecare tip de utilizatori (actor) are anumite drepturi de acces la resursele sistemului.
- Se determina obiectele partajate de actori: 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 diferitilor actori la obiectele partajate poate fi modelat prin **matricea de control a accesului**:

Actor	**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 (operatiile pe care le poate efectua fiecare actor cu fiecare obiect partajat)

Fluxul global al controlului (1)

Fluxul controlului: secventierea actiunilor/operațiilor la executia sistemului.

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

— Dirijat procedural (procedure-driven control)

— Dirijat de evenimente (event driven control)

— Bazat pe fire de executie (thread-uri)

Control dirijat procedural

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

Fluxul global al controlului (2)

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

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

Control dirijat de evenimente

- Sistemul contine o bucla principala in care se asteapta un eveniment extern.
- Atunci cand se produce un eveniment, el este transferat obiectului corespunzator.
- Evenimentele pot fi tratate secvential sau in paralel
- Este o structura de control simpla, care centralizeaza toate intrarile in bucla principala.
- Este uzuala in sistemele bazate pe evenimente generate de interfata grafica utilizator
- Nu este adecvata implementarii secventelor in mai multi pasi.

Fluxul global al controlului (3)

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

```
Iterator subscribers, eventStream;
Subscriber subscriber; Event event; EventStream eventStream;

while (eventStream.hasNext()) {
    /* se extrage evenimentul din eventStream si se transmite obiectelor care s-au
    inregistrat pentru 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 inregistrate.

Fluxul global al controlului (4)

Control bazat pe fire de executie

- Threadurile sunt executate in paralel
- Pot fi create/distruse la diferite momente de timp
- Intr-un thread se pot astepta intrari de la un actor.

Exemplu (Java): evenimentele sunt tratate in paralel

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

while (eventStream.hasNext()) {
    event = eventStream.next();
    eventHandler = new EventHandler(event)
    thread = new Thread(eventHandler);
    thread.start(); /  * porneste executia threadului   */
}
```

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

Lecturi suplimentare – stiluri arhitecturale

1. https://openclassrooms.com/en/courses/6397806-design-your-software-architecture-using-industry-standard-patterns/6896176-layered-architecture
2. https://www.jinfonet.com/resources/bi-defined/3-tier-architecture-complete-overview/
3. https://martinfowler.com/eaaDev/OrganizingPresentations.html
4. https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch02.html

Modelarea arhitecturii software în UML

Prof. univ. dr. ing. Florica Moldoveanu

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

UML pentru modelarea arhitecturii software

Reprezentarea arhitecturii

- *Diagrame de componente* – pentru redarea relatiilor structurale dintre componentele software

- *Diagrama de distributie* – pentru a reda repartitia artefactelor software pe noduri hardware, la momentul executiei.

Reprezentarea fluxului global al controlului, procese în sistem

- *Diagrame de activitate*

Gruparea elementelor de modelare în subsisteme

- *Pachete UML*

Componente UML(1)

In UML 1.x

O componenta este un element software fizic din componenţa unui sistem: fişier executabil, componentă binară, document, fişier cod sursa, fisier date, tabelă a unei baze de date.

➢ O componentă binară este o parte fizică substituibilă a unui sistem, care realizează si este în conformitate cu un set de interfețe.

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

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

Reprezentarea grafica a unei componente în UML 1.x

Componente UML(2)

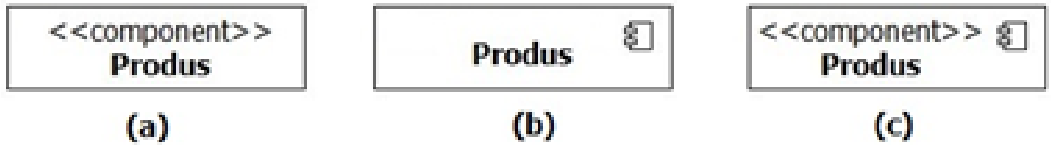
In UML 2.x

❖ **O componentă este o construcţie logică definită 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.

O componenta poate fi:

- **Un subsistem**
  - care nu are un corespondent la executie - de ex. subsistemul care reprezinta un nivel într-o arhitectură ierarhică
  - care are un corespondent la executie – de ex. un server de baze de date
- **O componentă binara**



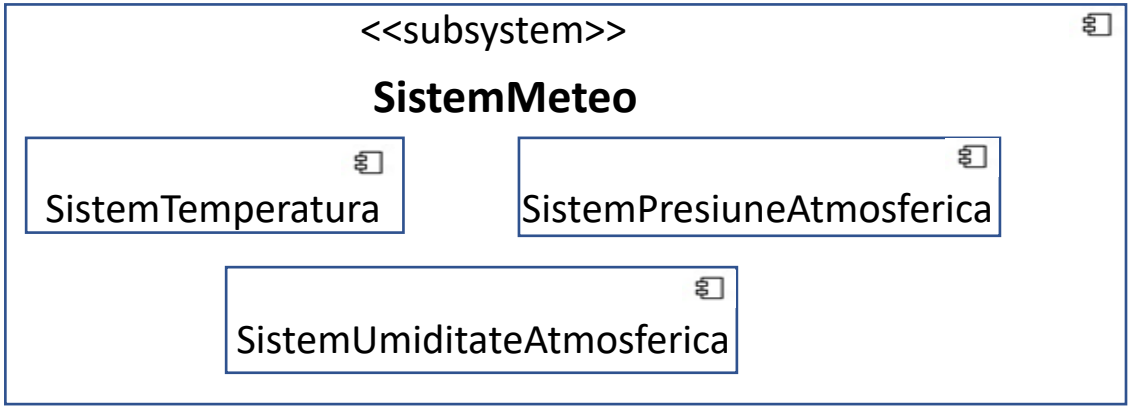
Reprezentarea grafica (de nivel inalt) a unei componente în UML 2.x.

Componente UML(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»

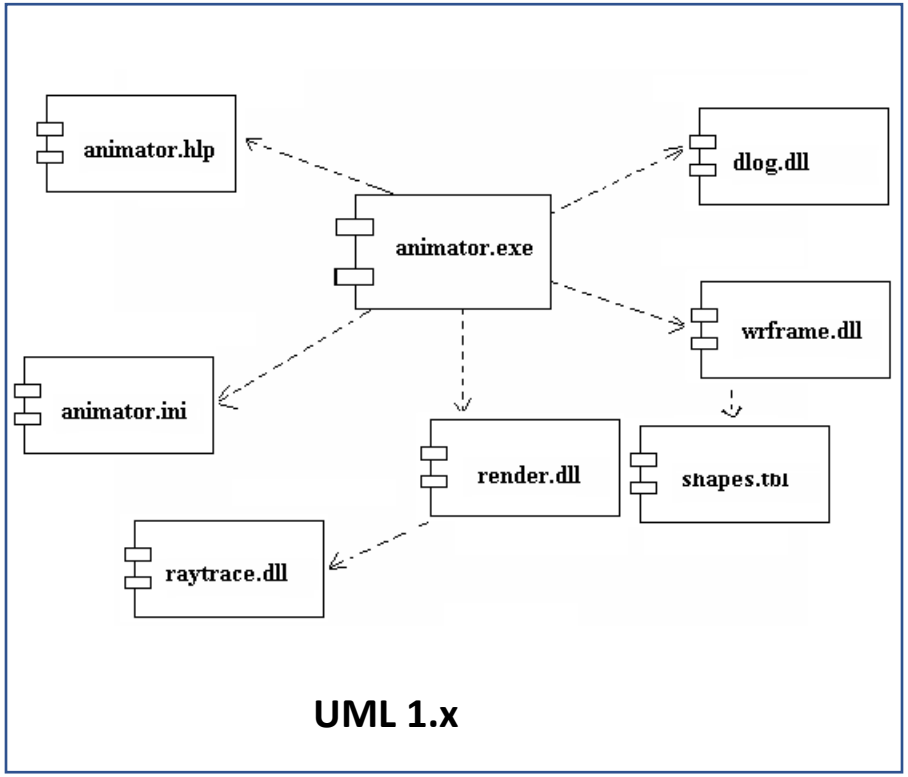
- Componentă bazată pe tranzacţii.

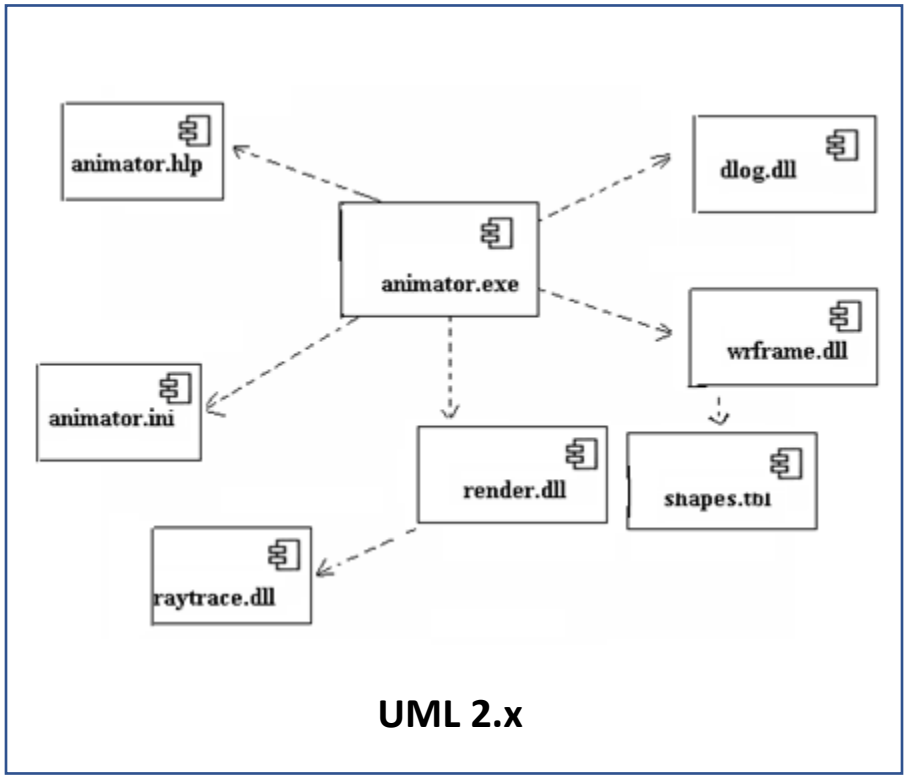
«service»

- Componentă funcţională, fără stare.

Diagrame de componente

- Redau relațiile structurale dintre componentele software ale unui sistem.

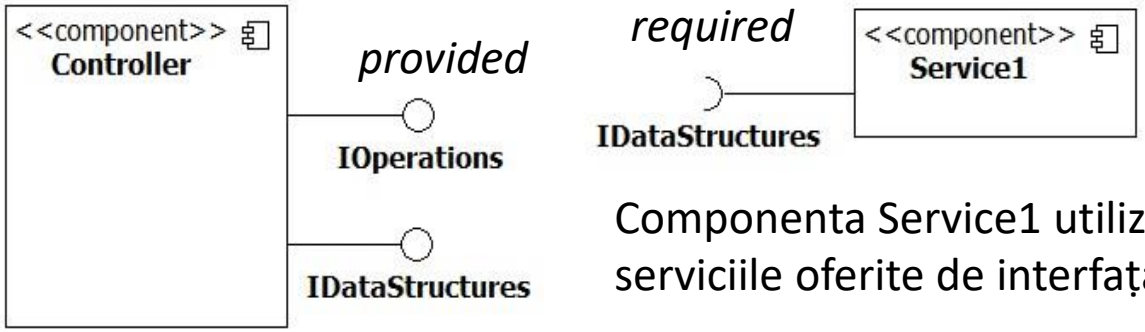




Interfeţe furnizate şi interfeţe necesare

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

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



Componenta Service1 utilizează (necesită) serviciile oferite de interfața IDataStructures

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

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

- necesare implementării componentei, sau

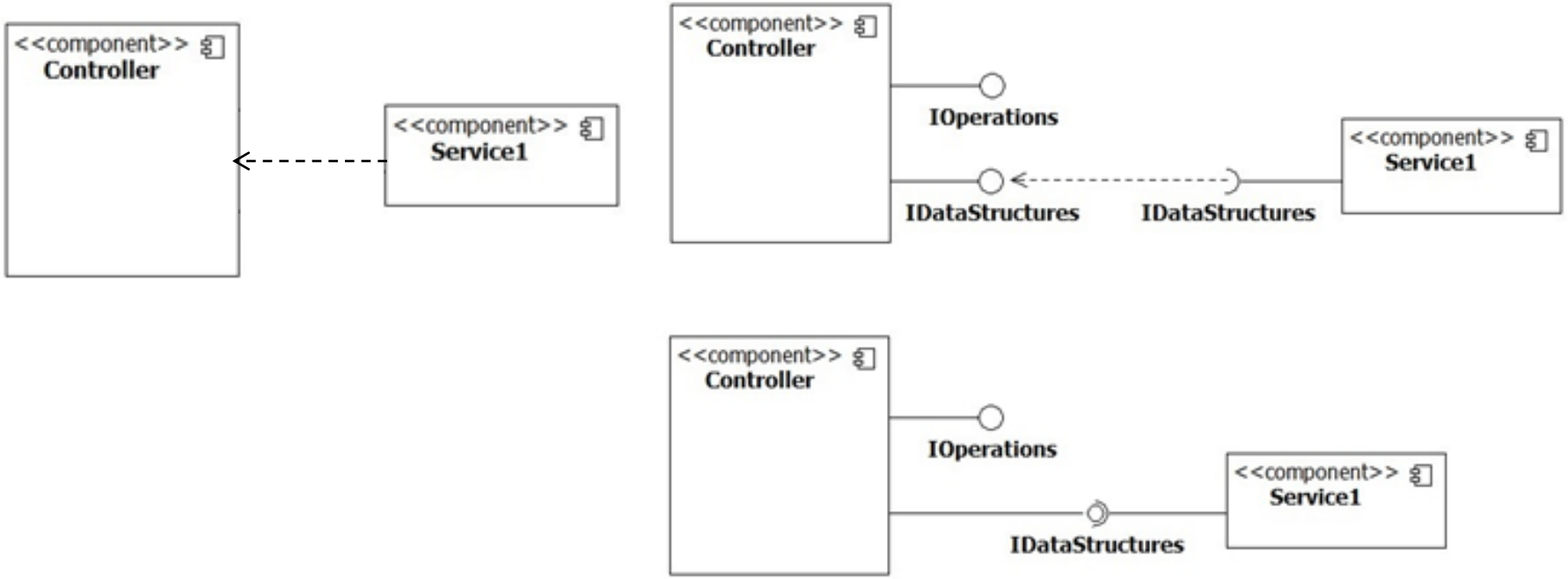
- necesare entităţilor care *realizează* componenta.

Relaţii între componente (1)

Dependența

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

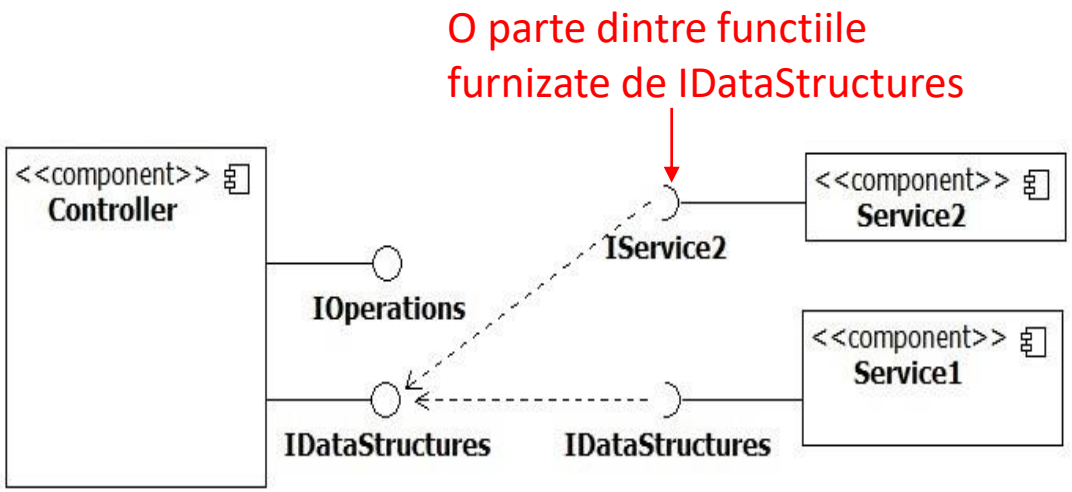
Reprezentări pentru relația de dependenţă



Relaţii între componente (2)

Dependența (cont)

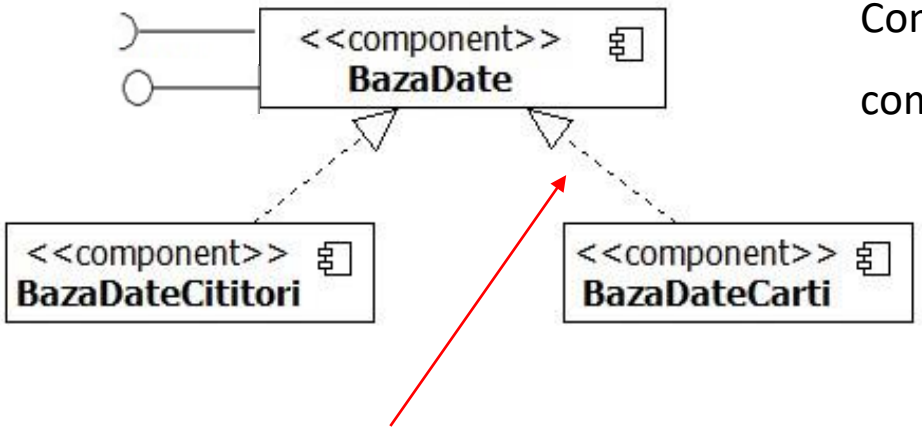
Daca interfaţa necesară unei componente conţine un subset dintre funcţiile furnizate de o interfaţă, numele interfeţei necesare poate fi diferit de cel al interfeţei care furnizează funcţiile.



Relaţii între componente (3)

Realizarea

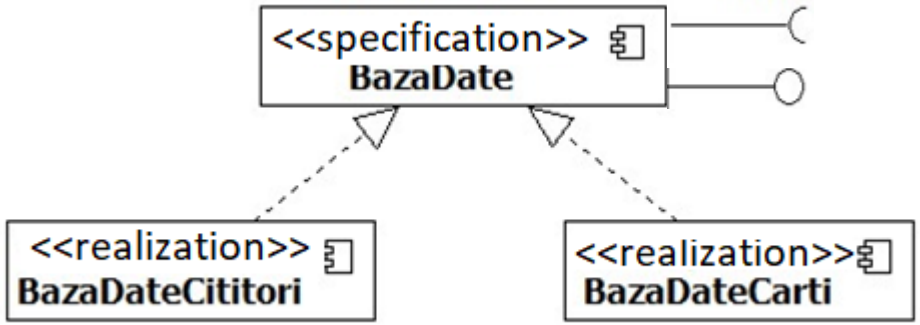
- **O componentă poate fi doar o specificaţie a unui comportament**, prin intermediul interfeţelor furnizate şi necesare.
- Comportamentul poate fi realizat (implementat) de alte componente, caz în care între componenta specificaţie şi componentele care o realizează există o **relaţie de realizare**.



Reprezentarea relației de realizare

Componenta **BazaDate** este realizata de componentele **BazaDateCititori** si **BazaDateCarti.**

Se pot folosi adnotari:



Relaţii între componente (4)

«specification»

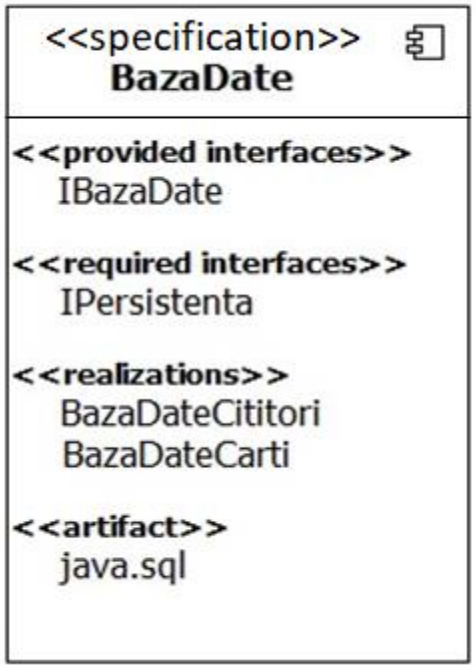
- Se foloseşte pentru a adnota o componentă care reprezintă numai o specificaţie a unui comportament, prin interfeţele furnizate şi cele necesare. Definiţia sa nu precizează entităţile care realizează specificaţia. Acestea pot fi ataşate componentei de tip specificaţie printr-o *relaţie de realizare*.
- O componentă specificaţie este o abstractie a unui comportament care poate avea mai multe realizari (implementari)

«realization»

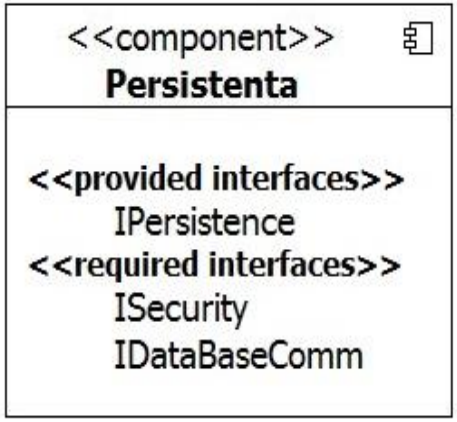
- Se foloseşte pentru a adnota o componentă care implementează comportarea specificată printr-o altă componentă.
- Stereotipurile «specification» şi «realization» sunt utilizate pentru a modela componente cu definiţii separate pentru specificaţie şi realizare, unde o specificaţie poate fi realizata de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

Diferite vederi ale unei componente

Vederea “black box” – prin intermediul simbolurilor interfeţelor furnizate/necesare:



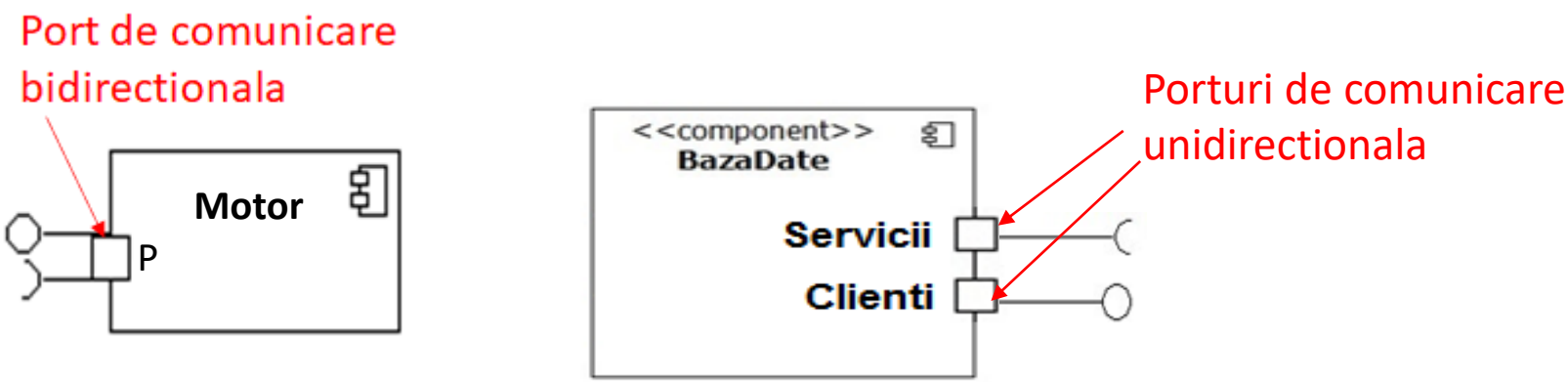
Vederea “white box”:



Componenta BazaDate este reprezentată fizic (la executia sistemului) de *artefactul* java.sql.

Porturi si conectori (1)

- **Un port** desemnează un **punct de comunicare între o componentă și mediul său extern.**
- **Un port este asociat cu una sau mai multe interfete.**
- O componenta poate fi utilizata in orice context care satisface constrangerile reprezentate prin porturile sale.
- Un port are asociat un nume și se reprezintă ca un dreptunghi plasat pe una dintre laturile dreptunghiului care încadrează componenta.



Porturi si conectori (2)

- **Un conector** este **o legătură care reprezintă comunicarea la momentul executiei dintre părti interne ale unei componente sau dintre o parte internă si mediul extern** .
- O instanta a unei componente: entitate prin care este reprezentata fizic componenta la momentul executiei.
- Conectorii sunt de două tipuri:

  - **Conectori de delegare**

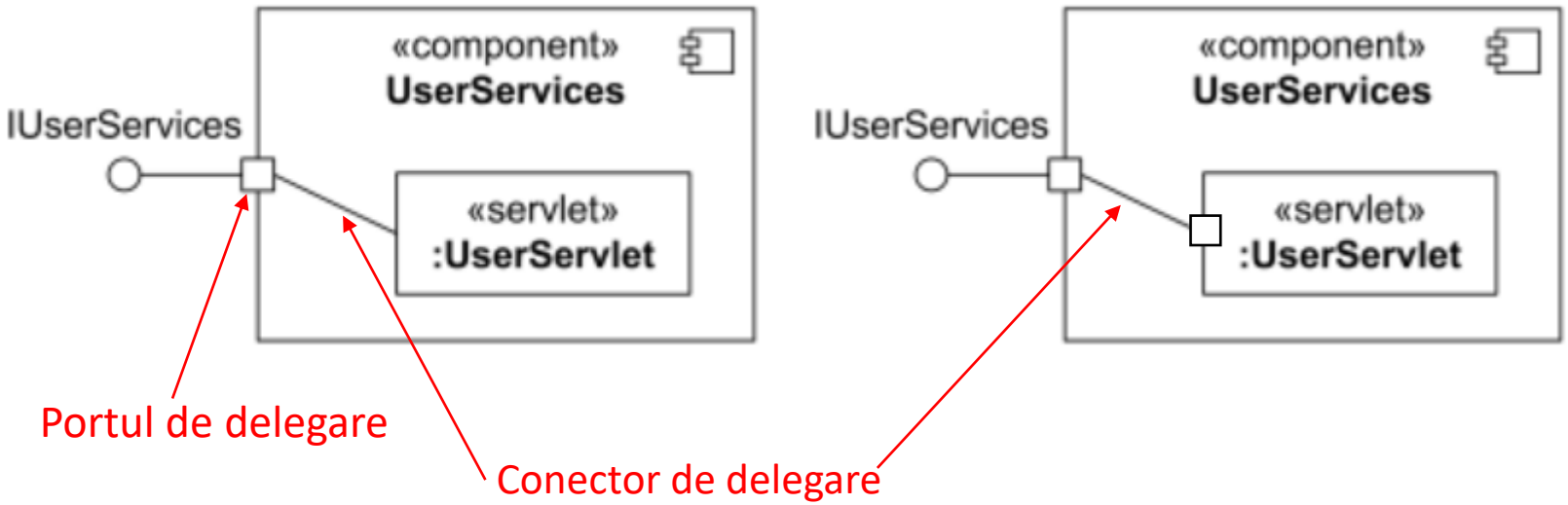
    - Leaga un port al unei componente cu o instanta a unei subcomponente
    - Reprezinta transmiterea evenimentelor din exteriorul componentei în interiorul său şi invers, printr-un port
    - «Deleaga» comportamentul extern reprezentat printr-un port unei părți interne

  - **Conectori de asamblare**

    - reprezinta comunicarea dintre partile interne ale unei componete

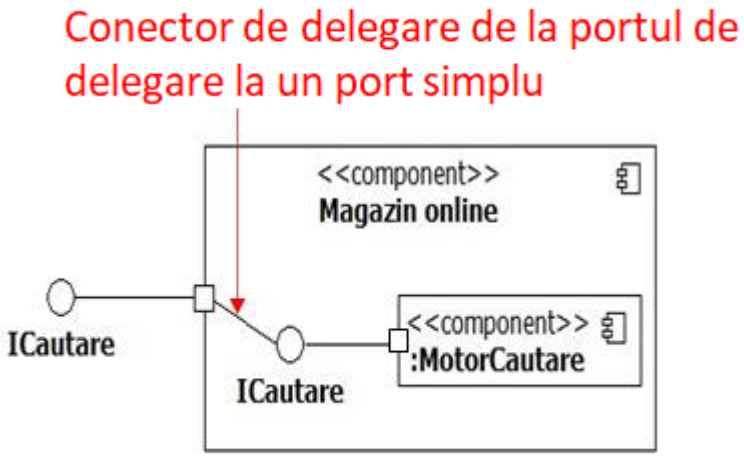
Porturi si conectori (3)

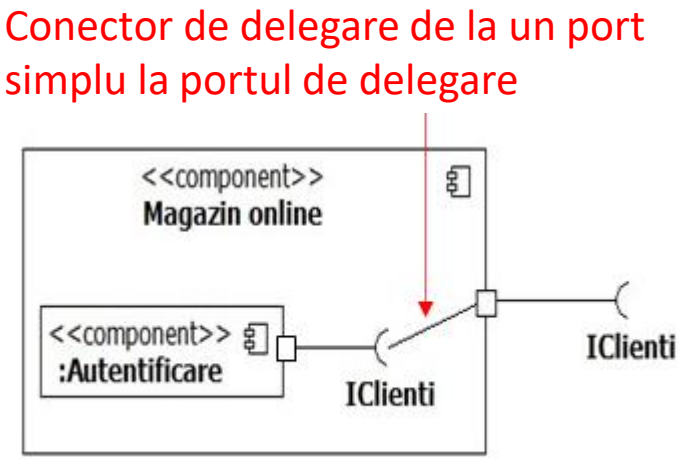
- Un conector de delegare leaga portul de delegare de partea interna care implementeaza functionalitatea portului de delegare.



Operatiile expuse de interfata *IUserServices* sunt implementate de o instanta a componentei *:UserServlet*

Porturi si conectori (4)





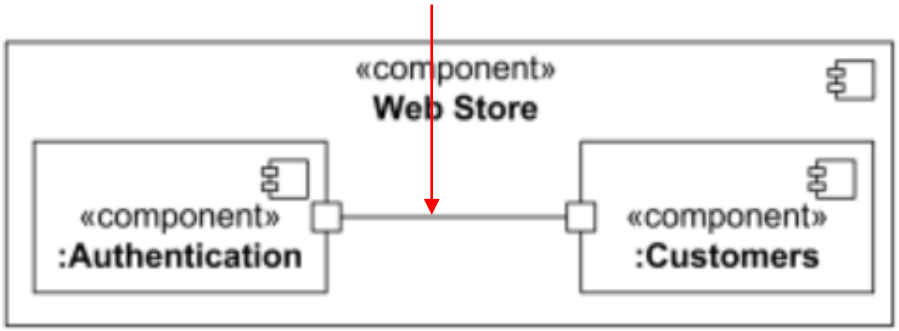
- Mesajele primite de componenta *Magazin online* prin interfața *ICautare* sunt redirecționate către o instanță a subcomponentei *MotorCautare* (care implementeaza operatiile expuse de componenta *Magazin online* prin interfața *ICautare*)

- Mesajele transmise de o instanta a subcomponentei *Autentificare* prin interfața *IClienti* sunt redirecționate către interfata *IClienti* a componentei *Magazin online.*

Porturi si conectori (5)

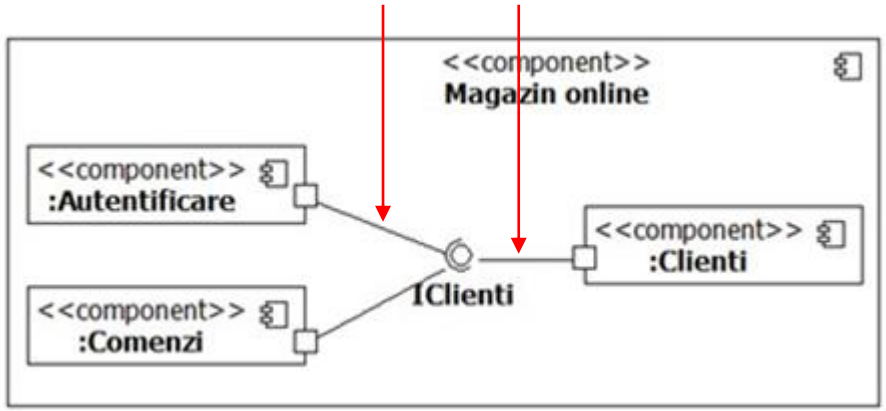
- **Un conector de asamblare** conectează două sau mai multe instanțe ale subcomponentelor unei componente.
- Se reprezinta ca o legatura între 2 porturi sau o legatura între un port şi o interfață.

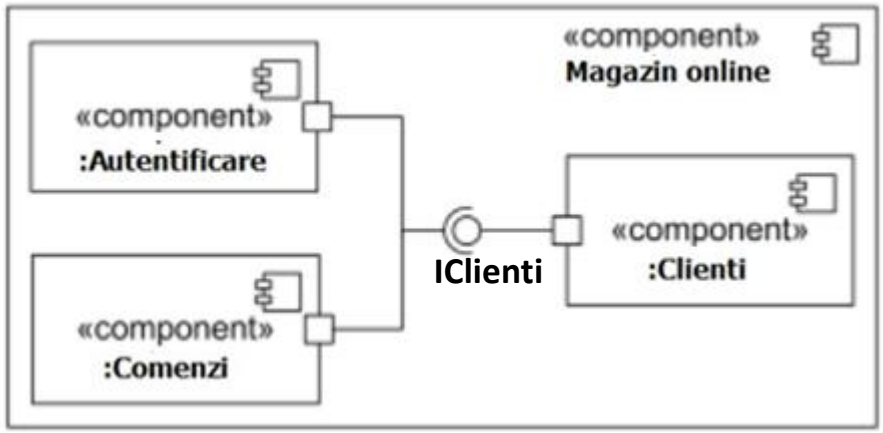
Conector de asamblare



Instantele *:Authentication* şi *:Customers* comunica prin porturile conectate, care pot fi associate cu mai multe interfete

Conectori de asamblare care conecteaza porturi simple

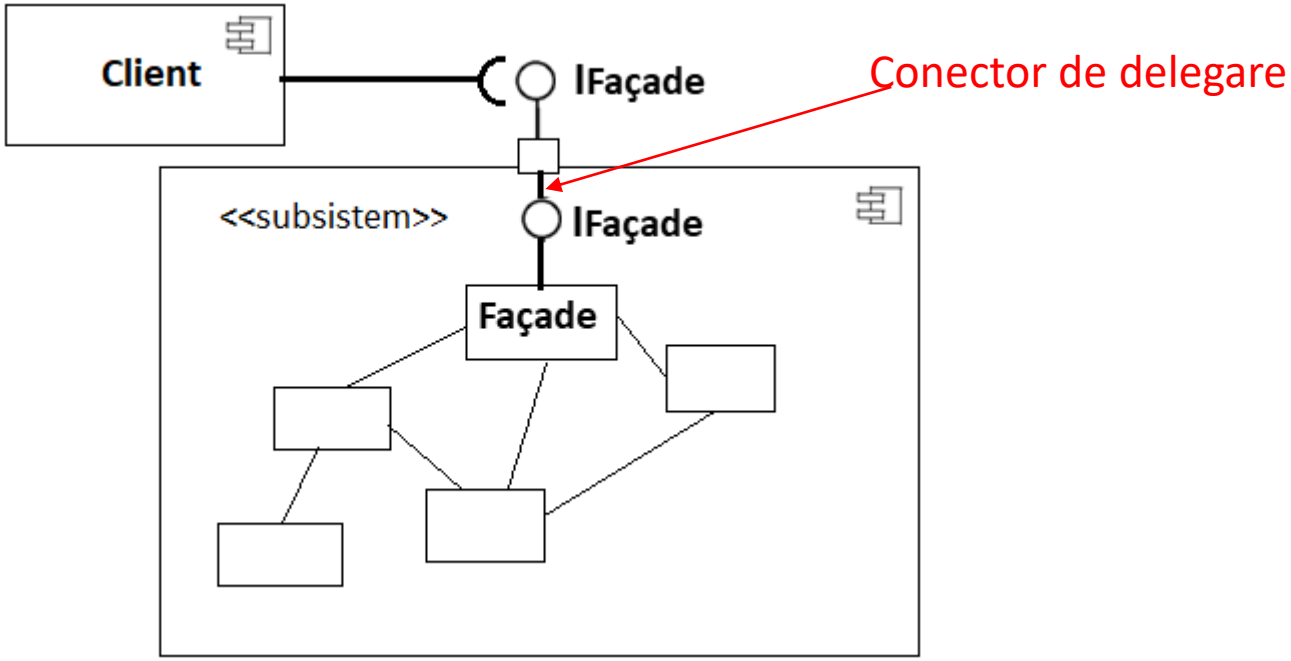




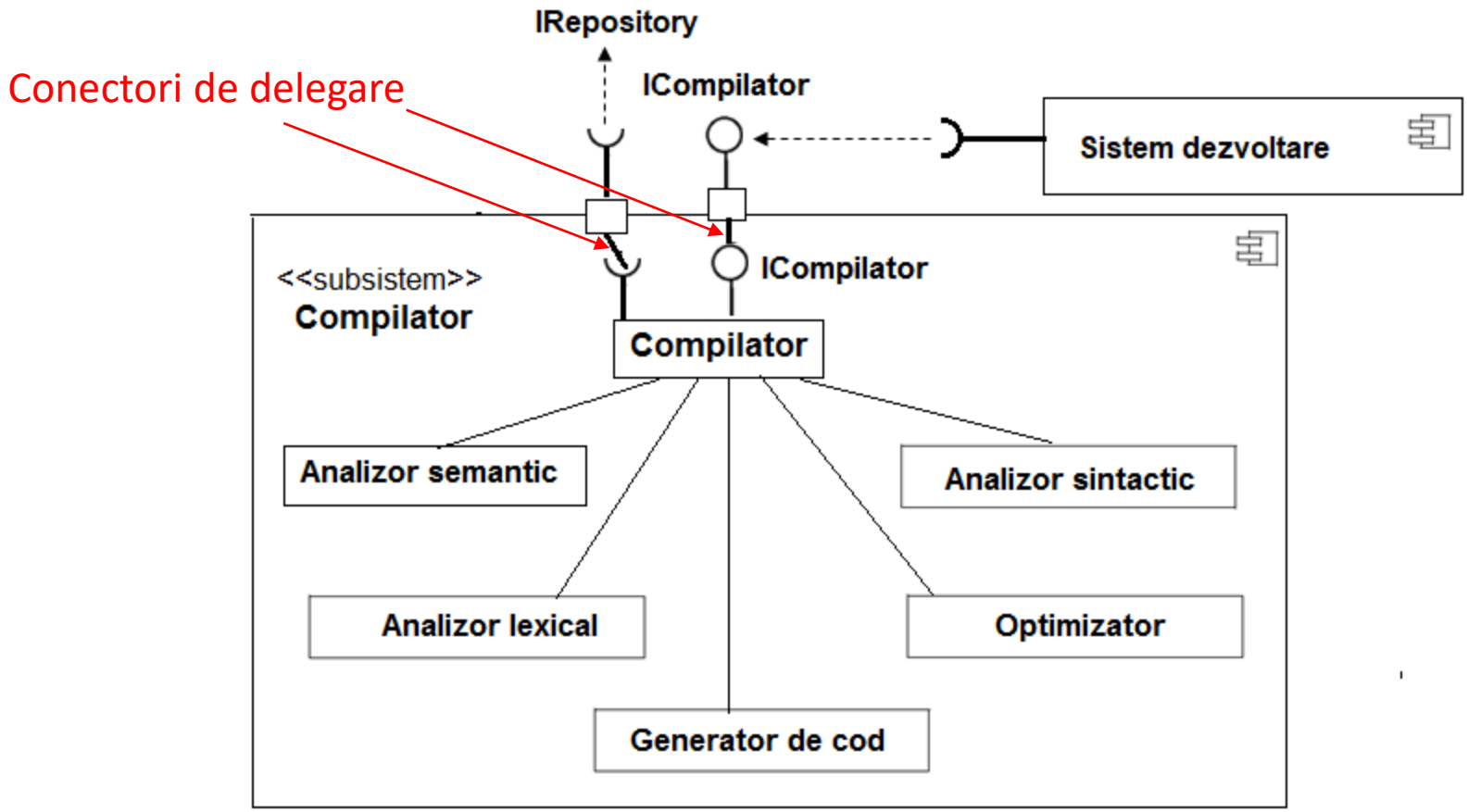
Porturi si conectori (6)

❖ **Sablonul de proiectare Façade** ofera un mecanism adecvat pentru gruparea claselor în subsisteme şi reducerea cuplarii între subsisteme: interfata subsistemului este implementata de o clasa **Façade** care deleaga cererile clientilor catre obiectele claselor subsistemului.

- Subsistemul este slab cuplat cu celelalte subsisteme
- Modificarea subsistemului nu afecteaza subsistemele care utilizeaza I**Façade**



Porturi si conectori (7)



Subsistemul **Compilator** implementeaza interfata I**Compilator** si utilizeaza interfata **IRepository.**

Clasa “faţadă” **Compilator** deleaga:

- cererile de servicii adresate prin interfata **ICompilator** catre obiectele claselor subsistemului
- cererile obiectelor din clasele subsistemului pentru serviciile oferite de interfata **IRepository**

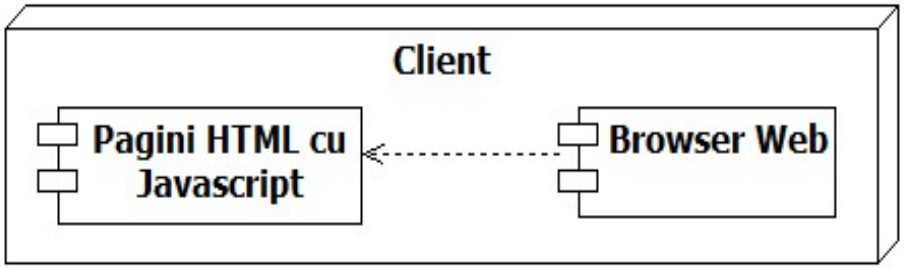
Utilitatea diagramelor de componente

- Furnizează o vedere arhitecturală de nivel înalt a unui sistem software, 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.

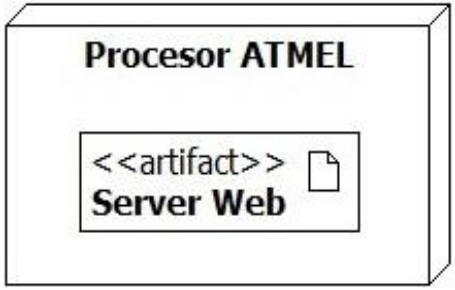
Diagrame de distributie (Deployment diagrams)

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

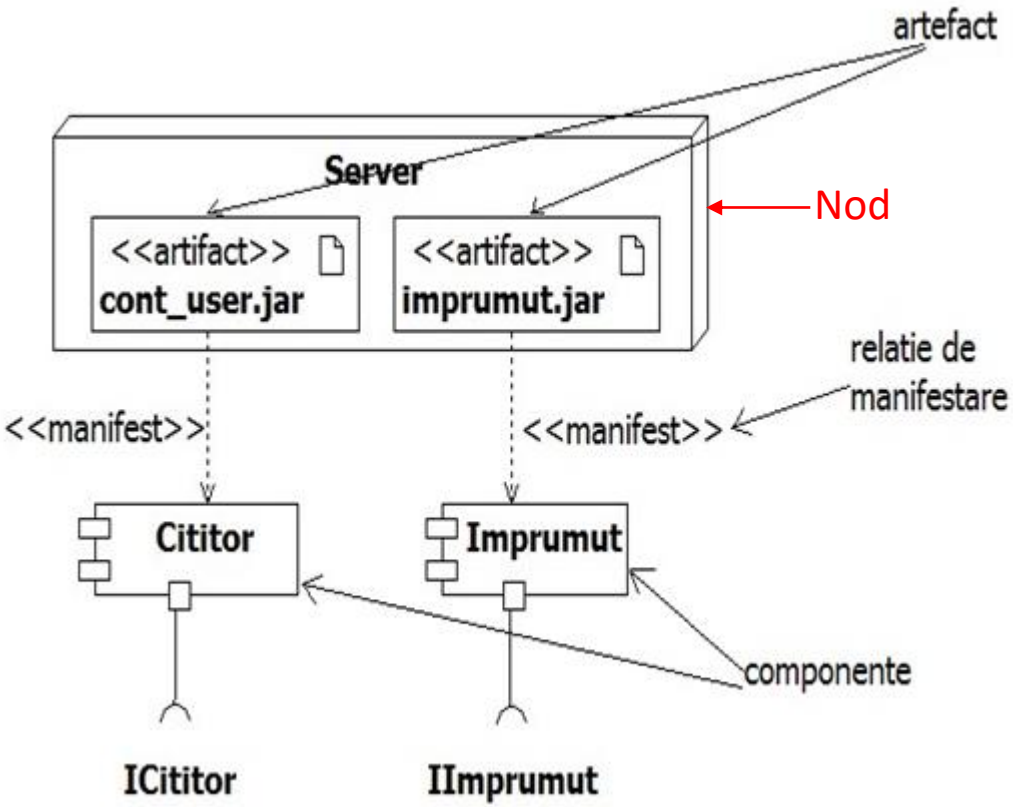
În UML 1.x, componentele sunt distribuite direct pe noduri, ele fiind entităţi fizice din componenţa unui sistem.



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



Componente, artefacte, noduri



Cont_user.jar este o implementare a componentei **Cititor**

Imprumut.jar este o implementare a componentei **Imprumut**

Noduri

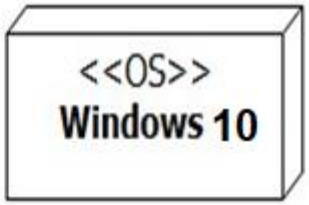
❖ Nodurile din diagramele de distribuție reprezintă resurse computaționale. Acestea sunt de două tipuri:

- **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>> respectiv ***<<executionEnvironment>>***

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



Noduri si artefacte

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

<<Server aplicatie>>
IBM system z

<<mobile device>>
smartphone

❖ ***Artefactele*** sunt entităţi fizice 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.**

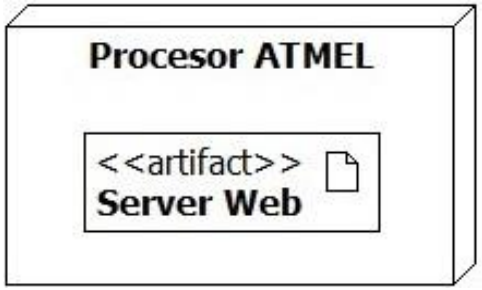
Reprezentarea artefactelor

BDComenzi

Distribuția artefactelor pe noduri

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

- prin amplasarea artefactului pe suprafața nodului



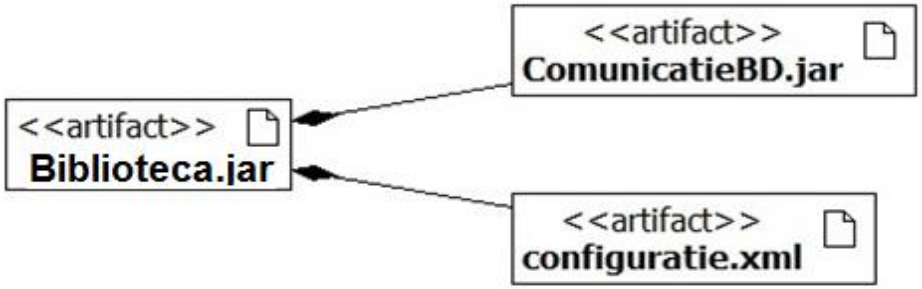
- printr-o relatie de distributie de la artefact catre nod

O alta metoda de reprezentare foloseste un specificator de distributie (a se vedea “UML practic”, Ed MatrixRom, 2014)

Relații între artefacte

❖ Între artefacte se pot stabili relații de asociere şi dependență.

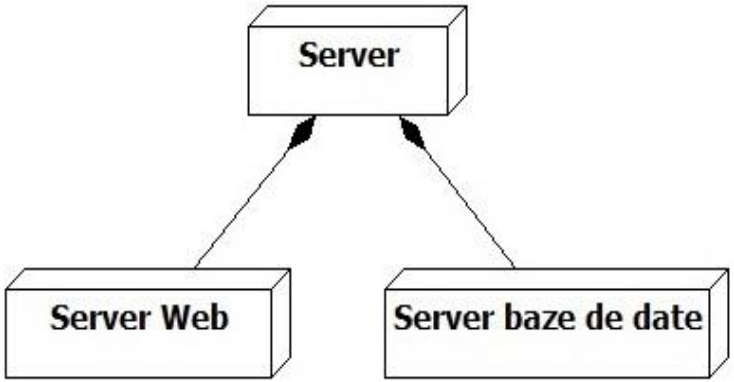
- O ***asociere*** între două artefacte este de tip ***compunere*** sau ***agregare*** – cu aceeasi semnificatie ca relatiile corespunzatoare dintre clase.



- ***Dependenţa***

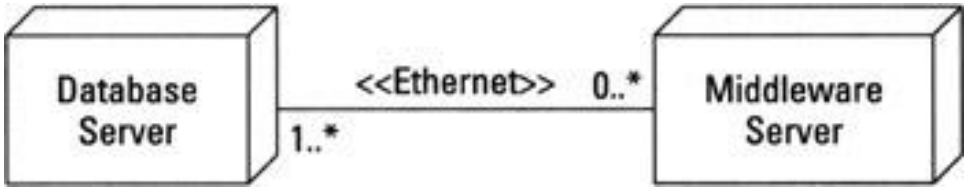
Relații între noduri (1)

Compunerea



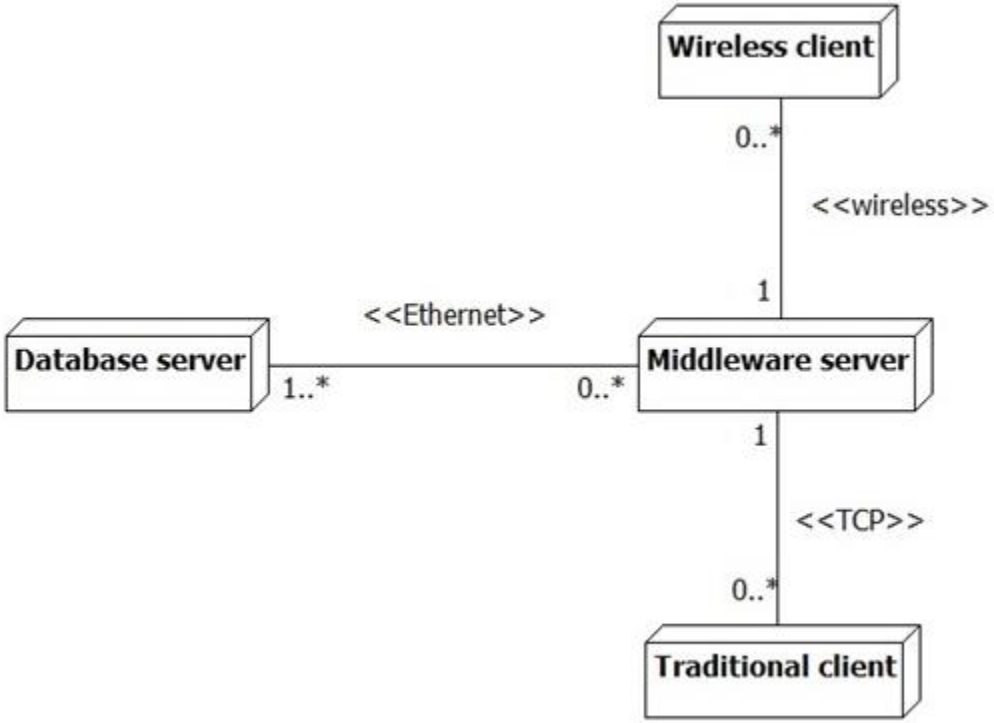
Asocierea

- 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.

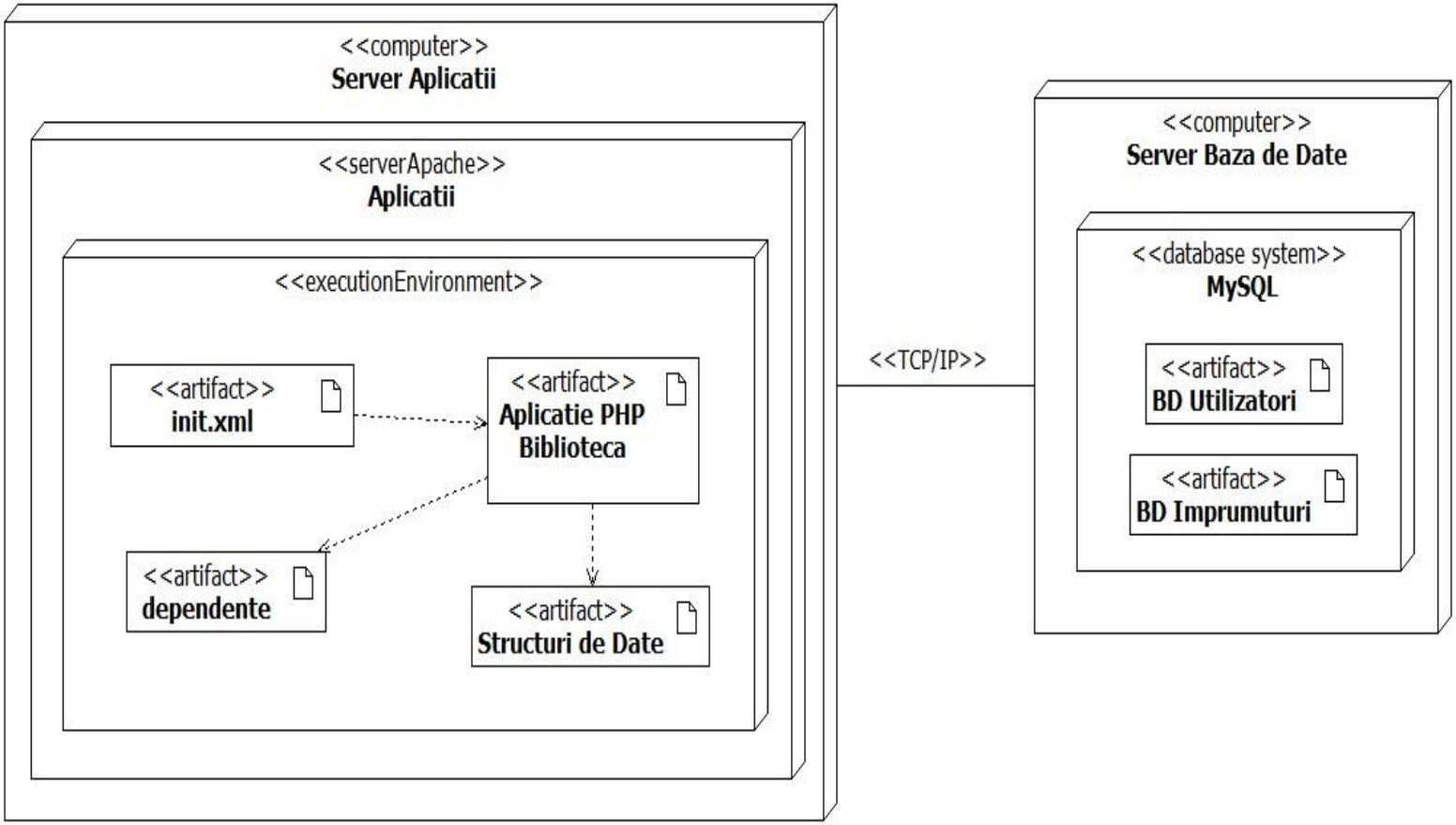


Relatii între noduri (2)

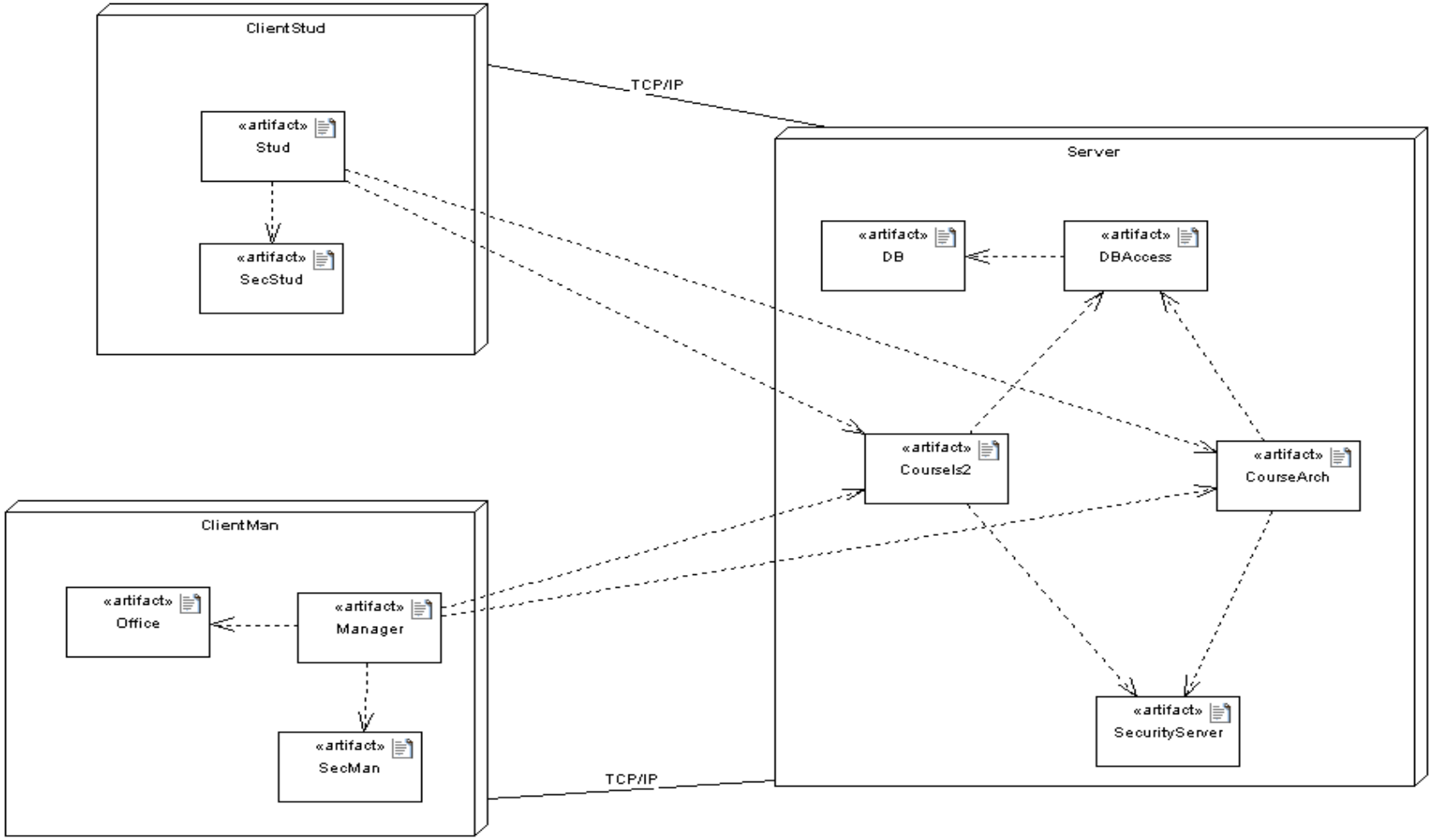
- Capetelor asocierii le pot fi atribuite multiplicităţi, reprezentând numărul de instanțe ce pot fi implicate în comunicare.



Diagramă de distribuție în UML 2.x



Diagramă de distribuție în UML 2.x



Lecturi suplimentare – diagrame de componente si de distributie

https://www.uml-diagrams.org/composite-structure-diagrams/connector.html

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

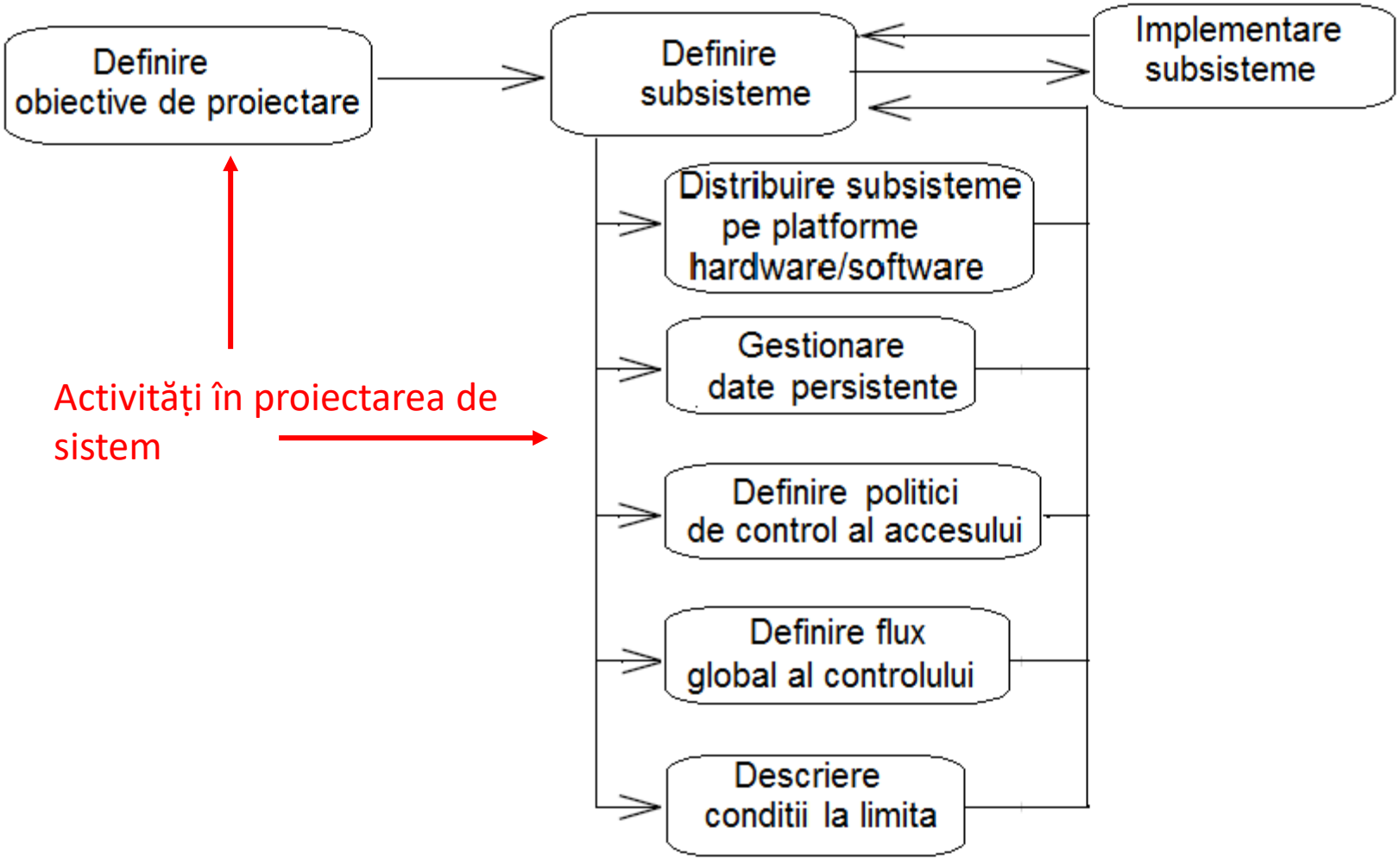
https://www.lucidchart.com/pages/uml-deployment-diagram/#section_5

Proiectarea arhitecturala -3

Prof. univ. dr. ing. Florica Moldoveanu

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

Activități în proiectarea de sistem



Identificarea condițiilor limită (boundary conditions)

- Proiectare de sistem considera, în principal, **comportamentul sistemului în starea sa stabila**.

- Este necesar sa se examineze şi **conditiile limită în functionarea sistemului**:

  - pornirea, initializarea, oprirea

  - căderile majore care pot duce la pierderea datelor, întreruperile în comunicatia prin retea, etc.
    – cauzate de:
    - un defect software
    - o greseala de operare
    - o întrerupere a sistemului electric, etc.

  - atacuri la securitatea sistemului

  - conditii de utilizare pentru administrarea sistemului

- Functionarea sistemului în aceste situatii este descrisa prin **cazuri de utilizare limită sau administrative** (**boundary use cases/administrative use cases**) – deoarece actorul care interactioneaza cu sistemul in aceste cazuri de utilizare este de regula administratorul de sistem

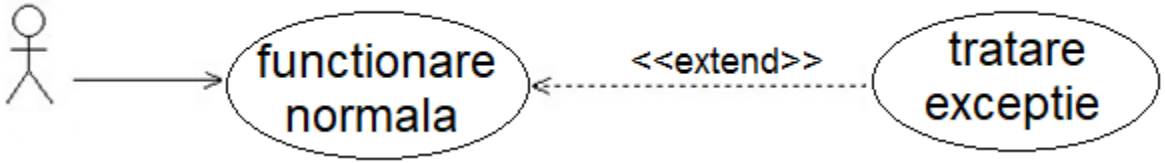
Cazuri de utilizare în condiții limită

Pornirea, oprirea si configurarea

- Pentru fiecare componenta executabila a sistemului se adauga 3 cazuri de utilizare: pornirea, oprirea si configurarea (initializarea) componentei.

Tratarea exceptiilor

- Anumite exceptii pot fi tolerate de sistem şi incluse în proiectarea unor componente.
- Pentru fiecare tip de **cădere prevazuta** se scrie un caz de utilizare care extinde unul dintre cazurile de utilizare descrise in faza de extragere a cerintelor.
- **Modul de tratare a acestor exceptii este decis in faza de proiectare de detaliu.**

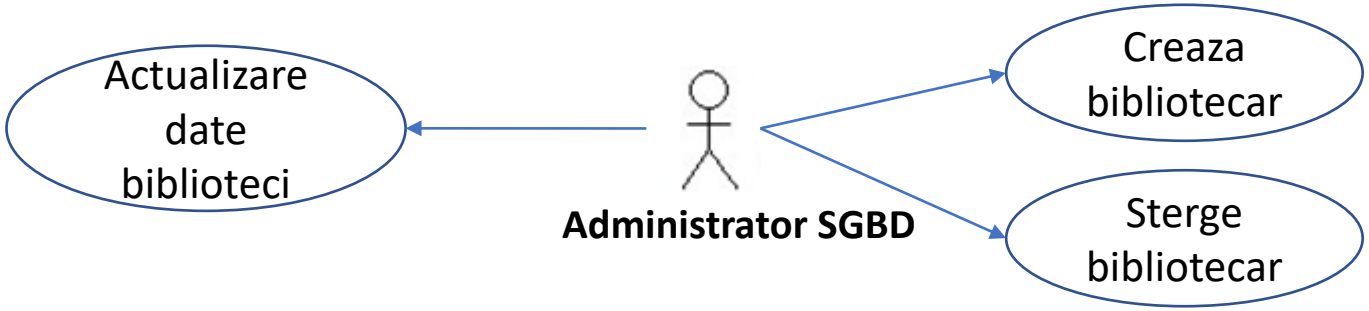


Cazuri de utilizare administrative

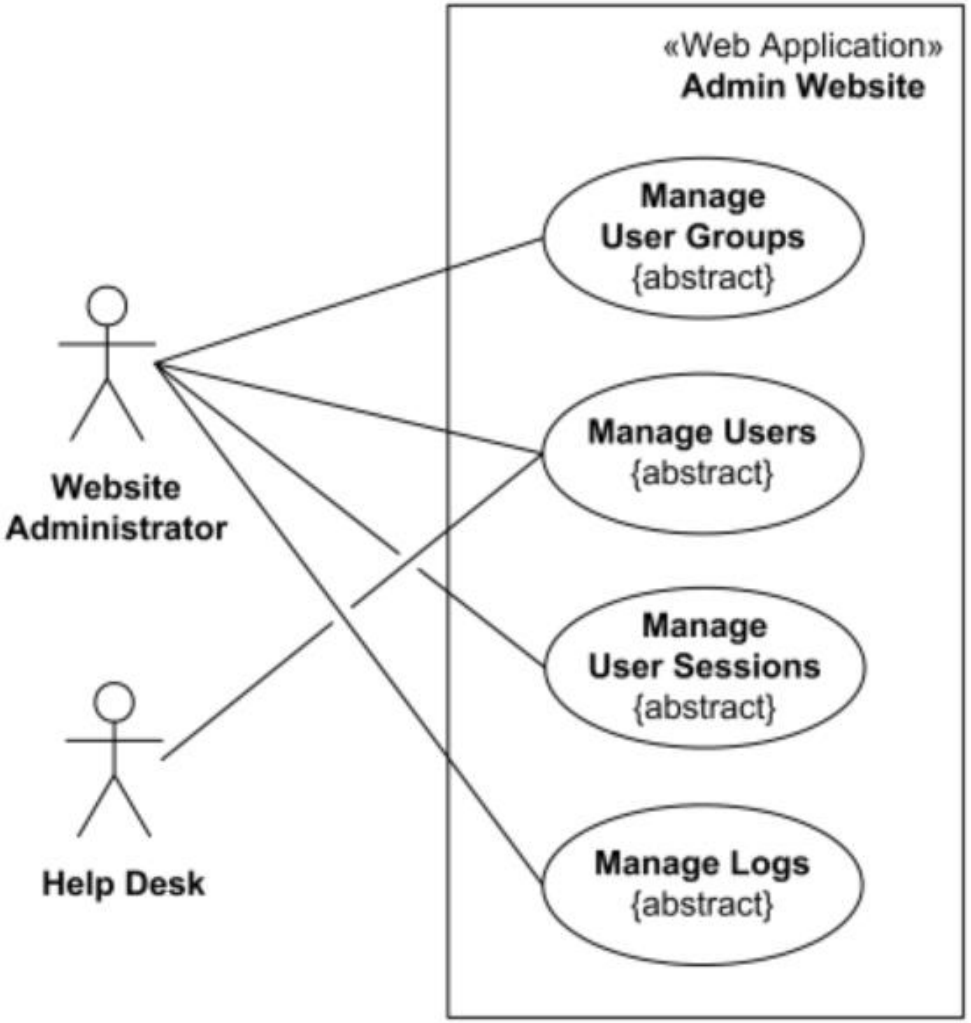
- Pentru operatii de administrare a sistemului, orice aplicatie ar trebui sa ofere **o interfata separata** de aceea care permite utilizatorilor accesul la functiile sistemului.
- De ex., **pentru o aplicatie web se recomanda** (OWASP Guide 2.0) **sa existe o aplicatie separata**, **dedicata sarcinilor administrative**, care sa poata fi accesata numai printr-un sistem de management de retea care asigura securitatea comunicarii. De ex, navigarea printr-o retea VPN (Virtual Private Network) este sigura deoarece traficul este criptat.

Exemplu – SGCB

Bibliotecarul gestioneaza cartile din biblioteci: adauga, sterge, modifica nr. de exemplare. Pentru accesul la sistem are nevoie de un cont creat de administratorul sistemului.



Exemplu: diagrama de cazuri de utilizare de nivel inalat pentru administrarea unui site web

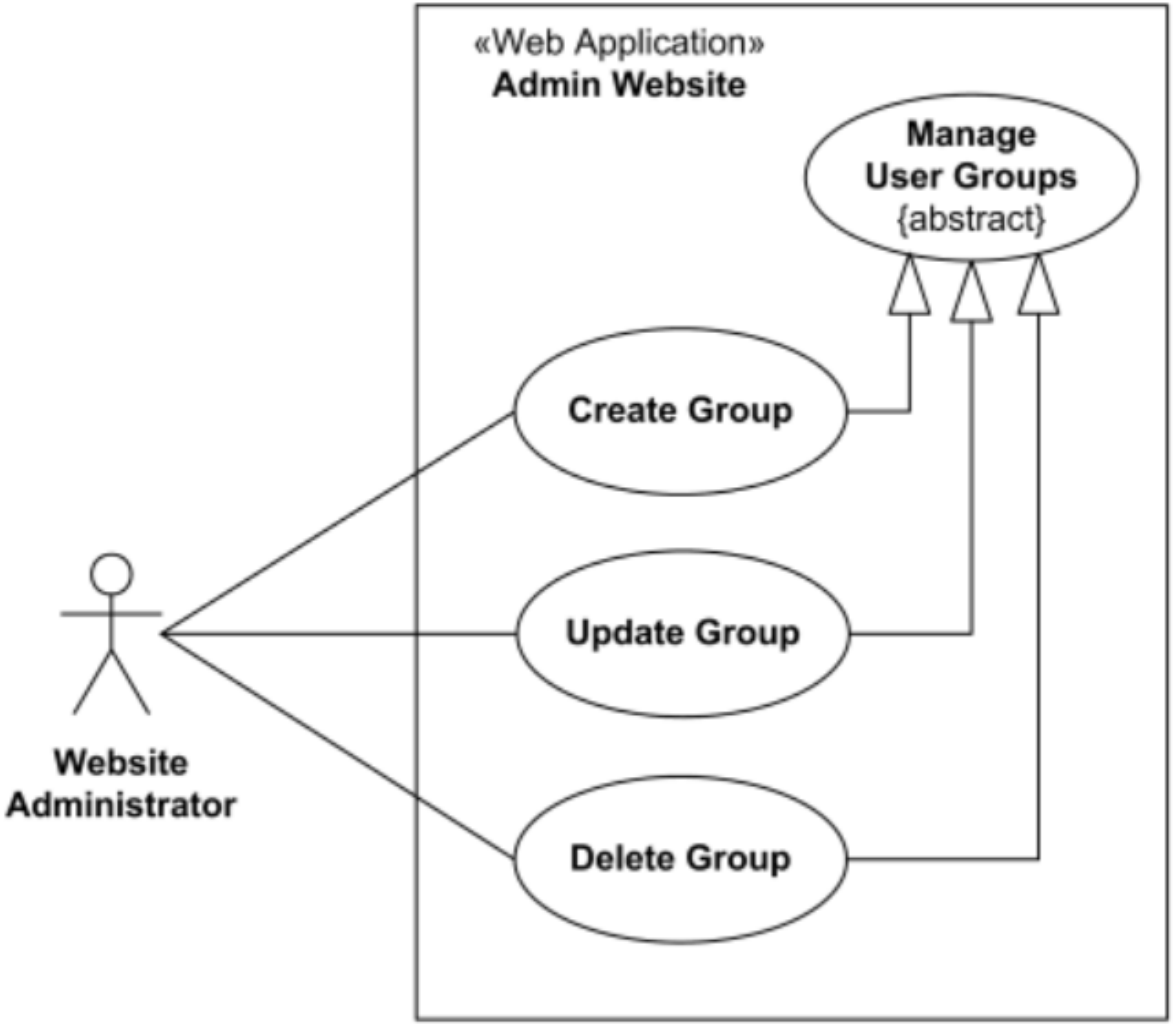


Caz de utilizare abstract:

- Neinstantiabil
- Poate fi un CU care:
  - Este inclus in alt CU
  - Extinde un alt CU
  - Generalizeaza alte CU

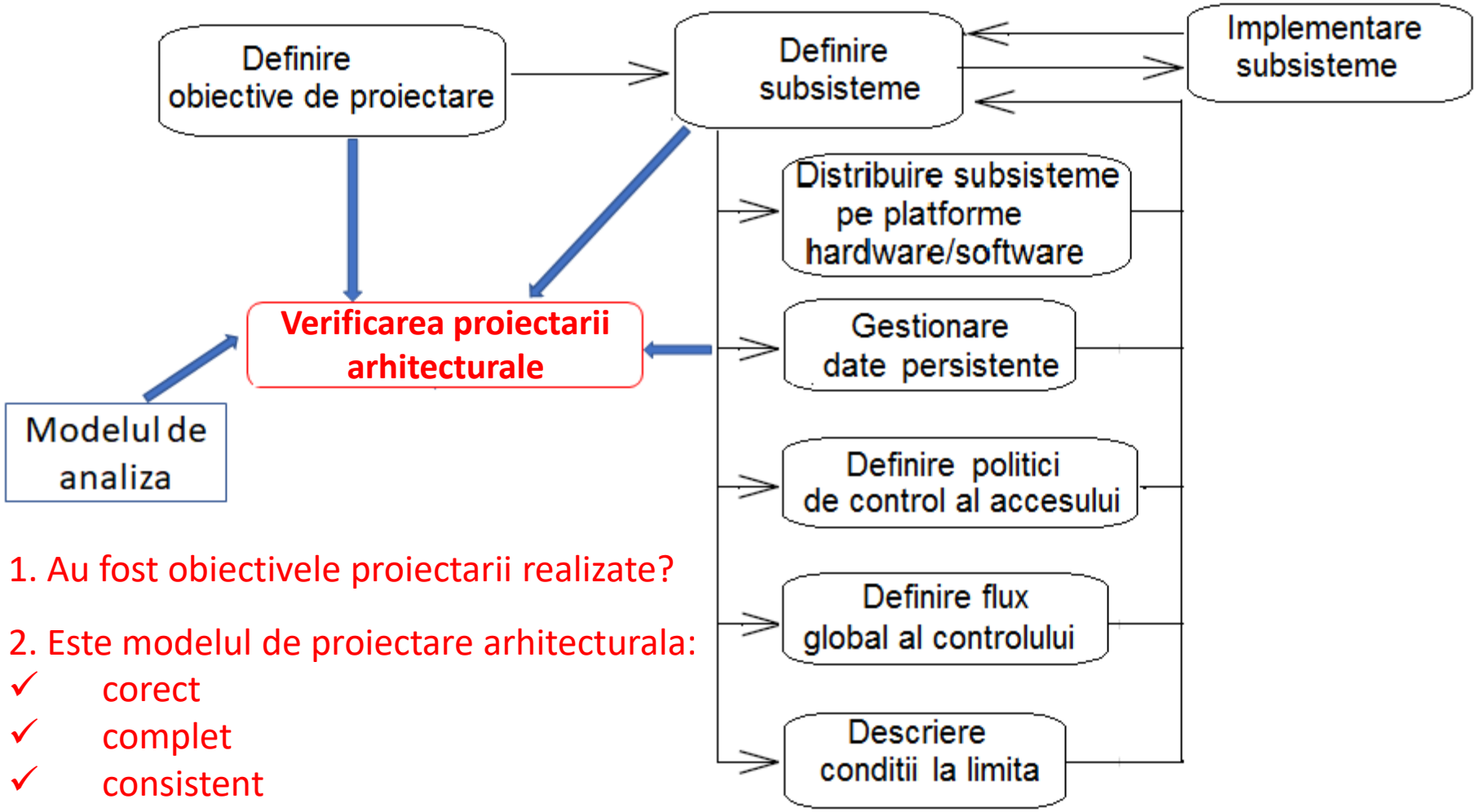
https://www.uml-diagrams.org/examples/website-admin-use-case-diagrams-example.html#google_vignette

Exemplu - continuare



https://www.uml-diagrams.org/examples/website-admin-use-case-diagrams-example.html#google_vignette

Verificarea proiectarii arhitecturale (1)



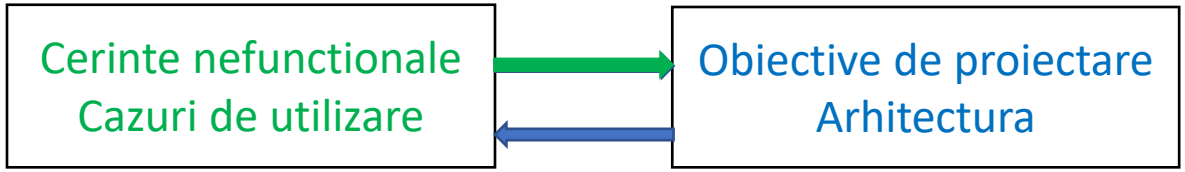
1. Au fost obiectivele proiectarii realizate?

2. Este modelul de proiectare arhitecturala:

- ✓ corect
- ✓ complet
- ✓ consistent
- ✓ realist
- ✓ usor de inteles

Verificarea proiectarii arhitecturale (2)

❖ **Modelul de proiectare este corect daca modelul de analiza se poate mapa pe modelul de proiectare si invers:**



— Poate fi mapat fiecare caz de utilizare pe un subsistem/set de subsisteme?

— Exista pentru fiecare subsistem un caz de utilizare sau o cerinta nefunctionala?

— Este fiecare cerinta nefunctionala adresata in modelul de proiectare?

— Poate fi mapat fiecare obiectiv de proiectare pe o cerinta nefunctionala?

— Exista pentru fiecare actor o politica de acces?

— Este fiecare politica de acces consistenta cu cerinta de securitate?

Verificarea proiectarii arhitecturale (3)

❖ **Modelul este complet daca toate cerintele au fost adresate in modelul de proiectare:**

— Exista functionalitati in cazurile de utilizare care nu se regasesc in subsistemele proiectate?

— Sunt prevazute conditiile limita?

❖ **Modelul este consistent daca nu exista contradictii:**

- Sunt prioritizate obiectivele de proiectare conflictuale?
- Exista obiective de proiectare in contradictie cu cerinte nefunctionale?
- Exista mai multe subsisteme sau clase cu acelasi nume?

❖ **Modelul este realist daca poate fi implementat**

❖ **Modelul este usor de inteles daca dezvoltatorii neimplicati in proiectarea de sistem înțeleg modelul.**

Lecturi suplimentare – proiectarea arhitecturală

1. Ian Sommerville, Software Engineering, Tenth Edition, Pearson Education Limited, 2015, ISBN 13: 978-1-292-09613-1
2. https://www.slideshare.net/software-engineering-book/ch6-architectural-design
3. http://software-engineering-book.com/web/archpatterns/
4. Mark Richards, Software Architecture Patterns, O'Reilly Media, Inc, 2015, ISBN: 9781491971437
5. Rick Kazman, Paul Clements, Len Bass, Software Architecture in Practice, Third Edition, Addison-Wesley Professional, 2012, ISBN: 9780132942799
6. MVC Tutorial for Beginners: What is, Architecture & Example: https://www.guru99.com/mvc-tutorial.html
7. https://www.uml-diagrams.org/examples/website-admin-use-case-diagrams-example.html#google_vignette

Diagrame de activitate

Prof. univ. dr. ing. Florica Moldoveanu

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

Diagrame de activitate: utilizari (1)

- Se folosesc pentru modelarea aspectelor dinamice ale unui sistem, la diferite niveluri:
  - la nivelul „business process” (modelarea domeniului aplicatiei) – in specificarea cerintelor;
  - pentru reprezentarea fluxului global al controlului – in proiectarea arhitecturala;
  - pentru modelarea unei operatiilor complexe din clase, a proceselor si firelor de executie paralele – in proiectarea de detaliu.
- O diagrama de activitate poate reda un flux de lucru, pasii unui proces de calcul, executia secventiala sau paralela a unor actiuni.
- O diagramă de activitate **redă o activitate descompusă în acţiuni** care se pot executa secvenţial sau în paralel.

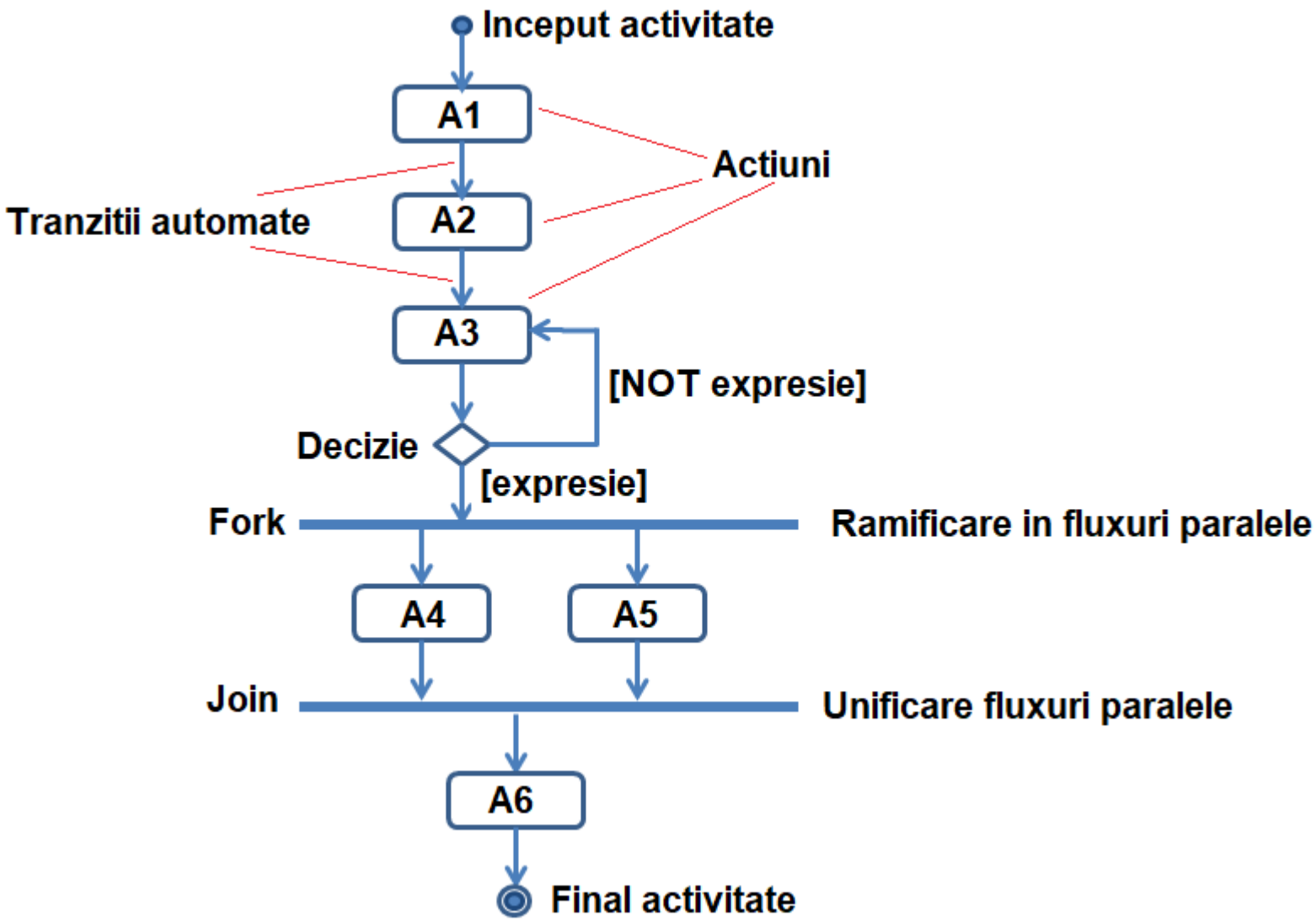
Actiuni şi tranziții

- O diagramă de activitate este un graf orientat în care **nodurile corespund acţiunilor** iar **tranzițiile indică ordinea în care acestea se execută.**

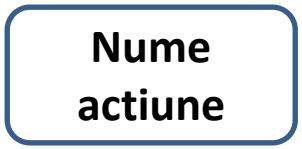
- O ***actiune*** reprezinta un singur pas intr-o activitate: un calcul, gasirea unor date, verificarea unor date, trimiterea unui semnal, etc.
- **Fluxul controlului** (trecerea de la o actiune la urmatoarea) este reprezentat prin **tranzitii.**

- 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: noduri şi tranzitii

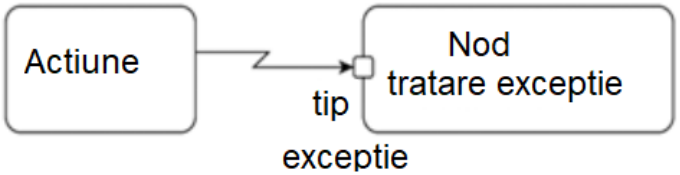


Noduri actiune (1)



Nod actiune general – poate repezenta orice operatie care reprezinta un pas în activitatea redată în diagramă: un calcul, o actiune de comunicare, etc.

Nod Tratare exceptie



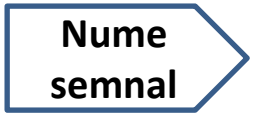
Tratarea cazurilor de exceptie la executia unei actiuni; nodul primeşte ca intrare tipul excepției

Noduri actiune cu specificatie speciala au reprezentari grafice particulare:

- **noduri semnal**
- **nodul aşteapta eveniment timp**

Noduri actiune (2)

Noduri semnal

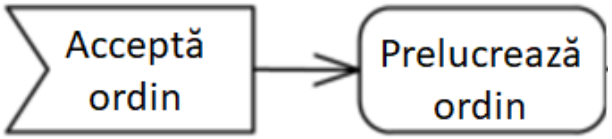


Trimite semnal - actiune care genereaza un semnal specificat

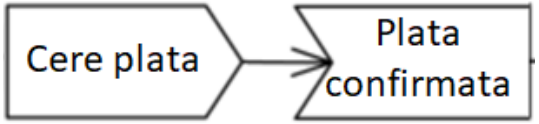
Acceptă semnal - actiune care se executa atunci cand se primeste semnalul specificat

- Nodurile **Send/Receive signal** reprezinta modul în care o activitate poate fi modificată din exteriorul sistemului.

- Daca actiunea “Accepta semnal” nu o sageata de intrare, ea este pornita odata cu activitatea din care face parte.



- Daca actiunea “Accepta semnal” are sageata de intrare, ea se declanseaza numai dupa terminarea actiunii anterioare.



Noduri actiune (3)

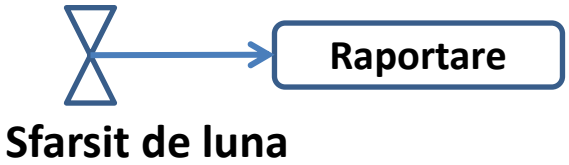
Noduri semnal

- La acceptarea semnalului “Acceptă ordin” este invocată actiunea “Prelucrează ordin”.
- La terminarea actiunii “Prelucrează ordin” este invocata actiunea “Cere plata”, care genereaza semnalul “Cere plata”.
- Fluxul controlului este oprit până când se primeste semnalul “Plata confirmata”.
- Acceptarea semnalului “Plata confirmata” este permisa numai dupa ce actiunea “Cere plata” s-a terminat (a fost generat semnalul “Cere plata”).
- La terminarea actiunii “Plata confirmata” este invocate actiunea “Transport”.

Noduri actiune (4)

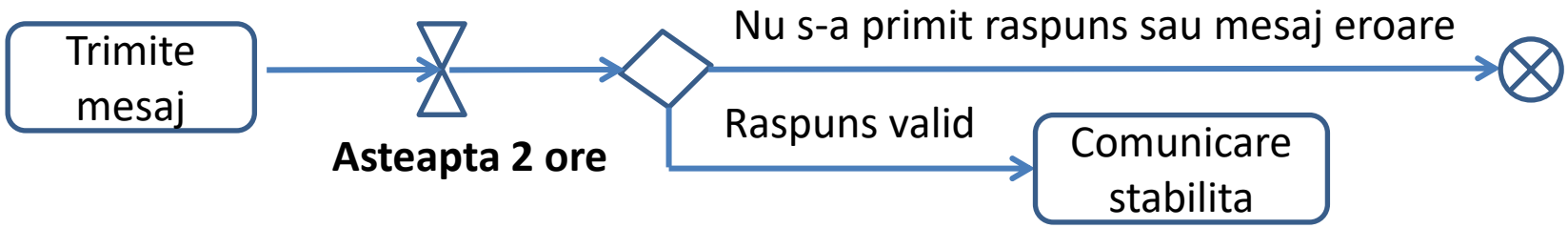
Nod aşteapta eveniment timp (Wait time action) – genereaza o iesire la terminarea unei perioade de timp

- Daca nodul nu are tranzitii de intrare: actiunea sa are loc cât timp se execută activitatea din care face parte.



Fluxul controlului este oprit până la sfârsitul lunii, cand se trece la actiunea Raportare.

- Daca nodul are tranzitii de intrare: actiunea sa este declansata la terminarea actiunii anterioare.



Noduri de control (1)

Nodul de început activitate (initial)

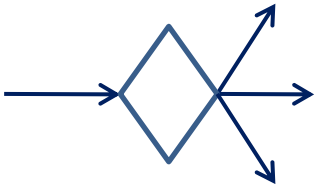
Nodul final activitate - desemneaza sfârşitul tuturor fluxurilor controlului din activitatea redată în diagramă

Nod final de flux - desemneaza sfârşitul unui singur flux al controlului din activitate

Cale de tip " flux control"

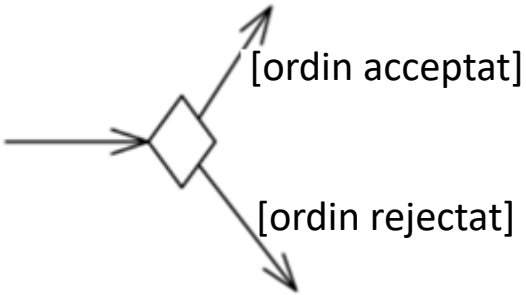
Nodul "final de flux control"

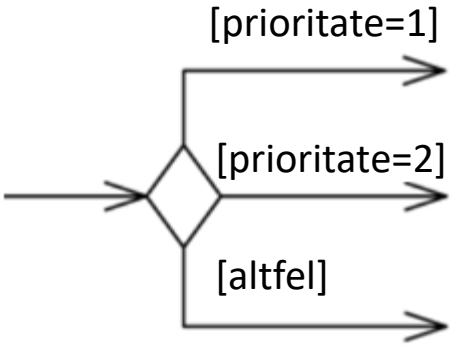
Noduri de control (2)



Nod decizie

- Reprezinta ramificarea unui flux de date sau flux al controlului.
- Tranzițiile dintr-un nod de decizie sunt controlate prin gărzi.
- Este posibilă o singură tranziție.

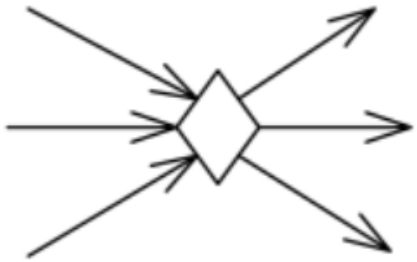




Noduri de control (3)

Nod unificare (merge)

- Uneşte mai multe fluxuri de date sau fluxuri ale controlului într-unul singur.
- Este asociat cu unul sau mai multe noduri de decizie.

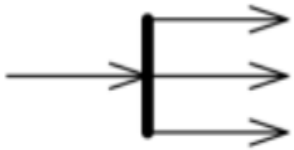


Nod de unificare combinat cu un nod de decizie.

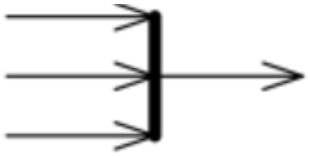
- Tranzițiile de iesire sunt controlate prin gărzi.
- Este posibilă o singură tranziție.

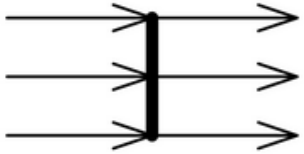
Noduri de control (4)

Nod Fork – ramifică fluxul de intrare în mai multe fluxuri concurente



Nod Join – sincronizează fluxurile de intrare concurente

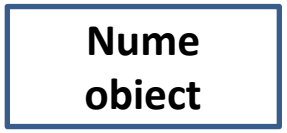




Noduri Join şi Fork combinate într-unul singur.

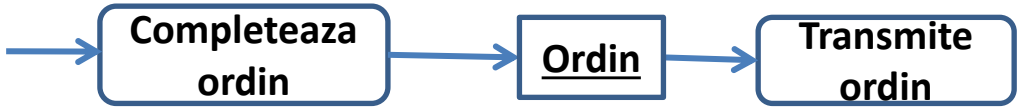
Noduri obiect şi flux obiect (flux de date) -1

Noduri obiect



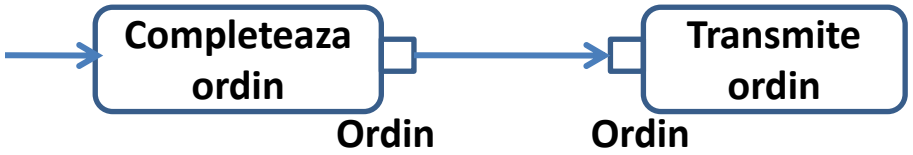
Reprezinta datele de intrare pentru o actiune sau datele generate de o actinue

Exemplu



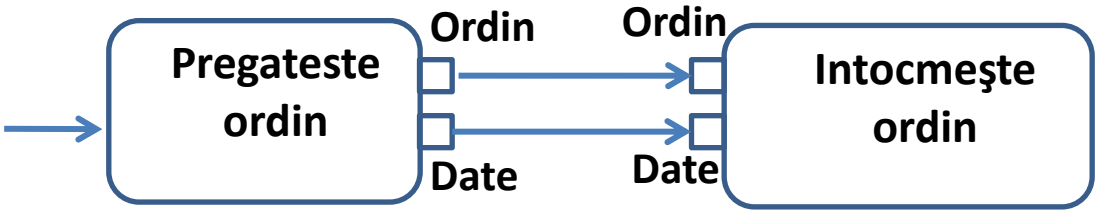
“Pini” de intrare/iesire actiune: noduri obiect prin care se reprezinta datele de intrare/iesire ale unei actiuni.

Exemplu



Este echivalenta cu reprezentarea de mai sus.

Noduri obiect şi flux obiect -2



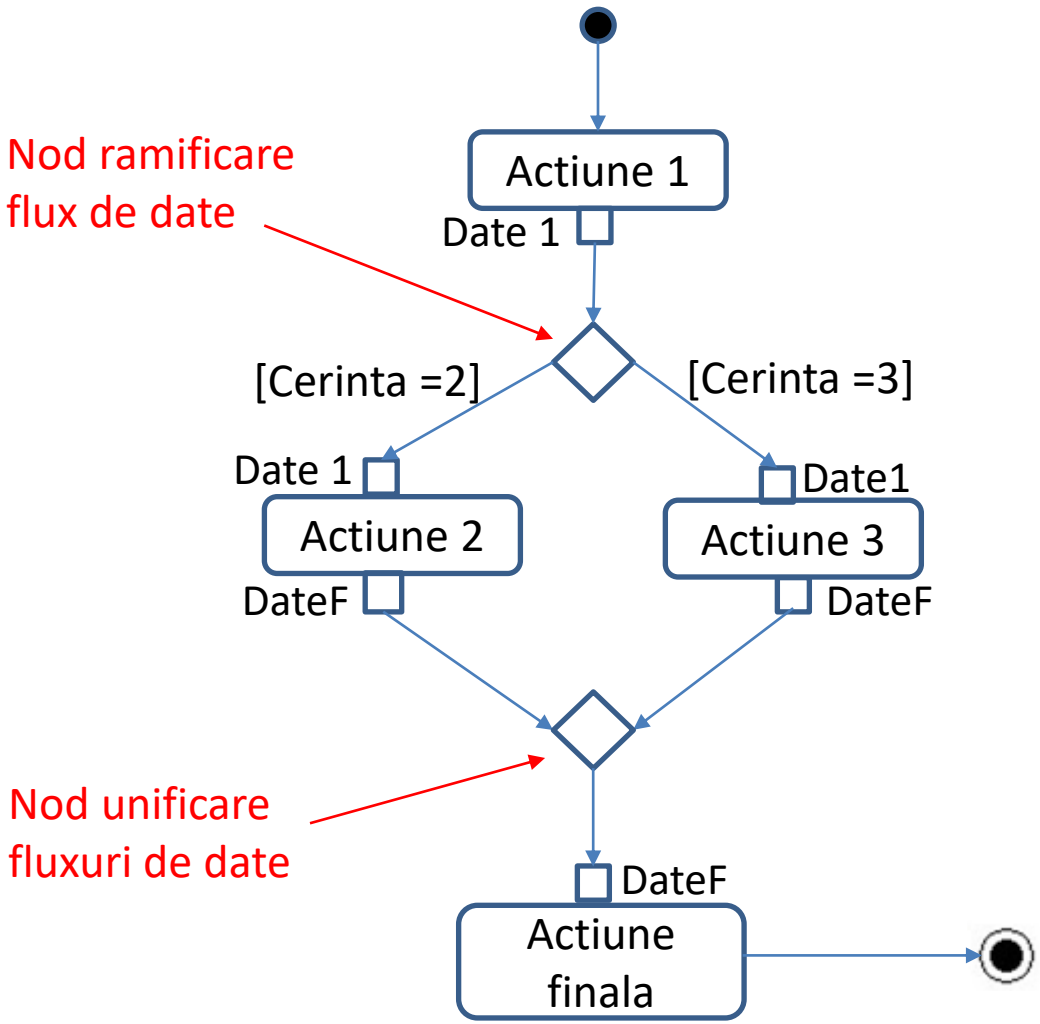
“Pini” de intrare/iesire actiune

Nod “data store” – reprezinta un buffer central pentru date permanente.

Exemplu:

- Se pot stabili reguli privind înlocuirea/stergerea/adăugarea datelor de intrare în nodul “data store”.

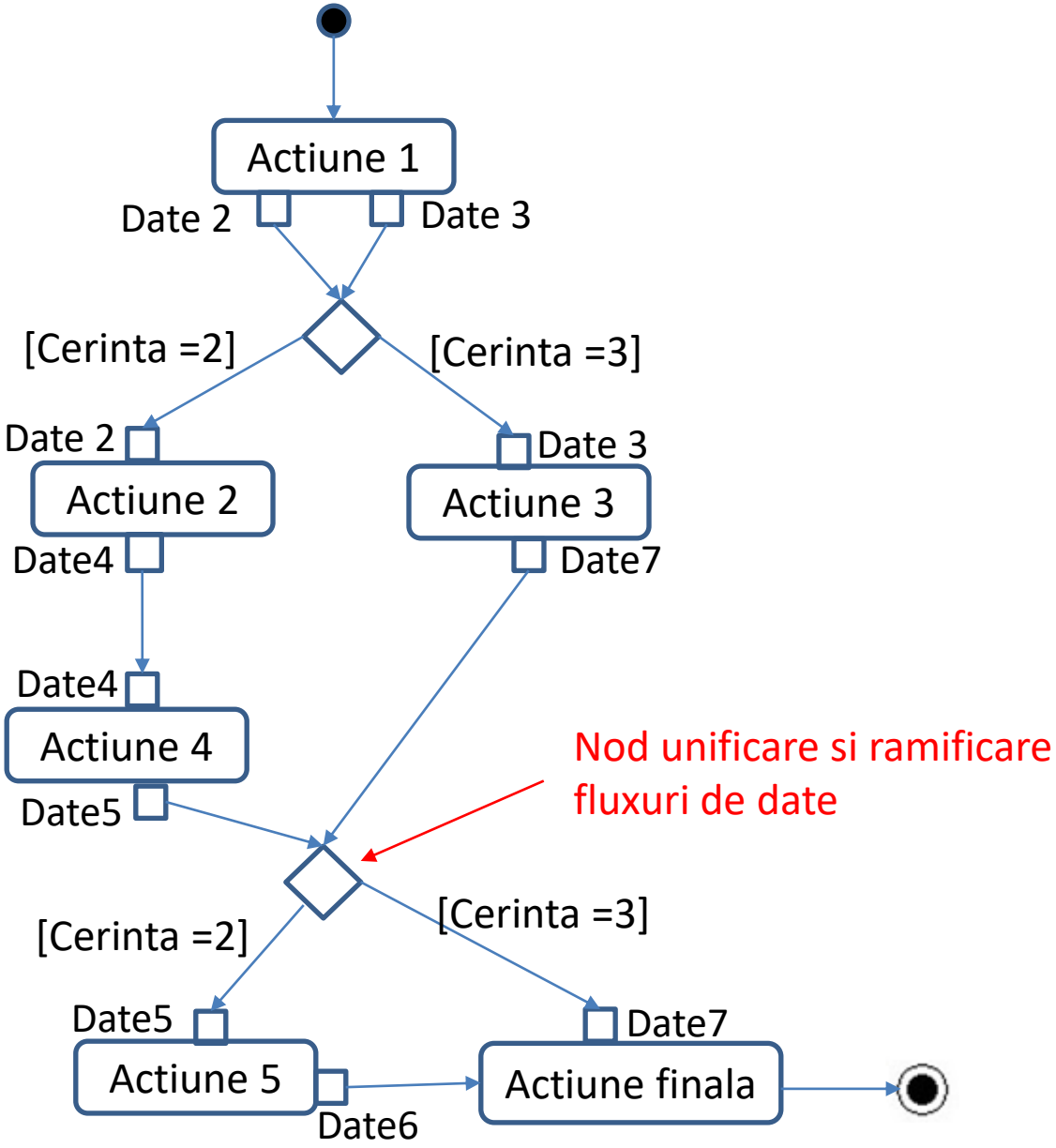
Exemplu flux obiect



Exemplu:

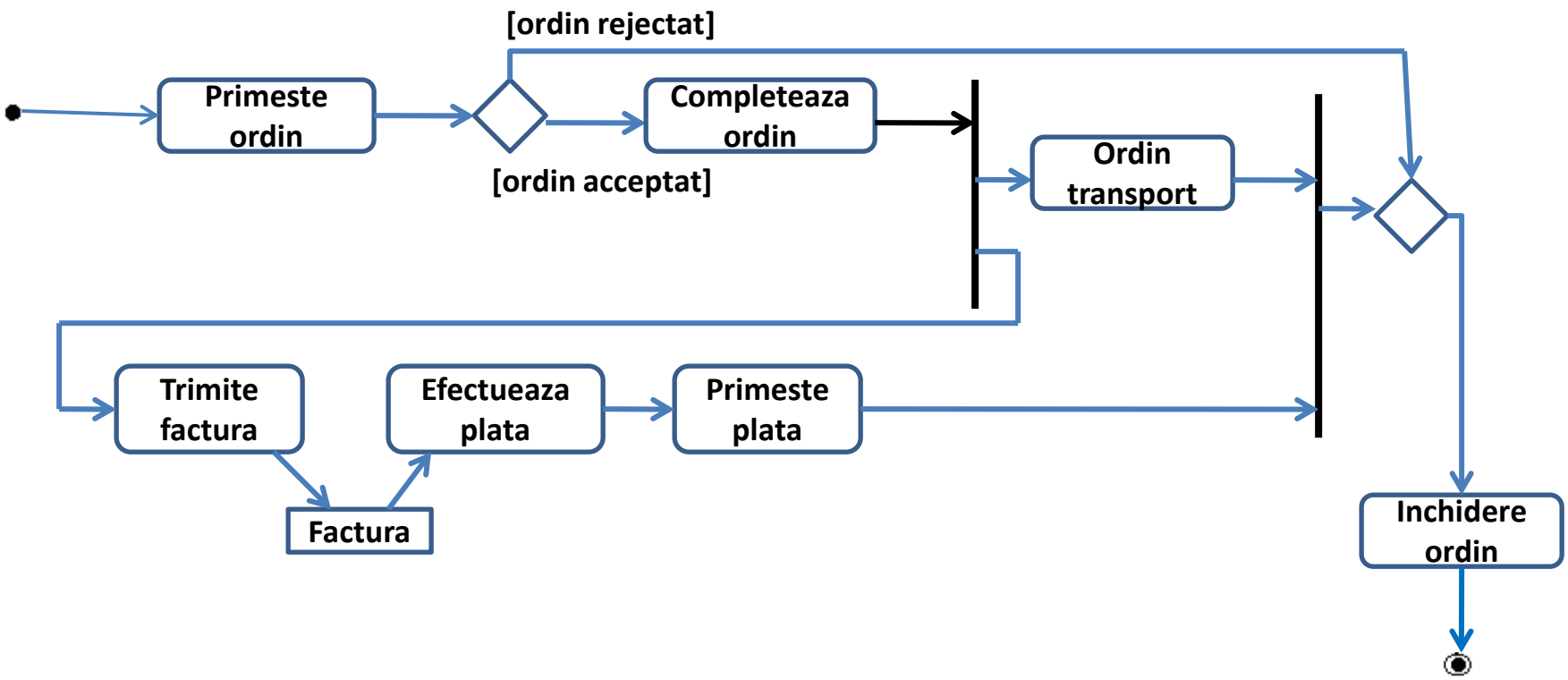
- Actiune 2 si Actiune 3 produc o imagine (DateF) in mod diferit:
  - o imagine fara umbre
  - o imagine cu umbre
- Actiune finala poate fi aceea de afisare a imaginii

Exemplu flux obiect



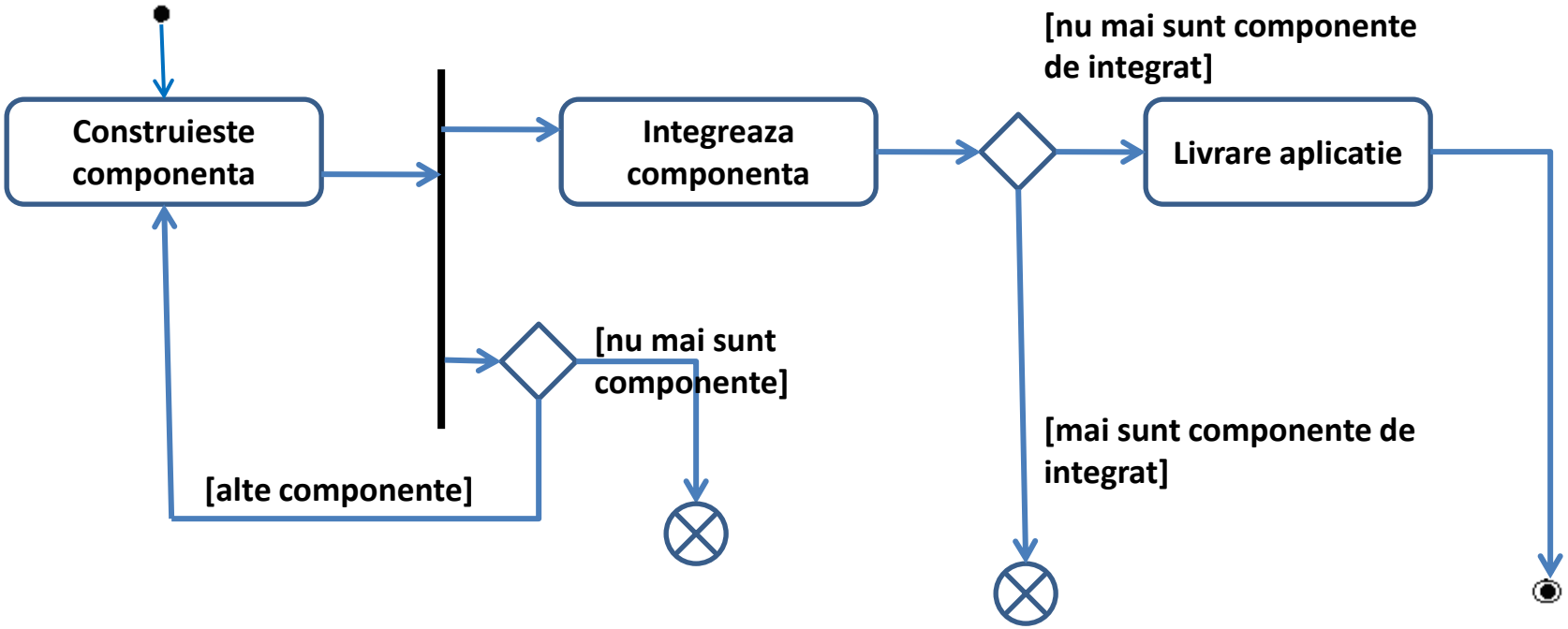
Diagrame de activitate: exemple(1)

Tratarea unui ordin

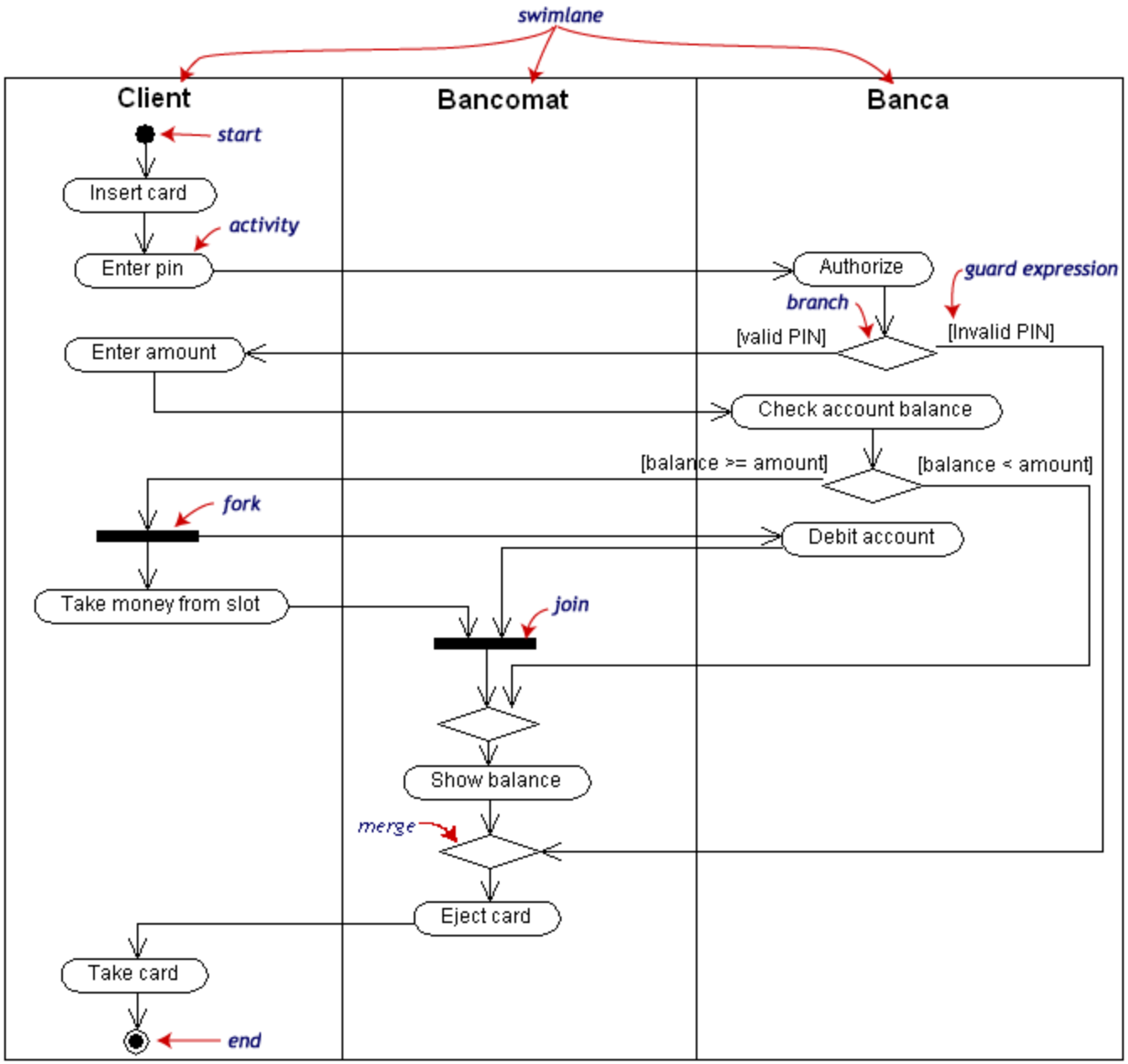


Diagrame de activitate: exemple(2)

Dezvoltare software alcatuit din componente



Diagramă de activitate partiţionată



- Actiunile pot fi aliniate pe “culoare” verticale sau orizontale distincte.
- Fiecare culoar corespunde unui obiect participant in activitate

- Corespondenta cu diagrama de secventa echivalenta

- In plus, diagrama de activitate poate reda fluxuri paralele

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 global al controlului in sistem
  - 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).
- Pornind de la o diagrama de activitate poate fi generat automat cod sursa („forward engineering”), atunci cand diagrama reprezinta o operatie.
- De asemenea, este posibila generarea diagramei de activitate pornind de la cod sursa („ingineria inversa” - „reverse engineering”).

Lecturi suplimentare

https://www.uml-diagrams.org/activity-diagrams-reference.html

https://sparxsystems.com/enterprise_architect_user_guide/15.2/model_domains/objectflowactivitydiagram.html

https://www.smartdraw.com/activity-diagram/#activityDiagramSymbols

Diagrame pentru managementul modelelor (Packages)

Prof. univ. dr. ing. Florica Moldoveanu

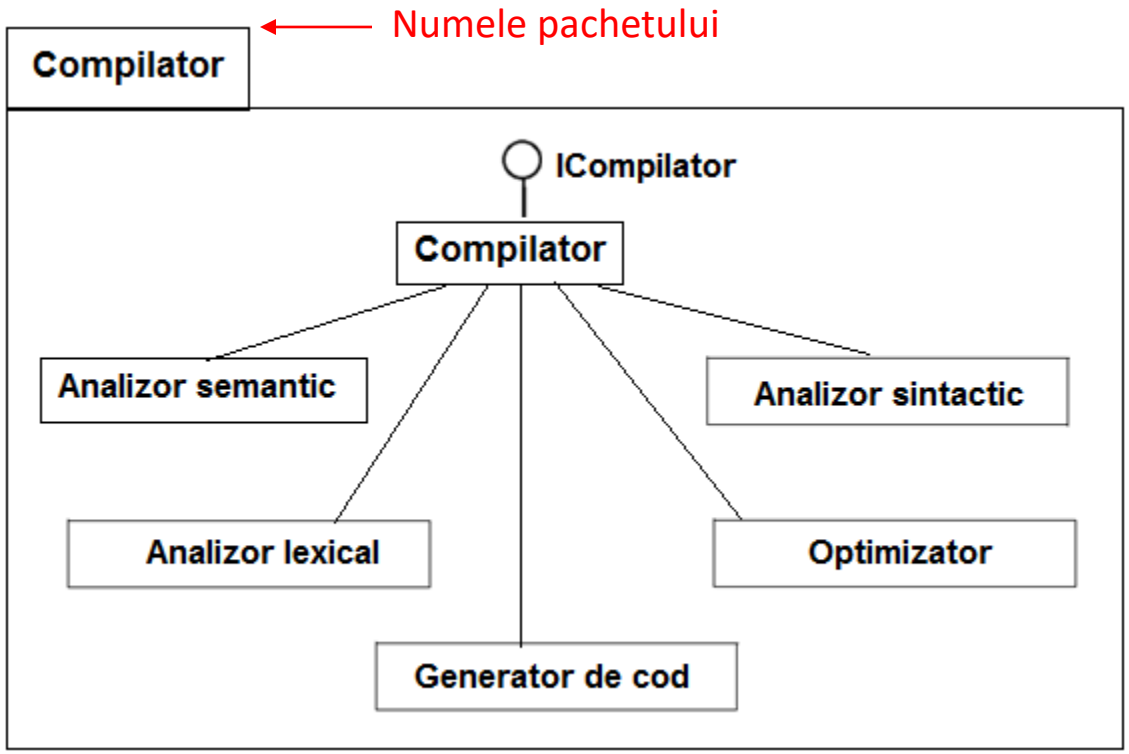
Curs *Ingineria programelor* – UPB, Automatică şi Calculatoare
2020-2021

Pachete - utilizari

- Se folosesc pentru organizarea artefactelor rezultate în 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 în pachete printr-un arbore similar cu cel afisat de Windows Explorer.
- Pachetele sunt containere. Daca se şterge sau muta un pachet, toate elementele din interiorul său sunt şterse/mutate.

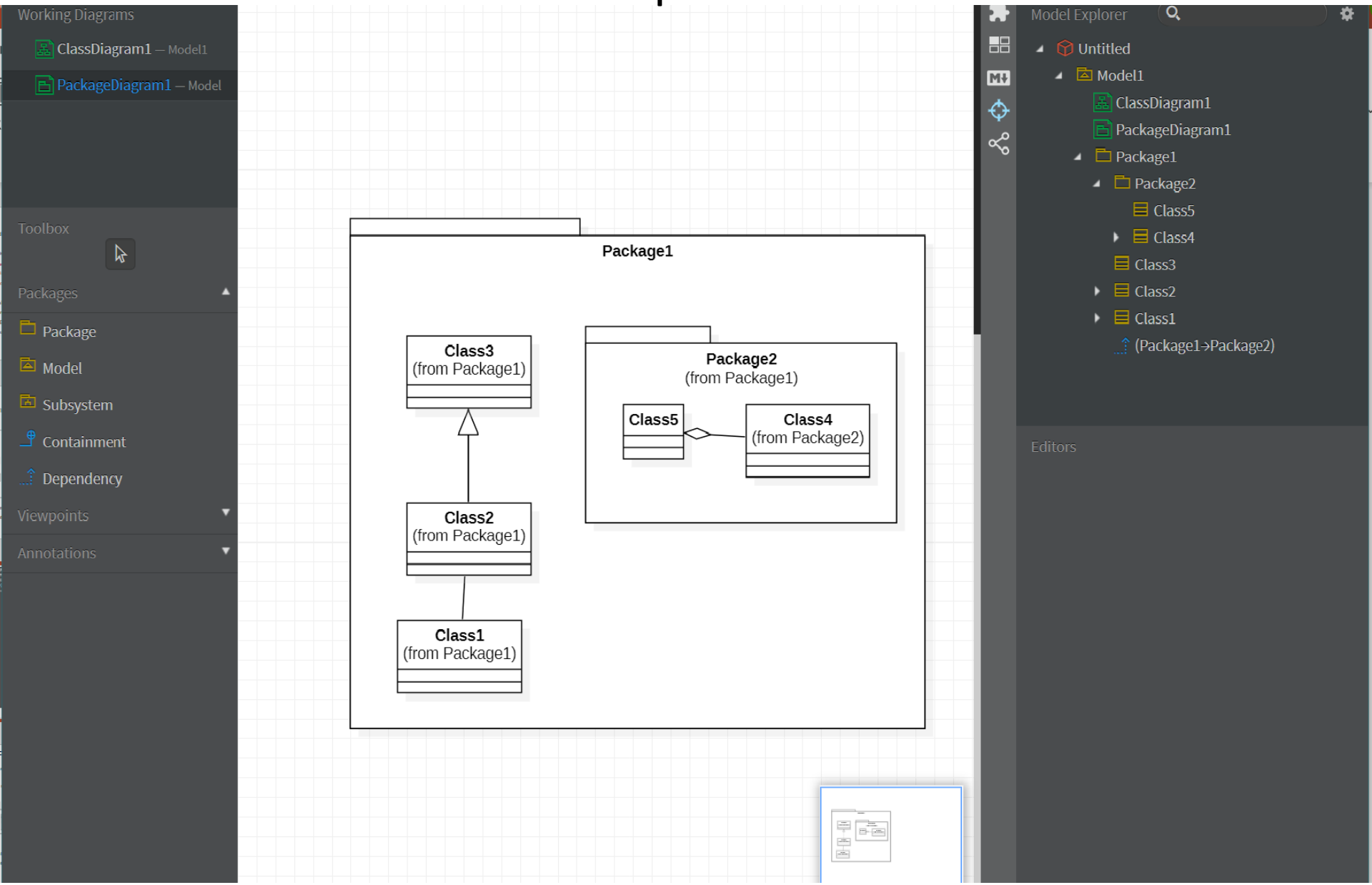
Exemplu de pachet UML

Gruparea elementelor de modelare ale subsistemului Compilator, într-un pachet UML.



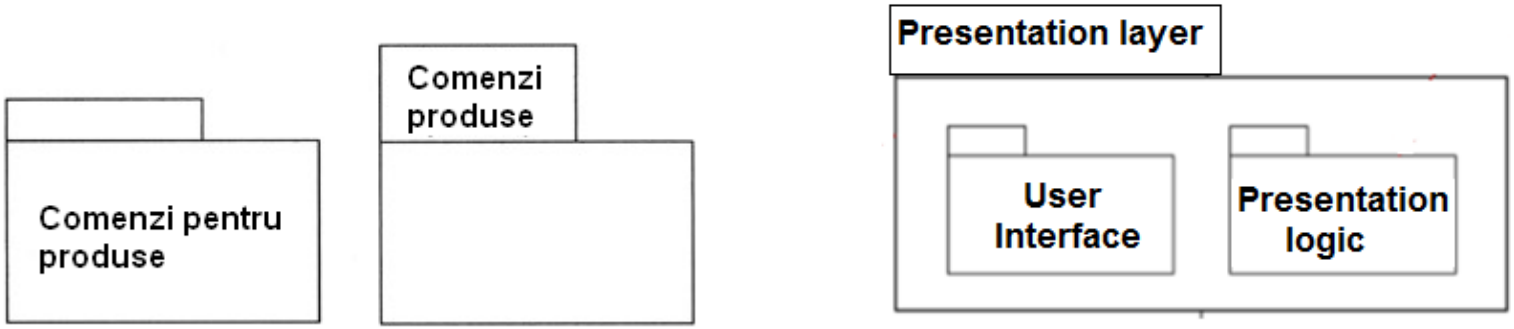
- Pachetul cuprinde: clasele subsistemului, clasa “fațadă” Compilator şi interfața **Icompilator**.
- Pot fi adaugate: cazurile de utilizare implementate de subsistem, diagrame de activitate, s.a

Ierarhie de pachete



Pachete – reprezentare grafica

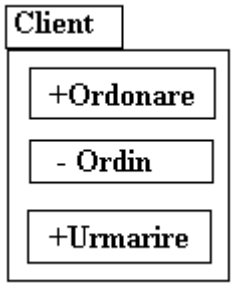
- Un pachet este reprezentat printr-un dreptunghi cu o mica “toartă” ataşată dreptunghiului in partea de stanga sus.
- Numele pachetului poate fi amplasat în centrul dreptunghiului sau pe toartă:



- Un pachet are asociat un „spatiu de nume” (namespace): elementele amplasate în acelasi pachet trebuie sa aiba nume unice; elemente din pachete diferite pot avea acelasi nume.

Vizibilitatea elementelor dintr-un pachet

- 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

- Un pachet poate **importa** fie membrii individuali ai altor pachete fie alte pachete în întregime.
- **Spatiul de nume al pachetului care importa este extins cu cel al pachetului important.**

➢ Elementele pachetului Prezentare pot fi referite in pachetul Aplicatie Web folosind nume necalificate.

Relatii intre pachete (1)
Extindere şi import

- Un pachet poate fi extins cu un alt pachet.
- Relaţiile de import şi extindere (merge) sunt relaţii de dependenţă între pachete.

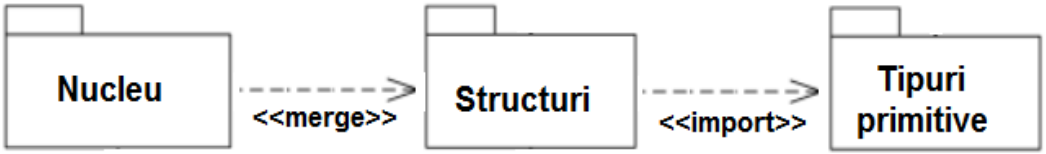
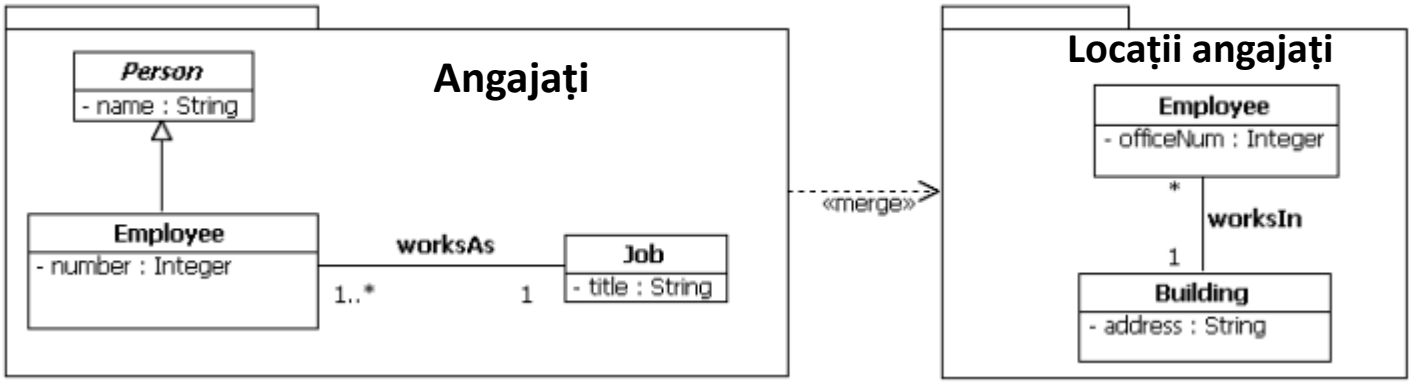


Diagrama de pachete

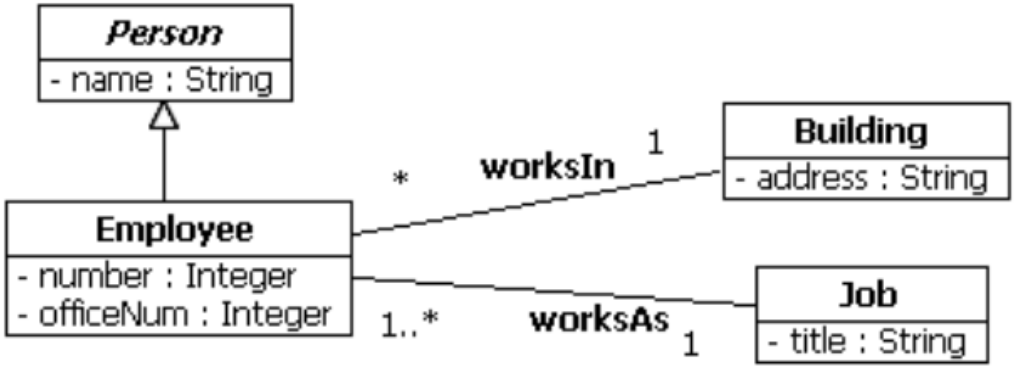
- Pachetul „Nucleu” este extins (combinat) cu pachetul „Structuri”.
- Pachetul “Tipuri primitive” este importat de pachetul “Structuri”, ceea ce permite referirea la elementele pachetului “Tipuri primitive” din pachetul “Structuri” utilizând nume necalificate: spaţiul de nume al pachetului “Tipuri primitive”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

Relatii intre pachete (2)

Extindere



- Elementele din **Locații angajați** care au acelasi nume cu elemente din **Angajați** sunt combinate (cu attribute si relatii) iar cele care nu se regasesc in pachetul **Angajați** sunt copiate în pachetul rezultat.



Continutul pachetului **Angajați** extins cu **Locatii angajati**

Relatii intre pachete (3)

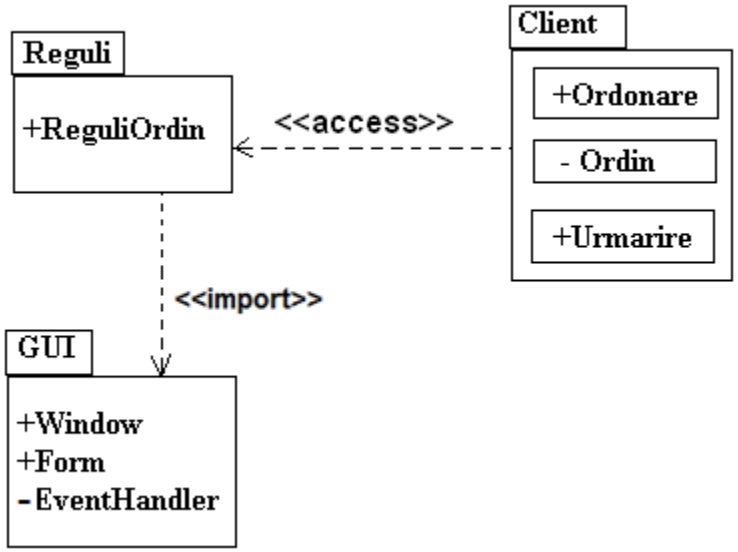
Import public şi privat

- 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;

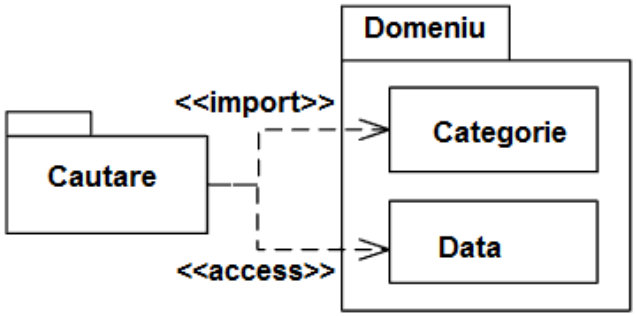
  - privat - elementele pachetului importat sunt accesibile numai în pachetul care-l importă.

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



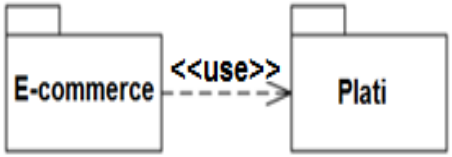
Relatii intre pachete (4)
Import membri individuali

- **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



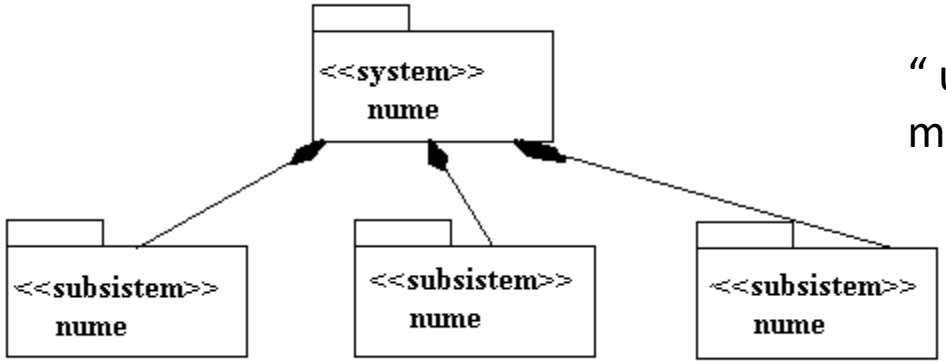
Relaţia <<use>> nu precizează cum elementul client (E-commerce) utilizează furnizorul (Plati). Astfel, pachetul „Plăţi” ar putea fi utilizat în definiţia sau în implementarea pachetului „E-commerce”.

Sisteme şi Modele(1)

- Un sistem este o grupare de elemente organizată pentru realizarea unui scop.
- Sistemele pot fi descompuse în subsisteme separate, fiecare subsistem putand fi vazut la randul sau ca un sistem, la un nivel de detaliu mai coborat.
- Subsistemele sunt grupări de elemente care pot fi dezvoltate relativ independent. Descompunerea are loc în etapa de proiectare arhitecturală, când se decide cum vor fi realizate cerințele.
- Relațiile sistem-subsisteme pot fi reprezentate:
  - In UML 1.x, printr-o diagramă de pachete
  - In UML 2.x, printr-o diagramă de componente

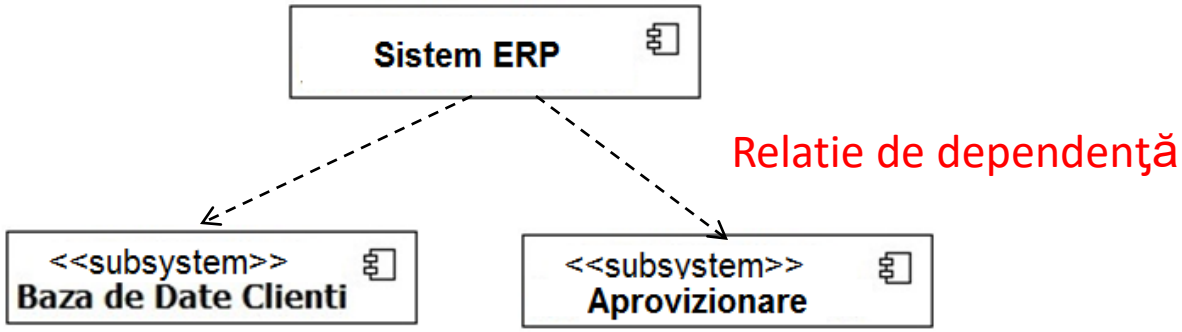
Sisteme şi Modele (2)

- 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"

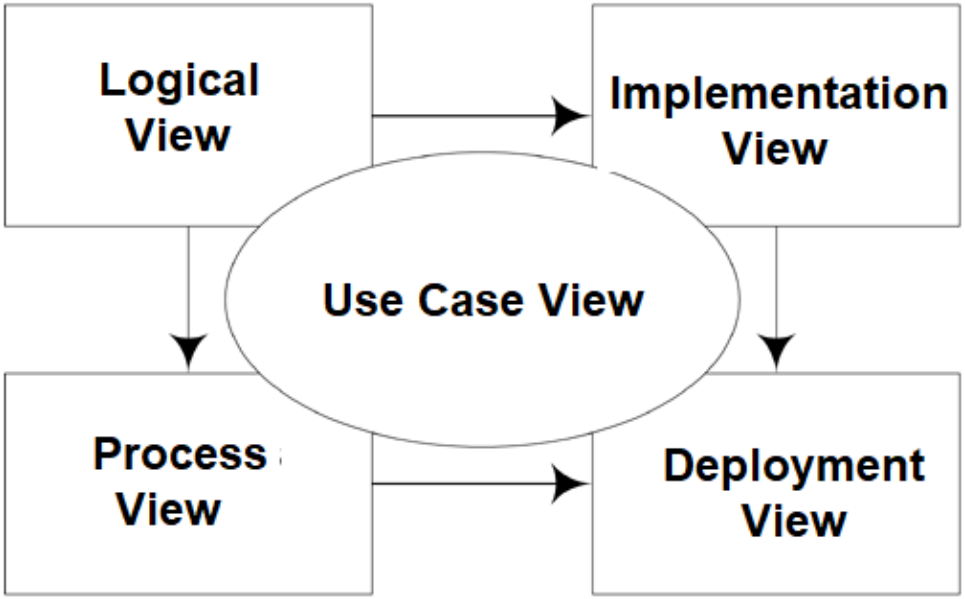
- In UML 2.x, un subsistem este un tip particular de componentă



Sisteme şi Modele (3)

- Modelul unui sistem este o simplificare a realității, o abstracție a sistemului, creata pentru a înțelege mai bine sistemul şi a-l specifica.
- O **vedere** este o proiecție a unui sistem dintr-un punct de vedere în care sunt omise aspectele nerelevante din punctul de vedere respectiv.

- Exemplu: cele 5 vederi ale unei arhitecturi software



Sisteme şi Modele (4)

Use Case View cuprinde:

- Cazurile de utilizare care descriu comportarea sistemului vazută de utilizatori, analisti şi testeri
- Diagrame de cazuri de utilizare
- Diagrame de interacțiune, de stări şi de activități

Logical View: vedere externă asupra sistemului, independentă de implementare

Cuprinde:

- Clasele si diagramele de clase din modelul de analiza - pentru modelarea aspectelor statice
- Diagrame de interactiune, de stari si de activitate - pentru modelarea aspectelor dinamice

Sisteme şi Modele (5)

Process View: procesele si firele de executie care formeaza mecanismele de concurență si de sincronizare ale sistemului.

- Diagrame de clase ale obiectelor care participa la fire si procese de executie.
- Diagrame de interactiune, de activitate si de stări

Implementation View

Descrie componentele care alcatuiesc sistemul fizic final. Se folosesc:

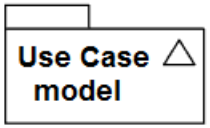
- Diagrame de clase si de componente pentru a modela aspectele statice
- Diagrame de interactiune, de stari si activitate, pentru modelarea aspectelor dinamice.

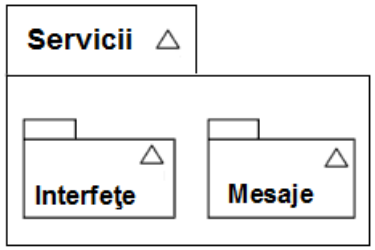
Deployment View: nodurile care formeaza topologia hardware a sistemului si repartitia elementelor software fizice pe noduri. Se folosesc:

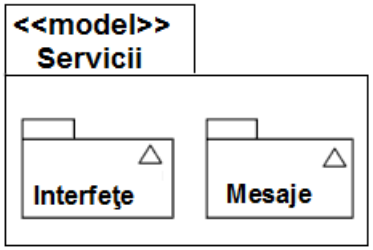
- Diagrame de distributie, pentru modelarea aspectelor statice
- Diagrame de interactiune, de stari si de activitate, pentru modelarea aspectelor dinamice.

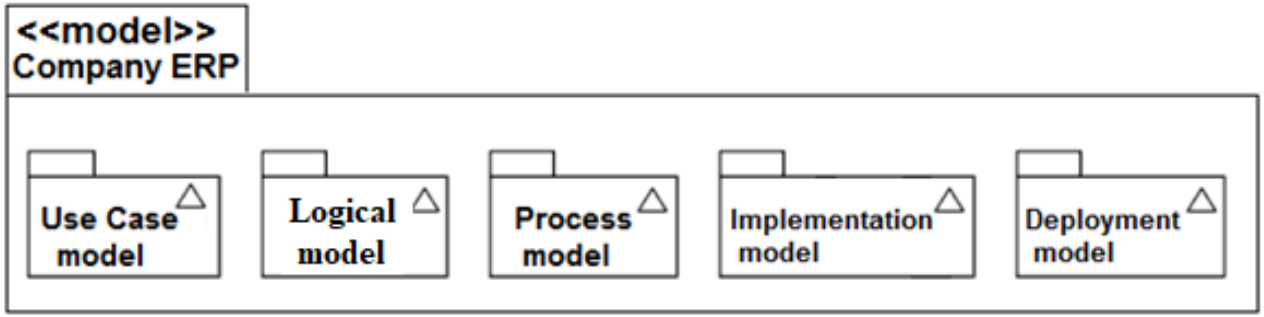
Sisteme şi Modele (6)

- **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:









Lecturi suplimentare - pachete

https://www.uml-diagrams.org/package-diagrams.html

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

https://www.uml-diagrams.org/package-diagrams-reference.html

https://www.win.tue.nl/~aserebre/2IW80/2013-2014/05%20-%20UML%20Structural%20Diagrams%20Other%20Diagrams.pdf

Modelarea interfetelor în UML

Prof. univ. dr. ing. Florica Moldoveanu

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

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, constructorul, distructorul si atributele.

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

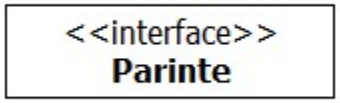
→ clientii clasei sunt fortati sa foloseasca numai interfata clasei

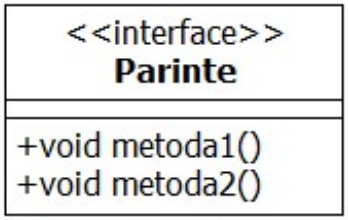
→ modificarea implementarii interfetei nu afecteaza clientii

Reprezentarea interfețelor in UML

❖ O interfață poate fi reprezentată într-o diagramă de clase sau într-o diagramă de componente.

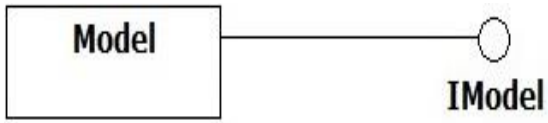
1. Reprezentare asemănătoare cu a unei clase:





Interfeţe (2)

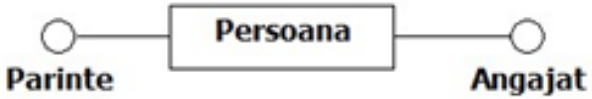
2. Reprezentare printr-un cerc adnotat cu numele interfeței, notatie numita *lollipop* (acadea):



Clasa *Model* implementează interfața *IModel*.

O clasa poate sa implementeze mai multe interfeţe:

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



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

→ **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**

❖ In diagramele de clase conceptuale interfetele furnizate de o clasa se reprezinta prin nume de rol atasate asocierilor clasei cu alte clase.

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
{.....................................}
```

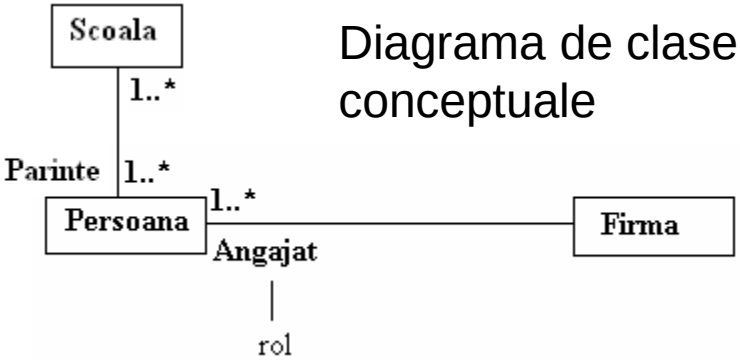
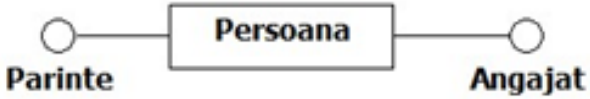
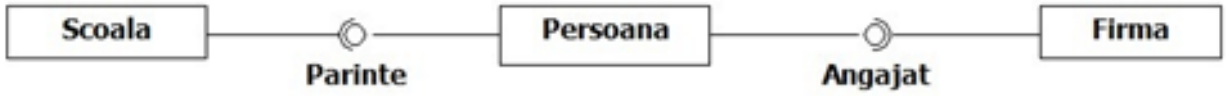


Diagrama de clase conceptuale

Interfeţele apar ca nume de rol in asocierea dintre 2 clase.



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



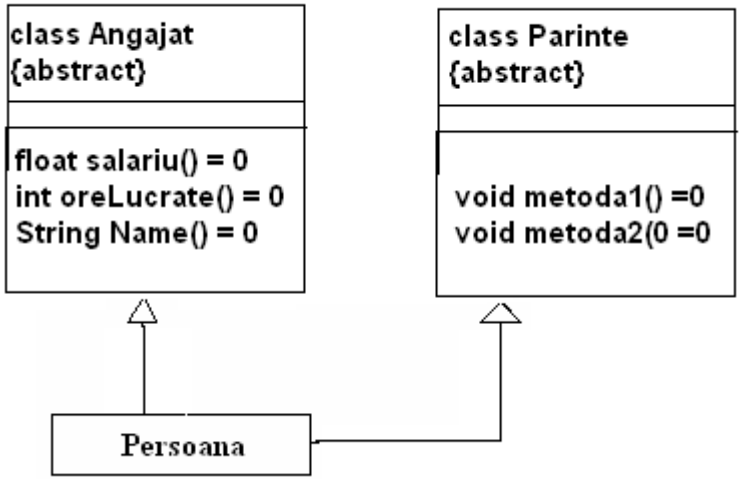
Interfete in C++

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



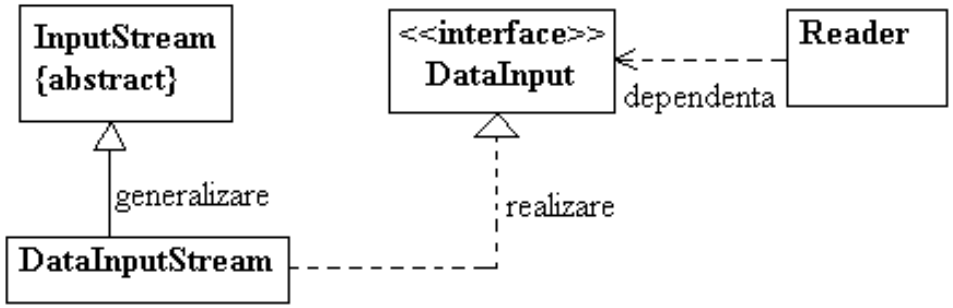
Interfeţe şi clase

In concluzie:

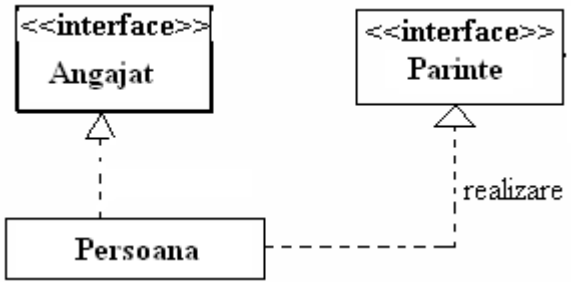
- O interfaţă este un set de metode corelate care definesc o anumita comportare.
- Toate metodele sunt publice si nu se specifica nici un fel de implementare pentru ele.
- O interfata nu are stare (nu contine variabile).

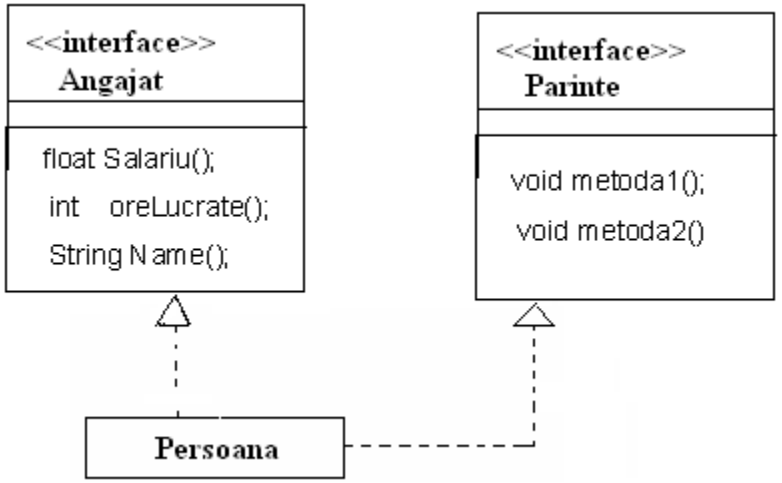
Intre interfete si clase pot fi stabilite relatii de realizare şi dependenţă:

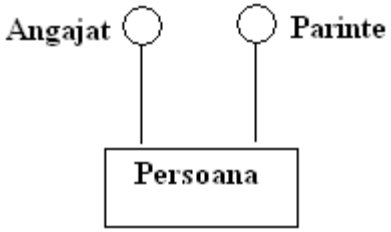
- clasa **InputStream** este abstracta.
- clasa **DataInputStream** ***implementeaza*** atat clasa abstracta **InputStream** cat si interfata **DataInput.**
- clasa **Reader** ***utilizeaza*** functiile oferite de interfata **DataInput**.

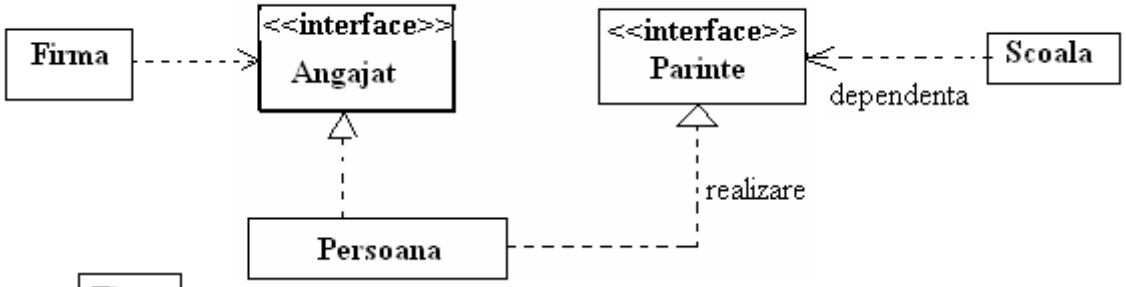


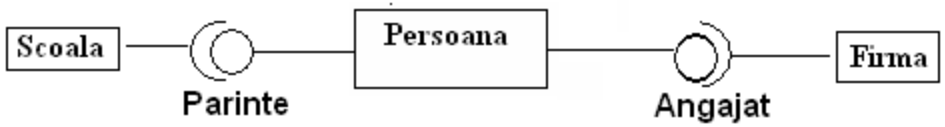
Reprezentarea relatiilor dintre clase si interfate











Intelegerea unei interfeţe

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

- pre si post conditii pentru fiecare operatie
- specificarea formala a semanticii, folosind OCL (Object Constraint Language, inclus in UML)
- se poate atasa un automat (diagrama de stari) pentru a specifica ordonarea in timp a operatiilor interfetei
- se pot atasa diagrame de colaborare pentru a specifica comportarea prevazuta pentru interfata.

Proiectarea de detaliu -1

Prof. univ. dr. ing. Florica Moldoveanu

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

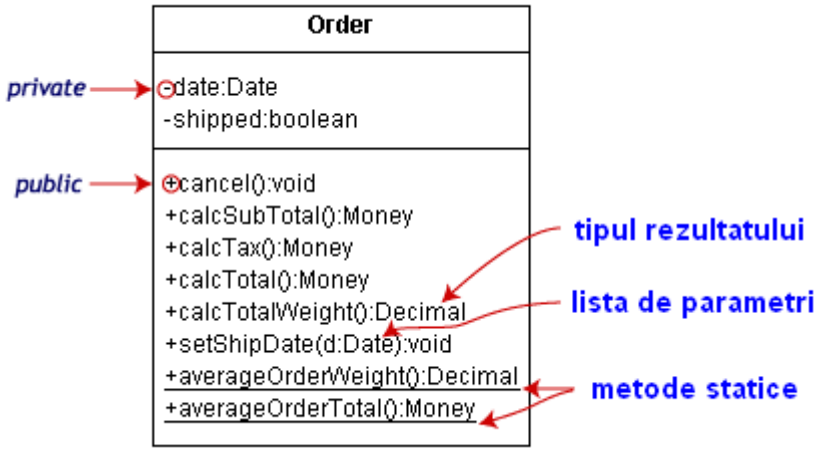
Proiectarea de detaliu

Activități:

- **Detalierea modelului arhitectural**: detalierea claselor, adaugarea de noi clase.
- **Optimizarea modelului arhitectural**: clasele existente în modelul arhitectural pot fi modificate pentru a se îmbunatăți timpul de raspuns, utilizarea memoriei, pentru a se reduce complexitatea codului, pentru a reduce cuplarea subsistemelor, pentru a obtine clase reutilizabile, s.a.
- **Reutilizarea**: folosirea soluțiilor existente.
- **Definirea interfeţelor subsistemelor** în limbajul de implementare.

Detalierea claselor şi specificarea interfețelor

Detalierea claselor



Pot fi identificate operații şi obiecte adiționale necesare pentru transferul datelor între subsisteme.

Interfeţele subsistemelor se specifică folosind interfeţele claselor care le implementeaza: operațiile, tipul parametrilor şi al valorii întoarse, tratarea excepțiilor.

Specificația interfeței furnizate de un subsistem, în limbajul de implementare, este adesea numita **Interfața de Programare a Subsistemului: subsystem API (Application Programming Interface)**

Recomandari pentru proiectarea de detaliu a claselor (1)

❖ Derivate din experienta mai multor proiecte industriale (**Lorenz,**1993)

- Dimensiunea medie a unei metode (Lines Of Code - LOC):

< 24 LOC pentru programe C++

- Numarul mediu de metode/clasa:

< 20

un numar mediu mai mare indica prea multa responsabilitate in putine clase

- Numarul mediu de variabile instanta (atribute) / clasa:

< 6 ; mai multe inseamna ca o clasa “face” prea mult.

- Numarul mediu de linii comentariu / metoda: >1

- Adancimea arborelui de mostenire (Depth of Inheritance Tree):

< 6, incepand de la radacina sau de la clasele bibliotecii de dezvoltare

Recomandari pentru proiectarea de detaliu a claselor (2)

- Numarul de relatii clasa - clasa in fiecare subsistem:

- trebuie sa fie relativ mare → coeziune in interiorul fiecarui subsistem
- daca o clasa dintr-un subsistem nu interactioneaza cu multe alte clase din subsistem→ ar trebui plasata in alt subsistem.

- Numarul de relatii subsistem - subsistem:

< numarul de relatii clasa - clasa din fiecare subsistem

- Utilizarea atributelor:

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

→ clasa ar putea fi divizata in mai multe clase

(sa existe coeziune prin date la nivelul fiecarei clasei)

- Numarul de reutilizari ale unei clase:

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

→ ar putea fi necesar sa fie reproiectata

Reutilizarea

Reutilizare: identificarea soluțiilor existente

La mai multe niveluri:

- Reutilizarea unor componente existente: se pot folosi c**omponente "of-the-shelf"** (componente binare) pentru implementarea componentelor din arhitectura sistemului.
- Reutilizarea unor soluții de proiectare: **sabloane de proiectare**
- Reutilizarea codului:

  - Reutilizarea codului unei clase existente: **mostenire**

  - Reutilizarea unei operații dintr-o clasă existentă: **delegare**

Adesea, subsistemele definite trebuie ajustate pentru folosirea componentelor "of-the-shelf" iar şabloanele de proiectare trebuie să fie adaptate pentru a fi utilizate în realizarea subsistemelor. Adaptarea se poate efectua prin clase "wrapper" sau utilizând moştenirea şi delegarea.

Sabloane de proiectare(1)
(Design patterns)

Ce este un şablon de proiectare?

- Un şablon de proiectare descrie o problemă care se întalneşte în mod repetat în proiectarea programelor şi soluția generală pentru problema respectivă, astfel încât să poată fi utilizată oricând dar nu în acelaşi mod de fiecare data.
- Soluția este exprimată folosind clase şi obiecte.
- Atat descrierea problemei cât şi a solutiei sunt abstracte astfel încat să poata fi folosite în multe situatii diferite.
- **Cartea de referinta**: “Design Patterns - Elements of Reusable Object-Oriented Software”, cunoscuta si sub numele “Gang of Four” [1]

Sabloane de proiectare(1)

Un sablon de proiectare are 4 elemente esentiale:

1. **Un nume** prin care poate fi referit →se creaza un vocabular de proiectare.
2. **Descrierea problemei**. Se explica problema pentru care ar trebui aplicat sablonul si contextul in care apare.
3. **Descrierea solutiei**: elementele care compun şablonul (clase, interfețe, obiecte), relațiile dintre ele, responsabilitatile si colaborarile.
4. **Consecintele aplicarii sablonului**: permit evaluarea alternativelor de proiectare, intelegerea costurilor si a beneficiilor aplicarii unui sablon.

Alte elemente: motivatia, aplicabilitatea**,** probleme de implementare, exemple de cod.

Șabloane de proiectare(2)

Clasificare - bazata pe scopul sabloanelor:

1. Creaționale (Creational patterns)

- Definesc **mecanisme de creare a obiectelor** adecvate unor anumite situații.

Exemple: Singleton, Abstract factory, Factory method, Builder, Prototype

2. Structurale (Structural patterns)

- Definesc **moduri de asamblare a claselor** pentru a obține structuri flexible şi eficiente, adecvate unor anumite situații.

Exemple: Adapter, Bridge, Composite, Decorator, Façade

3. Comportamentale (Behavioral patterns)

- Definesc **şabloane de comunicare între obiecte**, care cresc flexibilitatea în realizarea comunicării.

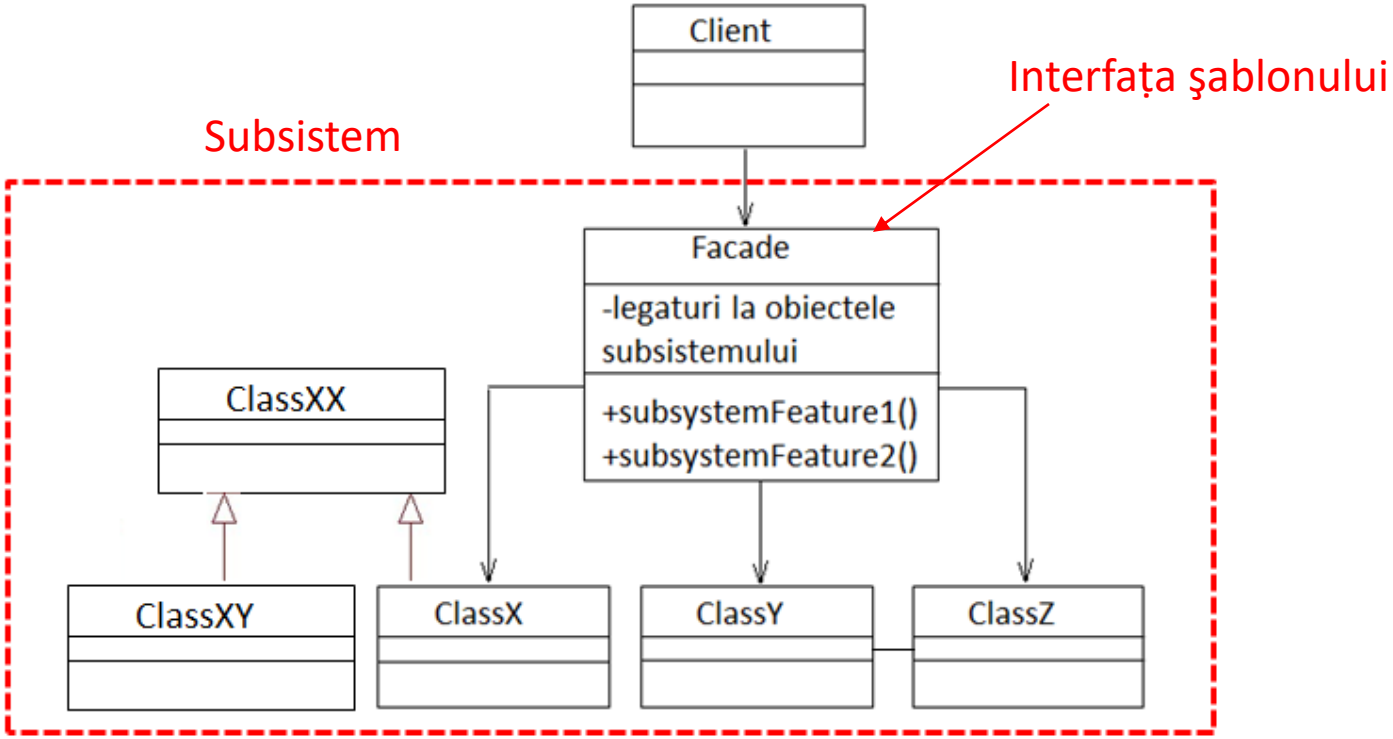
Exemple: Command, Mediator, Strategy, Observer, Visitor

Șablonul Façade (1)

Nume: Façade; **tipul**: structural

Descrierea problemei: simplificarea comunicării clienților unui subsistem complex cu obiectele subsistemului.

Soluția: Clasa *Façade* furnizează o interfață simplă pentru un sistem complex, oferind numai serviciile necesare clienților subsistemului.



Șablonul Facade (2)

Consecințe:

- Clientul apelează operațiile clasei Façade în loc să apeleze operațiile claselor subsistemului.
- Clientul nu cunoaşte clasele subsistemului. Dependențele dintre Client şi subsistem sunt minimizate. Clasele subsistemului pot fi modificate fara a afecta clientul.
- Clasele subsistemului nu cunosc existenta clasei Façade. Ele coopereaza în cadrul subsistemului pentru a realiza functionalitatile oferite de subsistem prin clasa Façade.

▪ Un subsistem poate oferi mai multe clase Façade, pentru tipuri diferite de clienți.

Clasa *Client* acceseaza şablonul.

Clasele *Client* pot fi clase existente într-o bibliotecă de clase sau clase noi ale sistemului în dezvoltare.

Interfața unui şablon este partea şablonului vizibilă clasei client. Poate fi o clasă, o clasă abstractă sau o interfață.

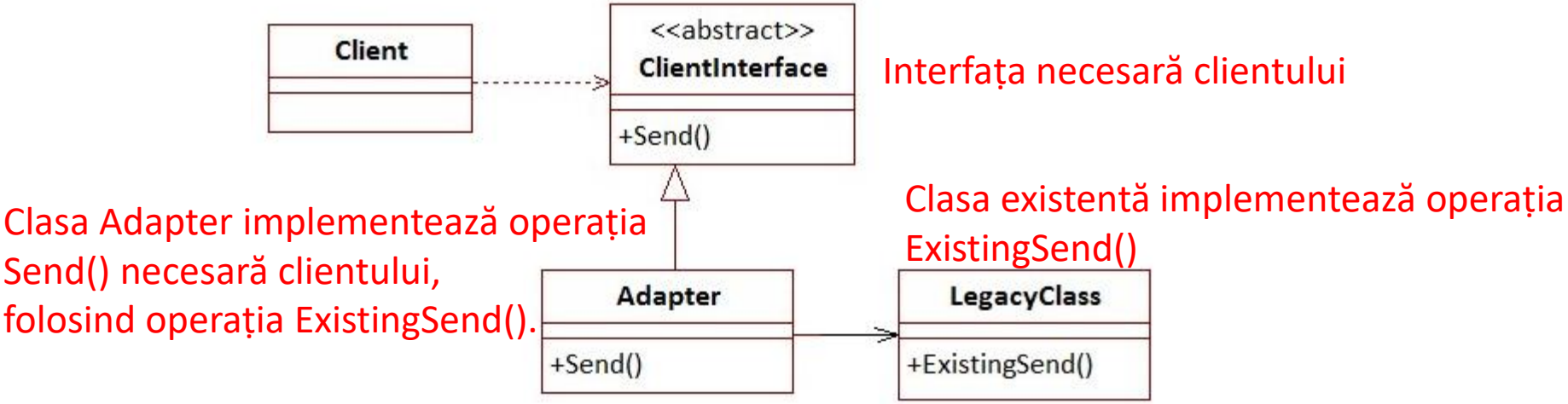
Interfața şablonului Façade este clasa Façade.

Șablonul Adapter (1)

Nume: Adapter; **tipul**: structural

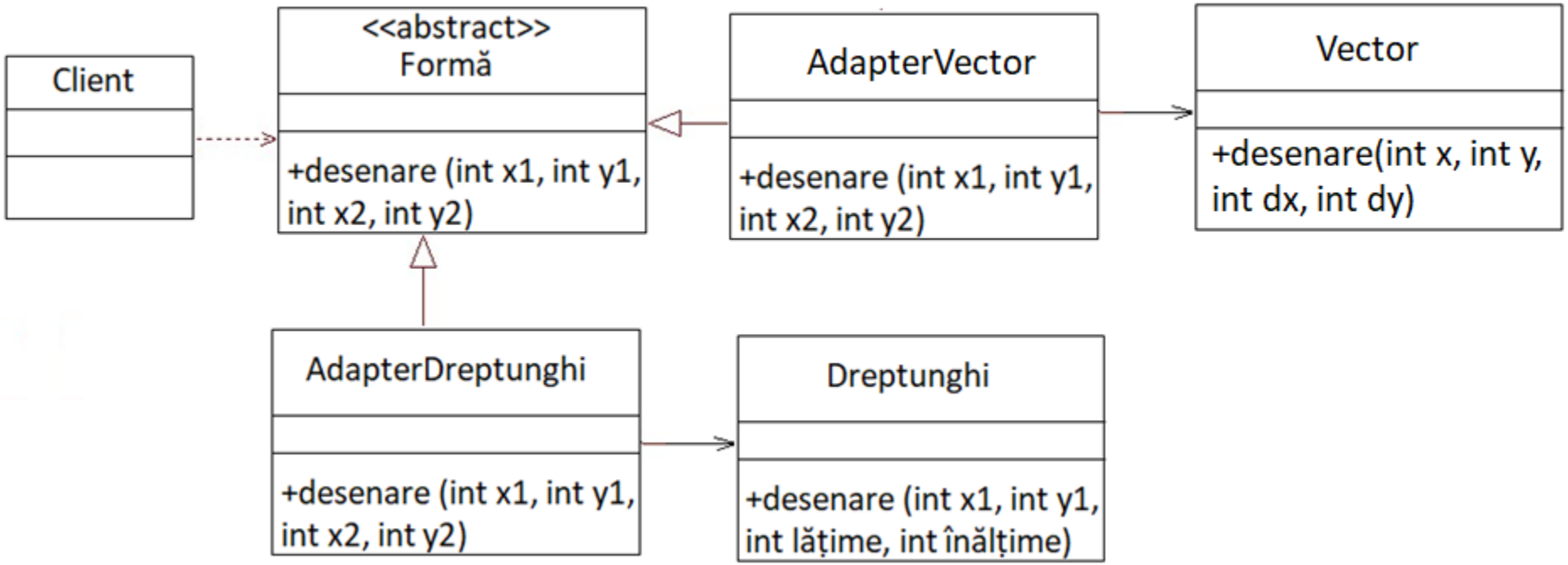
Descrierea problemei: utilizarea în implementarea unui sistem (Client) a unei clase/componente existente, care furnizează o interfață diferită de interfața necesară Clientului.

Soluția: Clasa *Adapter* implementează interfața necesară Cllientului delegând cererile *clientului* către clasa existentă (*LegacyClass*), după efectuarea conversiilor necesare asupra structurilor de date sau a comportamentului, astfel încat *Adapter* oferă comportamentul aşteptat de client.



Șablonul Adapter (2)

Exemplu de aplicare a şablonului



Șablonul Adapter (3)

Consecințe:

- Clasele Client si LegacyClass pot fi folosite împreuna fara modificarea niciuneia.
- Clasa Adapter se poate folosi împreună cu LegacyClass şi toate subclasele sale.
- **Pentru fiecare specializare (subclasa) a clasei ClientInterface trebuie creata o clasa Adapter.**

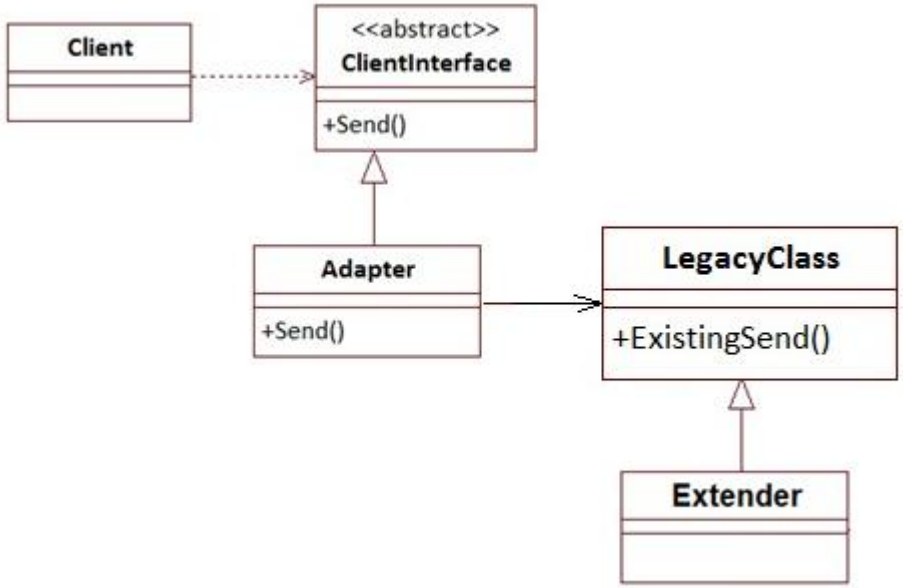
▪ **Interfata şablonului** este clasa abstractă *ClientInterface*.

Clasele care implementează comportamentul sablonului Adapter sunt *LegacyClass* si *Adapter.*

▪ Intr-un sablon de proiectare pot exista si **clase *Extender***, care specializeaza o clasa de implementare pentru a furniza o implementare diferita sau a extinde comportarea sablonului.

Șablonul Adapter (4)

- In sablonul **Adapter**, subclasele clasei *LegacyClass* sunt clase *Extender.*



❖ Sablonul **Adapter**

- permite reutilizarea deoarece nici interfața necesară clientului (*ClientInterface*), nici clasa existentă (*LegacyClass*) nu trebuie sa fie modificate.

- încurajează extensibilitatea: sablonul poate fi extins cu oricate clase *Extender* fara a fi necesara modificarea clasei *Adapter.*

Șablonul Bridge (1)

Nume: Bridge; **tipul**: structural

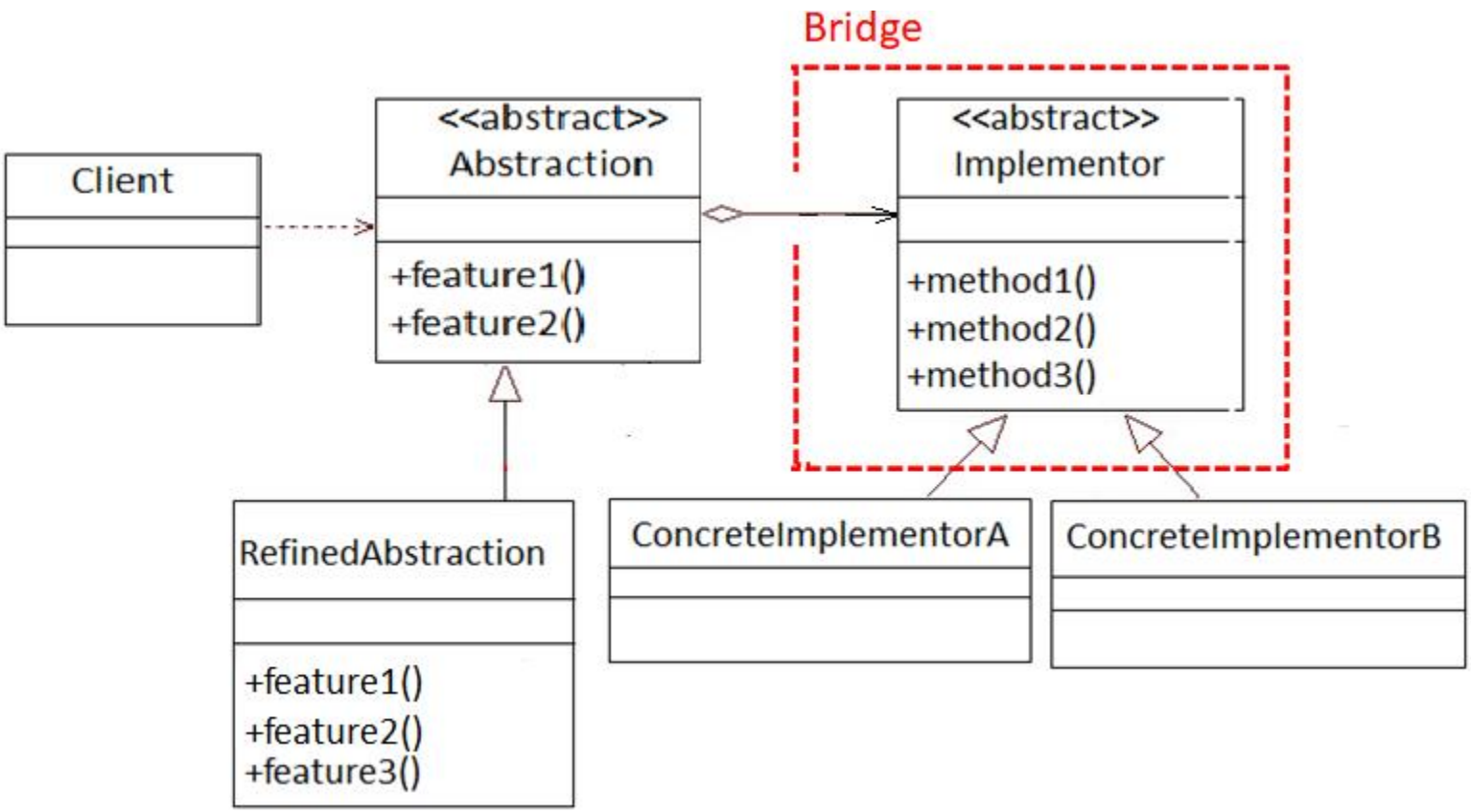
Descrierea problemei:

- Decuplarea unei abstracții de o implementare a sa, astfel încât ambele să poată fi modificate independent.
- Separarea unui set de clase corelate în două ierarhii de clase – *Abstraction* şi *Implementation* – care pot fi dezvoltate independent una de cealaltă.

Soluția: decuplarea se efectuează printr-o interfață care acționează ca un “pod” între abstracție şi implementările sale concrete

Șablonul Bridge (2)

- Clasa *Abstraction* defineşte interfața vizibilă clientului.
- *Implementor* este interfața care defineşte metodele necesare implementării clasei Abstraction.
- *RefinedAbstraction* implementează şi rafinează interfața *Abstraction.*
- Codul clientului leagă un obiect *RefinedAbstraction* la un obiect *ConcreteImplementor.*



Șablonul Bridge (3)

Consecințe

— Clientul este protejat de implementările concrete ale interfețelor *Abstraction* şi *Implementor*

— Ierarhiile *Abstraction* şi *Implementor* pot fi dezvoltate şi modificate independent.

Exemplu de aplicare: Clientul este o aplicație multi-platformă cu mai multe interfețe utilizator.

- *Abstraction* poate fi interfața grafică cu utilizatorul (GUI).
- *Implementor* poate fi API-ul sistemului de operare folosit de nivelul GUI – defineşte funcțiile apelate de implementarea GUI ca răspuns la interacțiunea cu utilizatorul.

Avantajele aplicarii şablonului: aplicatia Client poate fi extinsa:

- cu noi GUI - ierarhia GUI poate fi extinsă şi modificată independent de clasele de implementare (se adaugă/modifică clasele *RefinedAbstraction*)

- pentru a putea rula pe mai multe sisteme de operare- se adauga subclase *ConcreteImplementor*

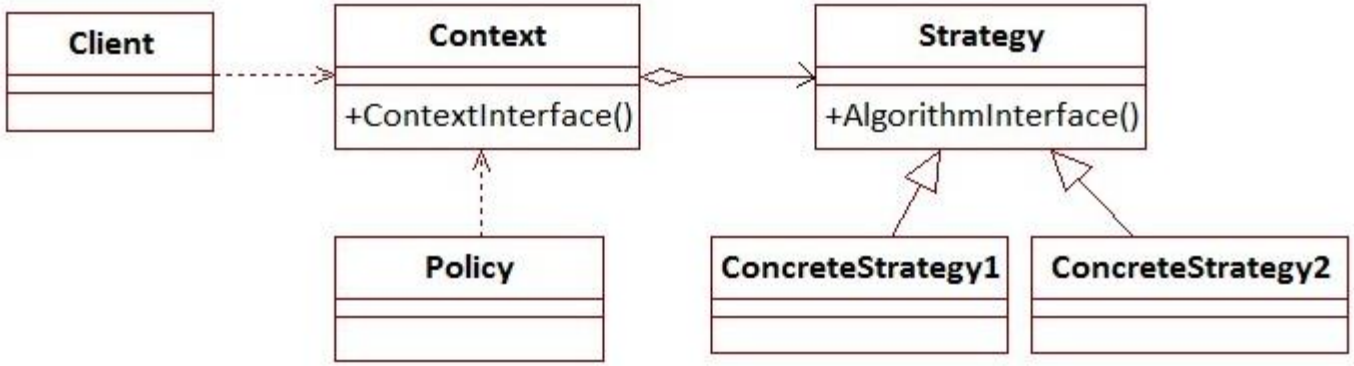
Șablonul Strategy (1)

Nume: Strategy; **tipul**: comportamental

Descrierea problemei: Decuplarea unei clase care decide o politica de clasele care implementeaza diferite strategii, astfel incat implementarile strategiilor sa poata fi inlocuite in mod transparent clientului.

Solutia: Un client acceseaza serviciile furnizate de o clasa Context, care sunt realizate de unul sau mai multe mecanisme, asa cum se decide intr-un obiect Policy.

Clasa Strategy descrie interfata comuna tuturor mecanismelor care pot fi utilizate de Context. Un obiect Policy creaza un obiect ConcreteStrategy si configureaza obiectul Context pentru a-l utiliza.



Șablonul Strategy (2)

Consecinte:

- Obiectele ConcreteStrategy pot fi inlocuite in mod transparent de Context.
- Obiectul Policy decide care strategie este mai buna in functie de cerinte (de ex., compromis intre viteza si spatiu de memorie, etc.)
- Pot fi adaugati noi algoritmi fara a modifica interfata sablonului (Context)

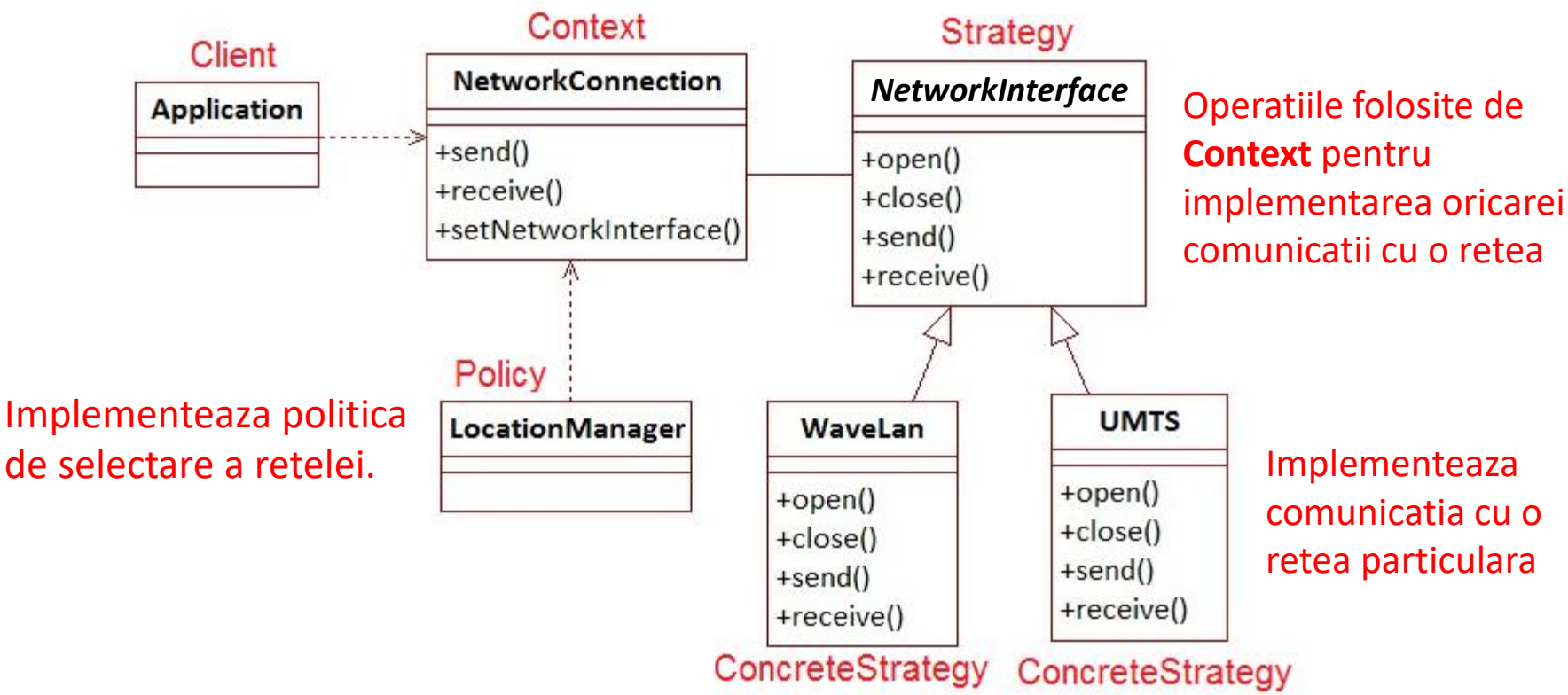
Sabloane corelate: Adapter si Bridge.

Exemplu: o aplicatie mobila trebuie sa schimbe in mod dinamic reteaua de comunicatie in functie de diferite criterii (locatia utilizatorului, costul comunicatiei, etc.).

Aplicatia trebuie sa nu depinda de reteaua de comunicatie folosita la un moment dat: are nevoie de o interfata comuna pentru diferitele retele.

Pentru a separa politica de selectare a retelei de interfata cu reteaua, se incapsuleaza implementarile protocoalelor de acces la retea folosind sablonul Strategy.

Șablonul Strategy (3)



- **LocationManager** configureaza dinamic **NetworkConnection** (**setNetworkInterface()**) cu un obiect **ConcreteStrategy** pe baza locatiei curente.
- Aplicatia utilizeaza **NetworkConnection** in mod independent de reteaua de comunicatie.

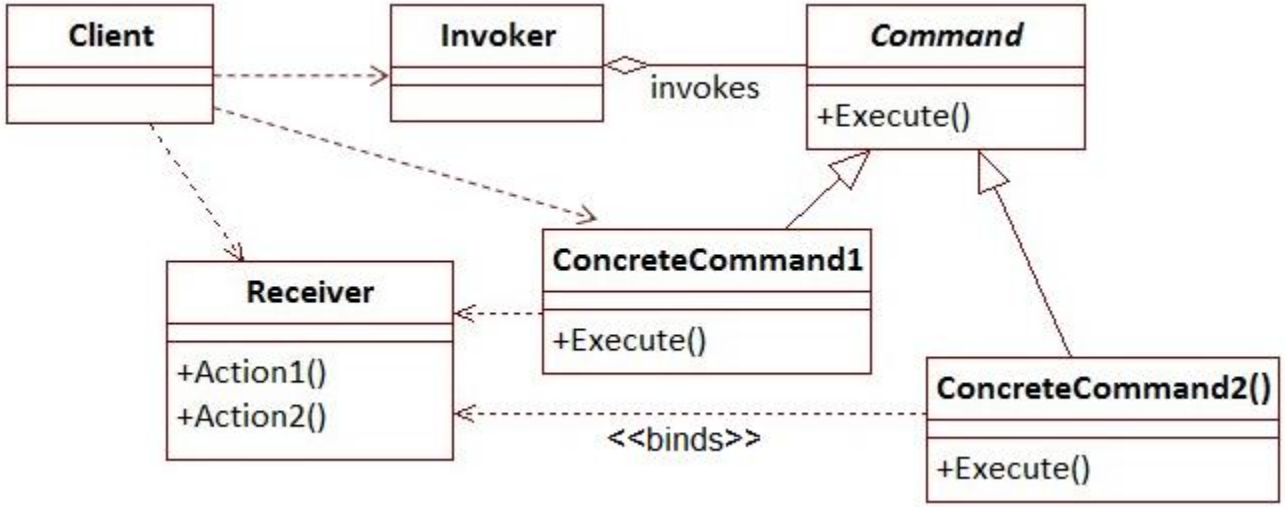
Șablonul Command (1)

Nume: Command; **tipul**: comportamental

Descrierea problemei: Separarea comenzilor de procesarea lor, astfel încat ele sa poata fi executate, anulate sau memorate într-o coada si procesate ulterior, independent de comanda.

Solutia: Clasa abstracta *Command* declara interfata care trebuie implementata de toate comenzile concrete (ConcreteCommands).

- Clientul creaza comenzi concrete si le "leaga" la obiecte *Receiver* specifice.
- Comenzile concrete sunt implementate folosind operatiile obiectului Receiver.
- *Invoker* executa sau anuleaza o comanda.

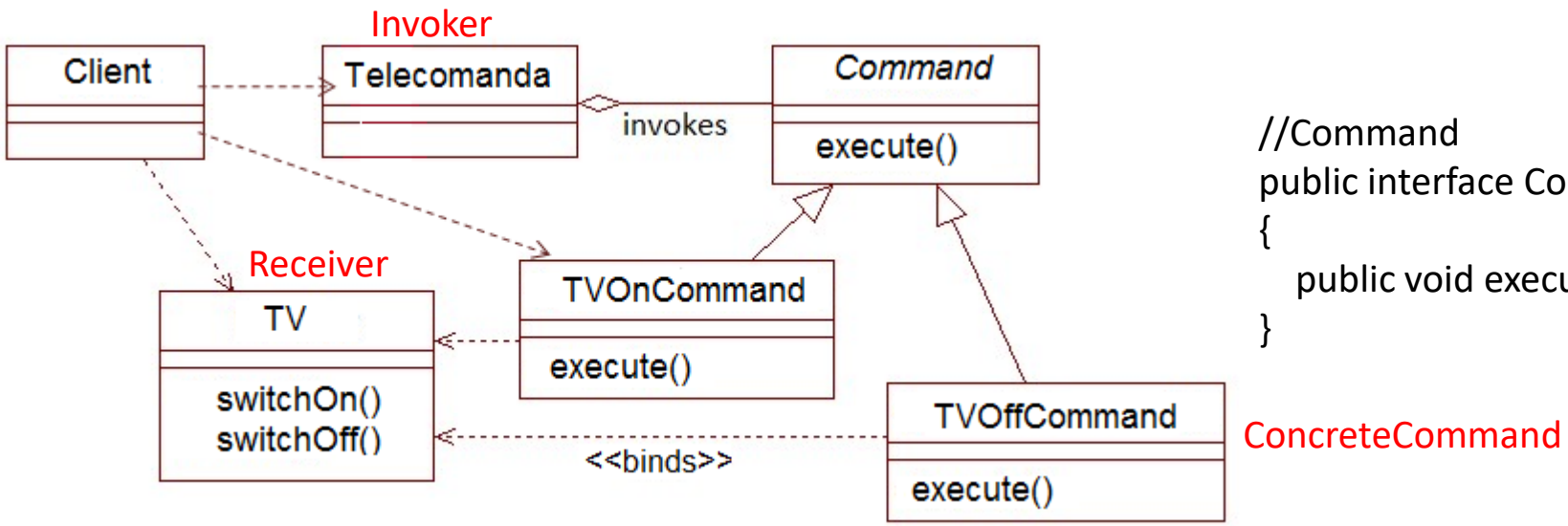


Șablonul Command (2)

Consecinte: Obiectul comenzii (Receiver) si algoritmul comenzii (ConcreteCommand) sunt decuplate.

- Invoker nu cunoaste comenzi specifice.
- ConcreteCommands sunt obiecte care pot fi create si memorate.
- Pot fi adaugate noi obiecte ConcreteCommands (noi functionalitati) fara modificarea codului existent.

Șablonul Command (3) – exemplu




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

```
//Comanda concreta „TVOn”
public class TVOnCommand implements Command
{  //referinta la un obiect Receiver
   TV tv;
   //  se leaga comanda concreta la un obiect Receiver
   public TVOnCommand (TV tv )
   {   this.tv = tv; }
   public void execute()
   {   tv.switchOn();  }
}
```

```
//Comanda concreta „TVOff”
public class TVOffCommand implements Command
{  //referinta la un obiect Receiver
   TV tv;
   //  se leaga comanda concreta la un obiect Receiver
   public TVOffCommand(TV tv )
   {   this.tv = tv; }
   public void execute()
   {   tv.switchOff();  }
}
```

Șablonul Command (4) – exemplu

```
//TV este clasa Receiver
 public class TV
{   private boolean on;
   public void switchOn()
   { on = true; }
   public void switchOff()
   { on = false; }
}
```

```
// Invoker: telecomanda
public class Telecomanda
{   private Command comanda;
   public void setCommand (Command comanda)
   { this.comanda = comanda; }
   public void pressButton()
   { comanda.execute(); }
}
```

```
// Client
public class Client
{   public static void main(String[] args)
   {
      Telecomanda control = new Telecomanda();
      TV myTV = new TV();
      Command TVOn = new TVOnCommand(myTV);
      Command TVOff = newTVOffCommand(myTV);
      //switch on
         control.setCommand(TVOn);
      //executa comanda
         control.pressButton();

      //switch off
         control.setCommand(TVOff);
         control.pressButton()
   }
}
```

Lecturi suplimentare

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software, cunoscuta si sub numele “Gang of Four” (GoF), Addison Wesley,1994.
2. https://sourcemaking.com/design_patterns
3. https://refactoring.guru/design-patterns

Proiectarea de detaliu -2

Prof. univ. dr. ing. Florica Moldoveanu

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

Șablonul Abstract Factory (1)

Nume: Abstract Factory; **tipul**: creațional

Descrierea problemei: Posibilitatea de a lucra cu un set de concepte (produse abstracte) pentru care sunt posibile mai multe implementări, fără a depinde de implementarile concrete.

De exemplu, toate platformele software implementează un acelaşi set de elemente de interfață utilizator: buton, checkbox, listbox, etc. Acestea reprezintă concepte (produse abstracte), căci funcționalitățile lor sunt aceleaşi, independente de modul de implementare.

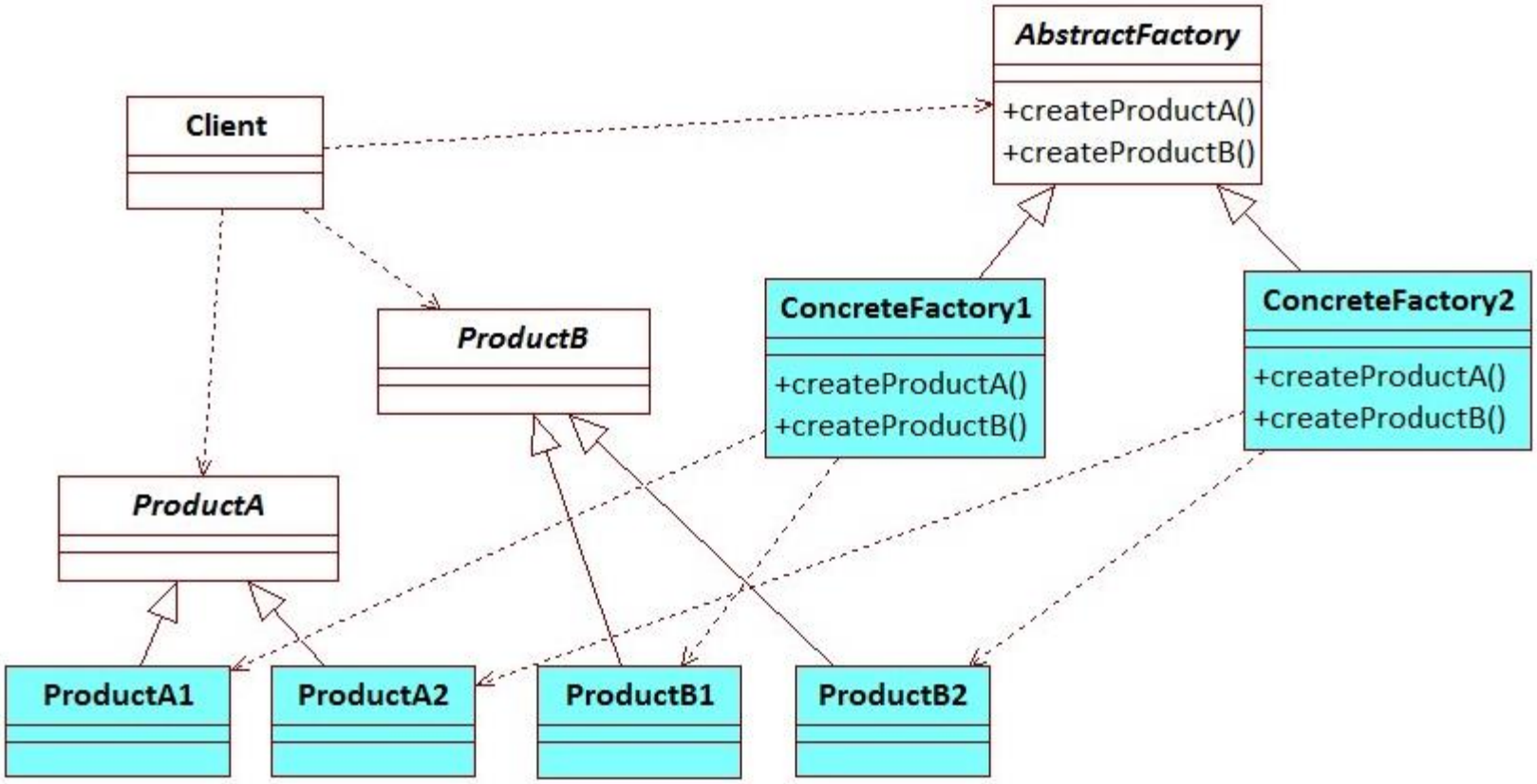
Soluția

Fiecare **produs abstract** dintr-o familie **este reprezentat printr-o interfață, *Product*.**

O platformă abstractă – *AbstractFactory***,** este reprezentată de un set de produse abstracte. Interfața *AbstractFactory* declară operațiile de creare a produselor abstracte individuale: CreateProductA(), CreateProductB(), etc.

O platformă specifică este reprezentată printr-o **clasă *ConcreteFactory*** şi un set de clase care implementează **produsele proprii platformei** (câte una pentru fiecare produs abstract).

Șablonul Abstract Factory (2)



Consecințe: clientul este independent de clasele produselor concrete, independent de platforma.

- Este posibila înlocuirea familiilor de obiecte în timpul execuției.
- Adaugarea de produse noi este dificilă deoarece trebuie extinse toate clasele ConcreteFactory.

Șablonul Abstract Factory (3)

Exemplu:

O aplicație pentru o casă inteligentă foloseste diferite tipuri de senzori (de temperatură, de fereastră deschisă/inchisă, bec aprins/stins, etc), identifică anumite conditii predefinite si trimite comenzi catre dispozitive de actionare (aer conditionat on/off, declanşarea unei alarme, etc).

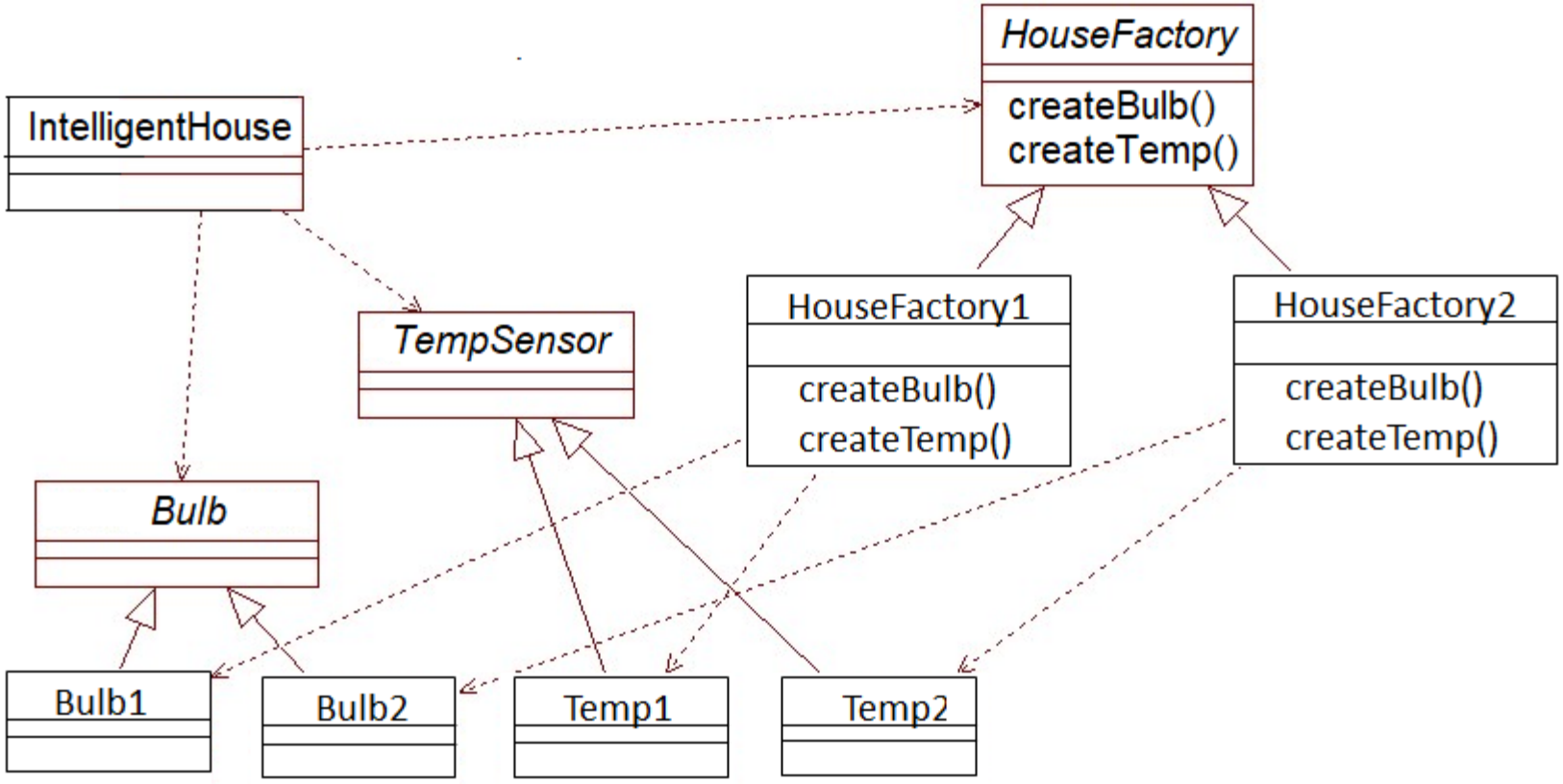
Aplicația nu trebuie să depindă de particularitătile de realizare a fiecărui dispozitiv utilizat, specifice fiecărui fabricant, de aceea lucrează cu dispozitive generice, produse abstracte (*Bulb*, *TempSensor*).

Fiecare fabricant este reprezentat printr-o clasă (HouseFactory1, HouseFactory2) care furnizează metodele de creare a dispozitivelor abstracte (CreateBulb, CreateTemp)**.**

Clasele Client accesează numai interfețele *HouseFactory, Bulb* şi *TempSensor*, fiind astfel protejate de particularitatile dispozitivelor produse de diverşi fabricanți.

Șablonul Abstract Factory (4)

Diagrama de clase a aplicației **Casa inteligentă**



```
#include <iostream>
class Bulb //clasa abstracta
{  public:
   virtual float cost() = 0;
};

class HouseFactory// clasa abstracta
{ public:
  virtual Bulb * createBulb() = 0;
//virtual TempSensor * createTemp()=0;
};

class Bulb1 : public Bulb
{ public:
  Bulb1() {}
  float cost() { return 3.5; }
};

class HouseFactory1:public HouseFactory
{ public:
  HouseFactory1() {}
  Bulb * createBulb()
  { return new Bulb1(); }
 // TempSensor * createTemp(){…};
};
```

```
class IntelligentHouse
{ HouseFactory * hF;
  Bulb * aBulb; // TempSensor * aTemp;
  public:
  IntelligentHouse
  (HouseFactory *_hF){ hF = _hF;}

  void configurate()
  { aBulb = hF->createBulb();
   //aTemp = hF->createTemp();
  }

  float cost()
  { return aBulb->cost()/*+ aTemp->cost()*/;}
};

int main()
{HouseFactory * aFactory = new  HouseFactory1();
IntelligentHouse * anIntelligentHouse = new
IntelligentHouse(aFactory);
anIntelligentHouse->configurate();
std::cout << anIntelligentHouse->cost();
……————————————————————————————————
}
```

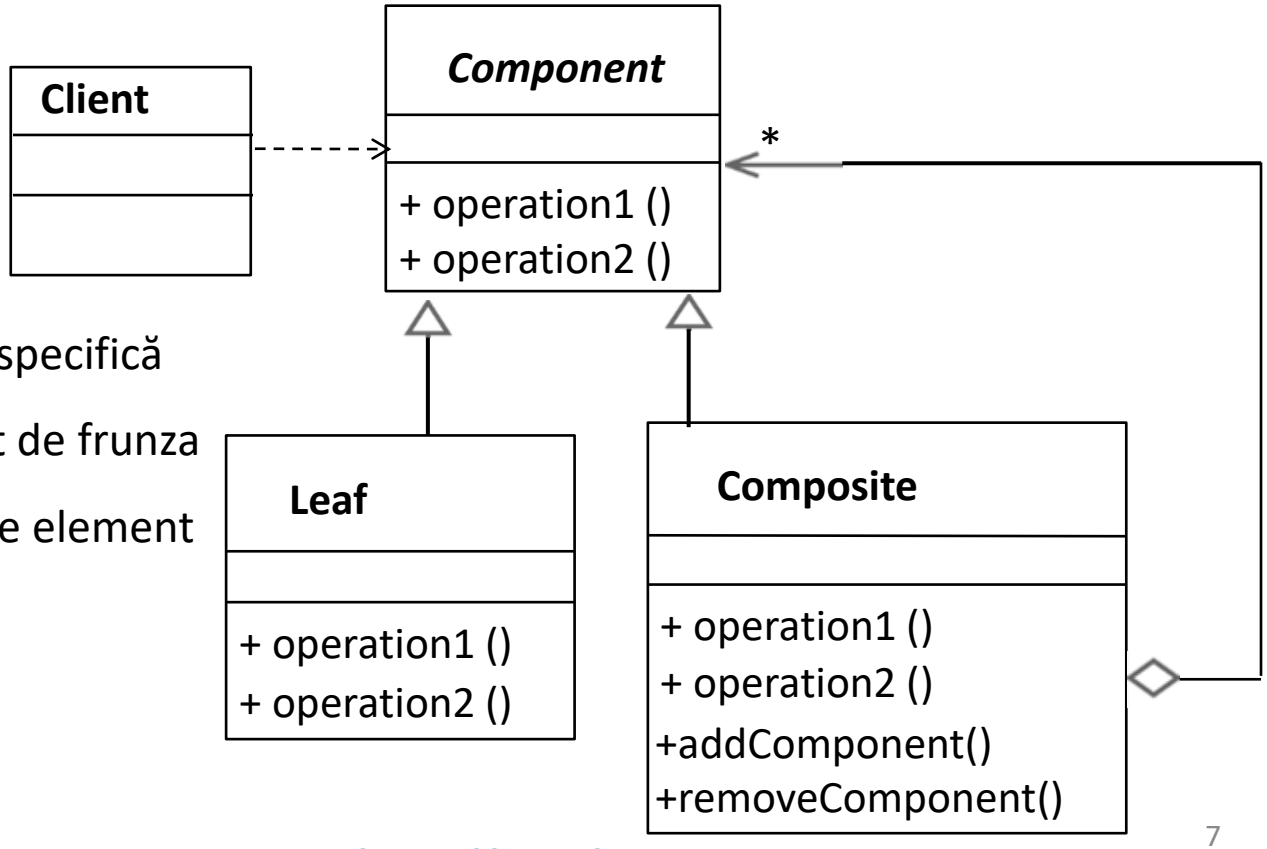
Șablonul Composite (1)

Reprezentarea ierarhiilor recursive

Nume: Composite; **tipul**: structural

Descrierea problemei: reprezentarea unei ierarhii de înaltime şi lățime variabile astfel încât frunzele şi agregatele să poată fi tratate uniform, prin aceeasi interfață.

Solutia: Interfața *Component* specifică serviciile care sunt oferite atât de frunza cât şi de agregat pentru fiecare element componenta sa.



Șablonul Composite (2)

- Agregatul implementează fiecare serviciu pentru fiecare componenta pe care o conține.
- Obiectele frunza (Leaf) implementează efectiv serviciile (Leaf.move(x,y)).

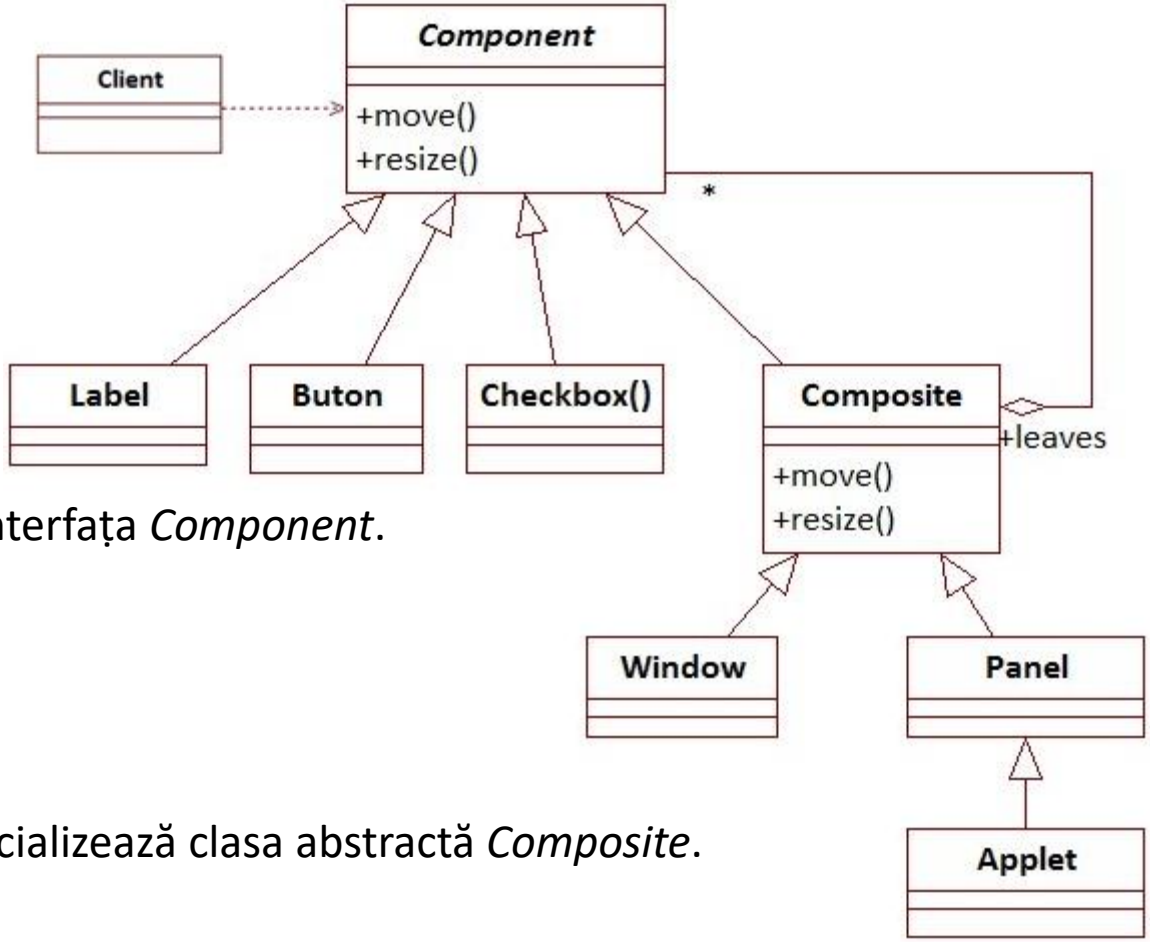
Consecințe:

- Clienții utilizează acelaşi cod pentru lucrul cu obiectele frunză şi cu obiectele agregat.
- Comportarea implementată în clasa Leaf poate fi modificată fără schimbarea ierarhiei.
- Pot fi adăugate noi clase de tip Leaf fără modificarea ierarhiei.

Exemple de ierarhii recursive:

- Grup de elemente grafice care pot fi translatate, scalate, etc. Un grup poate conține alte grupuri.
- Ierarhii de fişiere şi foldere. Un folder poate contine fişiere şi alte foldere. Aceleaşi operatii pot fi folosite pentru redenumire, mutare sau ştergere a fisierelor sau a folderelor.

Șablonul Composite (3)



Fiecare frunză implementează interfața *Component*.

Agregatele Panel si Window specializează clasa abstractă *Composite*.

Mutarea sau redimensionarea unui obiect Composite are impact asupra tuturor componentelor sale.

Șablonul Observer (1)

Nume: Observer, Subject-Observer, Dependents, Publish-Subscribe; **tipul**: comportamental

Descrierea problemei: definirea unei dependențe între obiecte, astfel încât atunci când un obiect îsi schimbă starea toate obiectele dependente de el să fie notificate şi actualizate.

Motivația: cresterea gradului de reutilizare a claselor, prin separarea obiectelor dependente în clase distincte slab cuplate.

Exemple:

1. In multe aplicații se doreste separarea datelor (si logicii) aplicației de prezentarile lor in interfața utilizator. In acest fel, **clasele care definesc datele şi logica aplicației şi cele care realizează prezentarea/prezentările pot fi reutilizate independent**.

   Obiectele care realizeaza prezentarile sunt dependente de obiectul care contine datele.

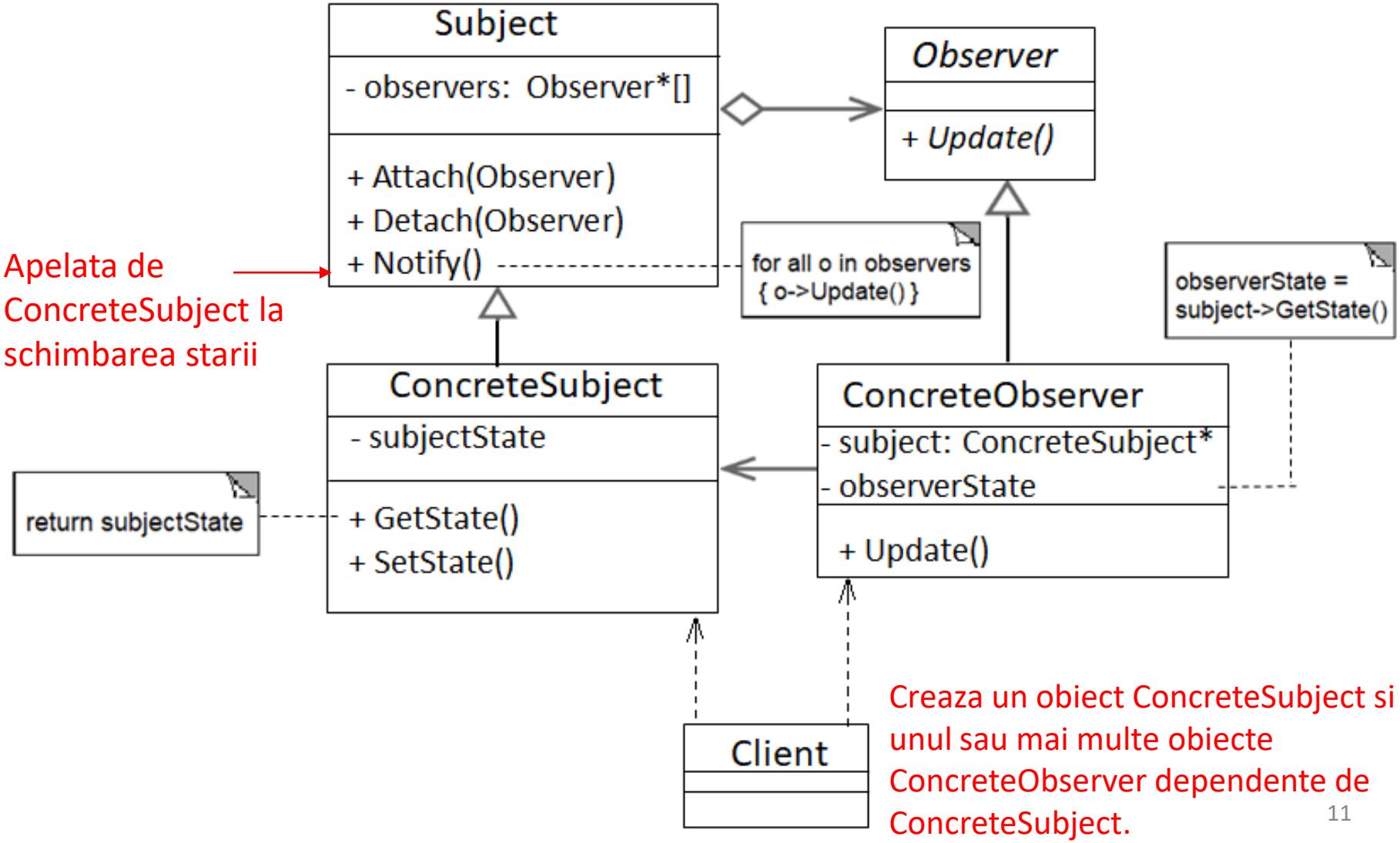
2. O aplicație care gestionează un magazin doreste să trimită notificări clienților care s-au abonat, de fiecare dată când se primeşte un produs, dar numai clienților care sunt interesați de acel produs.

   Clientii interesati de un produs sunt dependenti de continutul magazinului.

   Ei sunt notificati la aparitia produsului.

Șablonul Observer (2)

Soluția



Șablonul Observer (3)

Soluția - continuare

- Subiectul mentine o lista de observatori – obiectele dependente de subiect – pe care-i notifica ori de cate ori se modifica starea subiectului.

Subject

—Furnizează o interfață pentru ataşarea, detaşarea si notificarea observatorilor săi.

Observer

—Defineşte o interfață pentru actualizarea observatorilor.

ConcreteSubject

—Memorează starea de interes pentru obiectele **ConcreteObserver**.

—Trimite o notificare observatorilor săi atunci cand i se schimba starea.

ConcreteObserver

—Mentine o referinta la un obiect **ConcreteSubject**.

—Implementează funcția Update() pentru a păstra starea sa consistentă cu a subiectului.

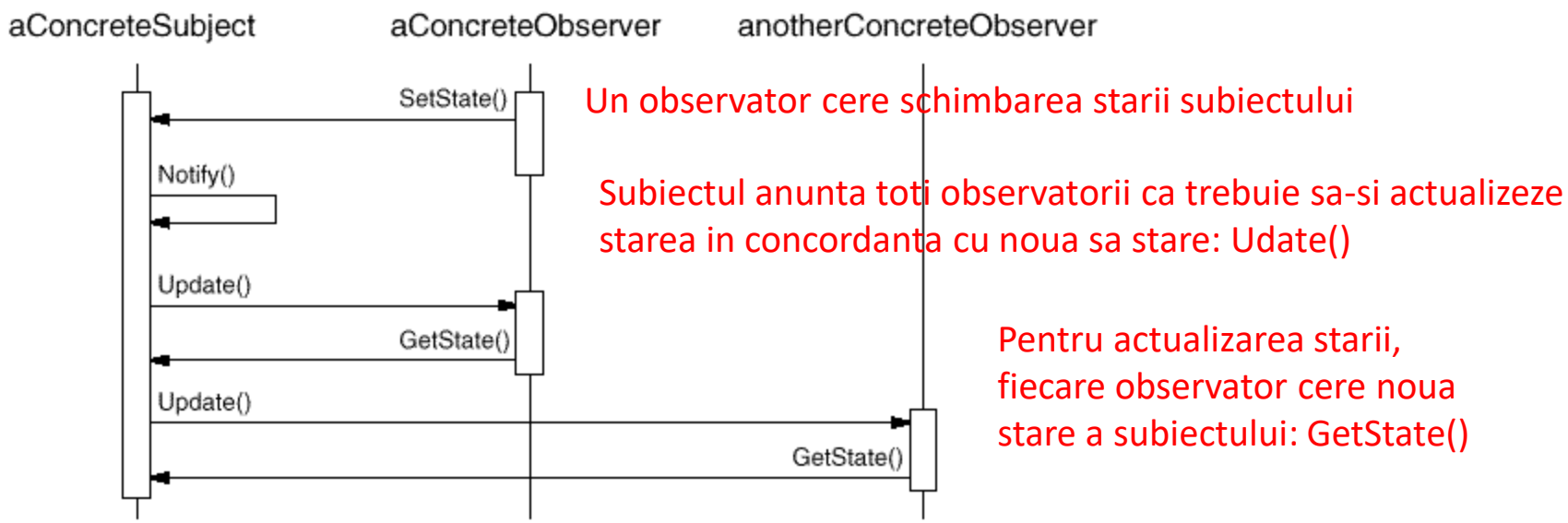
Șablonul Observer (4)

- Obiectele cheie in acest sablon sunt **subiectul** – obiectul independent (Subject, Publisher) si **observatorii –** obiectele dependente (Observers, Subscribers), care sunt notificate de subiect atunci cand anumite evenimente au produs schimbarea stării sale.
- Un subiect poate avea orice numar de observatori dependenti.
- Observatorii sunt notificati ori de cate ori subiectul isi schimba starea.
- Ca raspuns la notificare, fiecare observator va interoga subiectul pentru a-si sincroniza starea cu starea subiectului.
- Este posibil ca fiecare observator sa se înregistreze numai pentru anumite evenimente (schimbari de stare) produse la subiect, caz în care el va fi notificat numai la producerea acelor evenimente.
- Un observator se poate înregistra la mai multi subiecti.

Șablonul Observer (5)

Colaborări

- Un obiect *ConcreteSubject* notifica observatorii sai ori de cate ori are loc o schimbare care ar putea sa faca starea observatorilor inconsistenta cu starea sa.
- Dupa ce a fost informat de o schimbare la *ConcreteSubject*, un obiect *ConcreteObserver* il poate interoga pentru a obtine informatii.
- Obiectul *ConcreteObserver* utilizeaza aceste informatii pentru a-si adapta starea cu aceea a obiectului *ConcreteSubject.*



Șablonul Observer (6)

Consecințe

- Sablonul Observer permite definirea independenta a subiectilor si a observatorilor. **Subiectii si observatorii pot fi reutilizati independent unii de altii**. **Pot fi adaugati noi observatori fara a modifica subiectul sau alti observatori**.

Alte aspecte ale utilizarii sablonului Observer sunt:

- ***Cuplarea abstracta între Subject şi Observer***.

  Subiectul are doar o lista de observatori care expun interfata simpla a clasei abstracte Observer. Subiectul nu cunoaste clasa concreta a niciunui observator. In acest fel, **cuplarea intre subiecti si observatori este abstracta si minimala**. Deoarece subiectul si observatorul nu sunt puternic cuplati, ei pot apartine la diferite nivele de abstractizare ale unui sistem.

Șablonul Observer (7)

- ***Suport pentru o comunicare broadcast.***

  Atunci cand i se schimba starea, subiectul trimite o notificare catre toate obiectele observator care s-au inregistrat la el. Subiectul nu trebuie sa stie cate obiecte interesate exista; singura sa responsabilitate este sa notifice observatorii sai. In acest fel, observatorii pot fi adaugati sau eliminati in orice moment. Este datoria observatorului sa trateze sau nu o notificare.

- ***Actualizari neasteptate.***

  Deoarece observatorii nu stiu unul despre altul ei nu pot cunoaste costul unei modificari la subiect. Astfel o foarte mica modificare la subiect poate antrena un numar mare de actualizari la observatori si obiectele dependente de ei. Problema este agravata de faptul ca protocolul foarte simplu de actualizare nu da detalii despre ce anume s-a schimbat la subiect.

Șablonul Observer (8)
Exemple de implementare în C++

Exemplul 1

```
#include <list>
using namespace std;
class Observer;

class Subject
{ public:
      virtual ~Subject();
      virtual void Attach(Observer*);
      virtual void Detach(Observer*);
      virtual void Notify();
 protected:
      Subject();
 private:
      list<Observer*> observers;
 };

 void Subject::Attach (Observer* o)
 { observers.push_back(o); }

 void Subject::Detach(Observer*o)
 { observers.remove(o); }
```

Interfața **Observer** este definita printr-o clasa abstracta:

```
class Observer
{     public:
          virtual ~ Observer();
           virtual void Update(
                          Subject* theChangedSubject) = 0;
      protected:   Observer();
};
```

Implementarea admite ca un observator sa aiba mai multi subiecți.

Subiectul transmis operatiei **Update()** permite observatorului să determine care subiect şi-a schimbat starea atunci cand el observă mai multi subiecți.

Șablonul Observer (9)

```
void Subject::Notify ()
 {
     for (auto it=observers.begin(); it!=observers.end(); it++)
                         (*it)->Update(this);
}
```

ClockTimer este un ***subiect concret*** pentru memorarea si actualizarea orei curente. El notifica observatorii la fiecare secunda.

```
class ClockTimer : public Subject
{  public:  ClockTimer();
          virtual int GetHour();
          virtual int GetMinute();
          virtual int GetSecond();
          void Tick();
  protected:
       int ora, minutul, secunda; // starea interna
}
void ClockTimer::Tick() // apelata de un timer intern al calculatorului, la intervale regulate
{ * actualizeaza starea interna a subiectului concret
   Notify(); // anunta observatorii despre schimbarea starii subiectului
 }
```

Șablonul Observer (10)

Sunt definite doua clase “ConcreteObserver”: DigitalClock si AnalogClock. Ambele implementeaza interfata **Observer** şi afiseaza timpul curent memorat în clasa ClockTimer.

Pentru afişarea timpului se utilizeaza o clasa a interfetei grafice, numita Widget:

```
class Widget
{ public:
  virtual void Draw();
};

class DigitalClock: public Widget, public Observer
{   public: DigitalClock(ClockTimer*); //pointer la subiectul concret
        virtual ~DigitalClock();
        virtual void Update(Subject*); // implementeaza operatia Update() a Observer-ului
        virtual void Draw();  // redefineste operatia clasei Widget: defineste modul de afisare
                              //a ceasului digital;
    private:
            ClockTimer* subject; // pointer la “subiectul concret”
 };
```

Șablonul Observer (11)

```
 DigitalClock::DigitalClock (ClockTimer* s)
{   subject = s;
    subject->Attach(this); // se ataseaza la subiect
 }
DigitalClock::~ DigitalClock ()
 {  subject->Detach(this);
 }
void DigitalClock::Update (Subject* theChangedSubject)
{  if (theChangedSubject == subject)
      Draw();
}
 void DigitalClock::Draw ()
{   // preia starea curenta a subiectului
   int hour = subject->GetHour();
   int minute = subject->GetMinute();
   // etc.
  *draw the digital clock
  }
```

Șablonul Observer (12)

In mod similar este definita o clasa AnalogClock.

```
class AnalogClock : public Widget, public Observer
{  public:  AnalogClock(ClockTimer*);
   virtual void Update(Subject*);
   virtual void Draw();
    // ...
};
```

Client

// Creaza un obiect AnalogClock şi unul DigitalClock care vor afişa intotdeauna acelasi timp:

```
ClockTimer* timer = new ClockTimer;    //subiectul concret
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

Șablonul Observer (13)

Exemplul 2

```
#include <list>
using namespace std;

class Observer;

class Subject
{   public:
      ~Subject() {}
      virtual void Attach(Observer* o)
          { observers.push_back(o); }
      virtual void Detach(Observer* o)
          { observers.remove(o); }
      virtual void Notify()
      {   for (auto it=observers.begin(); it!=observers.end(); it++)
            (*it)->Update(this);
      }
   protected:
      Subject() {}
   private:
      list<Observer*> observers;
};
```

```
class Observer
{  public:
      virtual ~ Observer() {}
      virtual void Update(Subject* theChangedSubject) = 0;
   protected:
      Observer() {}
};
```

Șablonul Observer (14)

```
#include <iostream>
using namespace std;
#include <math.h>

class SubjectX : public Subject
{
   int x;
   public:
    SubjectX (int _x=0) {}
    void SetX (int _x)
    {
      x=_x;
      Notify();
    }
    int GetX ()
    {  return x; }
};

class Observer2 : public Observer
{
   public: void Update(Subject * s)   { cout << pow( ((SubjectX*)s)->GetX(), 2 ) << endl; }
};
class Observer3 : public Observer
{
   public: void Update(Subject * s)   { cout << pow( ((SubjectX*)s)->GetX(), 3 ) << endl; }
};
```

```
int main()
{  Observer2 o2;
   Observer3 o3;

   SubjectX sx;
   sx.Attach(&o2);
   sx.Attach(&o3);

   SubjectX sxi;
   sxi.Attach(&o2);

   sx.SetX(5);
   sxi.SetX(10);

   return 0;
}
```

Output:
25
125
100

Lecturi suplimentare

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software, cunoscuta si sub numele “Gang of Four” (GoF), Addison Wesley,1994.
2. https://sourcemaking.com/design_patterns
3. https://refactoring.guru/design-patterns
4. https://www.tutorialspoint.com/design_pattern

Testarea pe parcursul dezvoltării programelor

Prof. univ. dr. ing. Florica Moldoveanu

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

Definitii (1)

Scopul testarii: verificarea si validarea programelor.

Testarea in vederea verificarii programului:

- **Este implementarea produsului în conformitate cu specificația sa?**
- Efectuata la mai multe niveluri: unitate functionala, clasa, componenta, subsistem, sistem.
- Sunt verificate atât proprietățile interne cât şi cele externe ale produsului.

Testarea in vederea validarii programului, numita si testare de acceptare:

- **Este produsul în conformitate cu cerințele utilizatorilor/tuturor părților interesate?**
- Se efectueaza in mediul real de operare al programului cu participarea utilizatorilor finali, a clientului şi a celorlalte părți interesate in proiect
- Sunt verificate proprietățile externe ale produsului

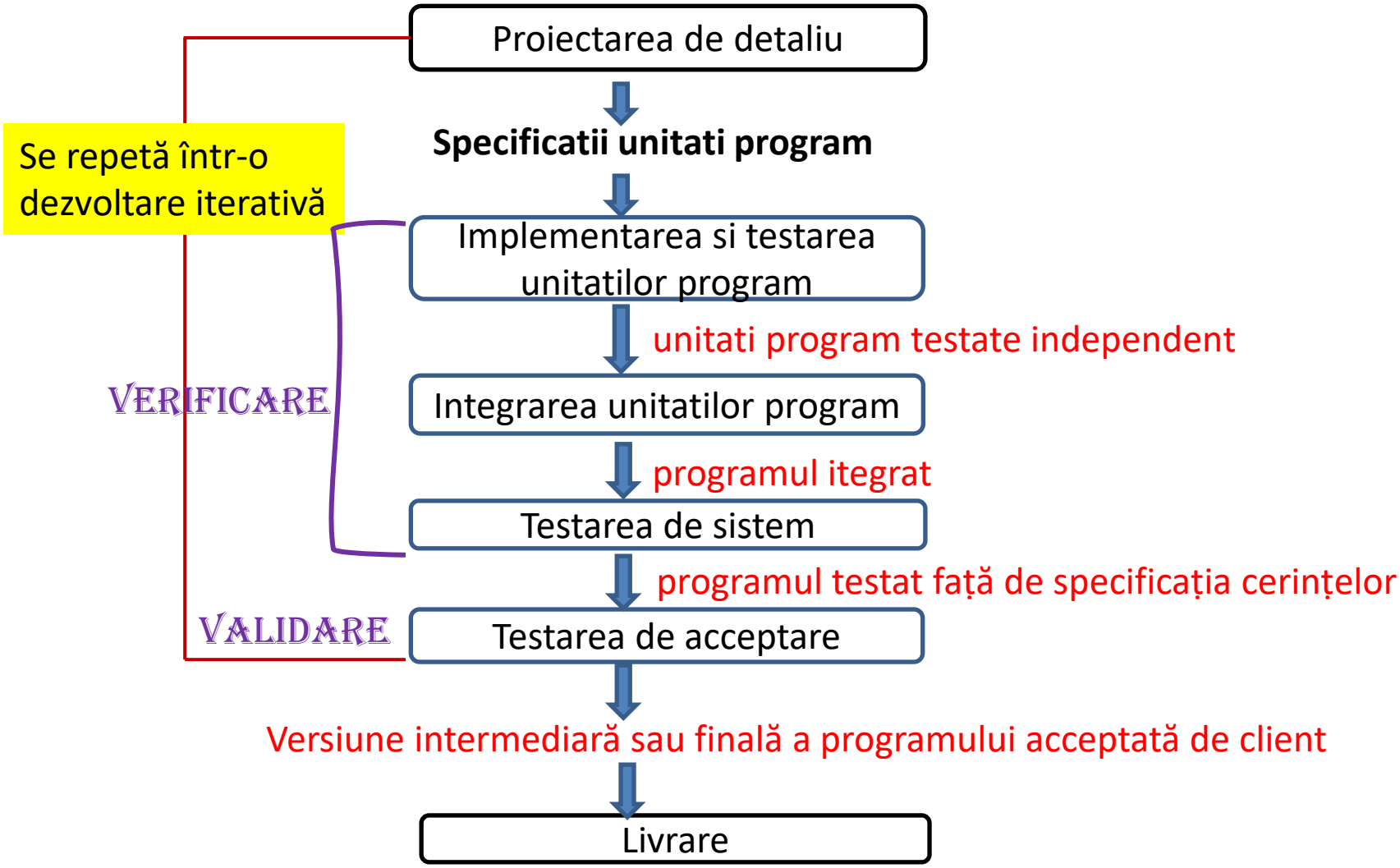
Definitii (2)

- Un caz de test (test case) specifica o executie de test a unui program, prin:
  - Datele de intrare – numite si date de test
  - Conditiile de executie (ex, preconditii)
  - Procedura de executie a testului (ex, ordinea in care se introduc datele de intrare)
  - Rezultatele asteptate
- O suita de test (test suite): un set de cazuri de test.
- Testarea: activitatea de conceptie a cazurilor de test, de executie a cazurilor de test si de evaluare a rezultatelor executiilor de test, in diferite etape ale ciclului de viata al unui program.
- Cădere (Eroare la o executie) = orice rezultat neconform cu specificatia.
- Defect = "bug" : o anomalie în produs (greşeala logică/ codificare) care are ca efect o cădere.

➢ Prin testare nu se poate demonstra corectitudinea unui program: nu poate fi executat pentru toate seturile de date de intrare posibile.

➢ Cazurile de test se aleg a.i. sa se verifice satisfacerea specificatiilor si descoperirea defectelor.

Testele pe parcursul dezvoltarii unui produs software

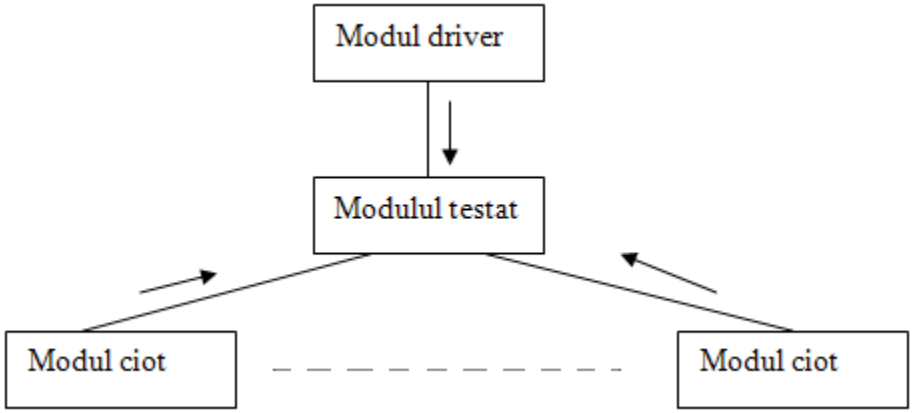


Testele unitare (1)

- Se efectueaza la nivelul unei unitati functionale de program: functie, procedura, functie membru a unei clase.
- Sunt realizate de programatorul care implementeaza unitatea program.

- Unitatea testata este tratata ca o entitate independenta, care nu necesita prezenta altor unitati ale programului.
- Unitatile program cu care interactioneaza unitatea testata sunt simulate prin: module « stub » (ciot) si un modul « driver ».
- Modulele ciot simuleaza modulele apelate de modulul testat. Un modul "ciot" are aceeasi interfata cu modulul real si realizeaza in mod simplificat functia sa.
- Modulul driver simuleaza cazurile de apel ale modulului testat de catre celelalte module ale programului.

Testele unitare (2)



Structura programului executabil pentru testarea izolata a unei unitati program.

- Fiecare executie este specificata printr-un caz de test
- La o executie modulul driver apeleaza modulul testat cu datele de intrare ale cazului de test.

Datele de intrare ale fiecarui caz de test pot fi generate de modulul driver, pot fi preluate dintr-un fisier sau furnizate de tester intr-o maniera interactiva.

- Rezultatele fiecarei executii sunt afisate sau scrise intr-un fisier pentru a fi comparate ulterior cu rezultatele asteptate.
- Cazurile de test in testarea unitara pot fi specificate printr-un tabel de perechi: intrari – rezultate asteptate.

Testele unitare (3)

Exemplu de modul ciot:

Modulul testat apeleaza o functie de sortare a unui vector, cu antetul:

```
void sortare(int n, int *lista);
```

Functia poate fi simulata prin urmatoarea functie ciot:

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

Modulele ciot si modele driver folosite in testele unitare se pastreaza pentru a fi folosite in **testele regresive (teste efectuate dupa corectarea unor defecte)**.

Testele unitare(4)

Metode de alegere a datelor de test pentru testarea unitara:

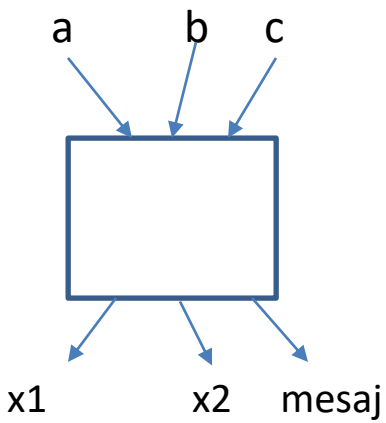
- **Testarea functionala (“black box”)**: datele de test se aleg pe baza specificatiei unitatii testate
- **Testarea structurala (“white box)**: datele de test se aleg pe baza codului unitatii testate:
  - Testarea fluxului controlului
  - Testarea fluxului datelor
- **Alte metode**:
  - **Testarea random:** 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

Alegerea datelor de test

Testarea unui program pentru rezolvarea ecuatiilor de grad 2

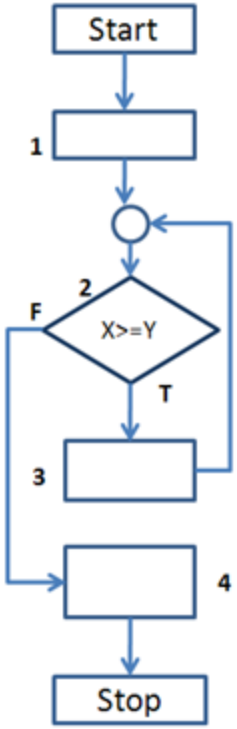
$a \cdot x^2 + b \cdot x + c = 0$

Testarea functionala
(pe baza specificatiei)



Testarea structurala
(pe baza codului)

Se aleg valorile pt a,b,c pe baza unui criteriu de acoperire a GFC.



Graful fluxului controlului (GFC)

Se aleg cel putin 2 cazuri de test:

1) Valorile pt a,b,c astfel incat sa rezulte radacini reale
   mesajul afisat : radacini reale
   valorile variabilelor x1, x2: cele 2 radacini asteptate
2) Valorile pt a,b,c a.i. sa rezulte radacini complexe
   mesajul afisat : radacini complexe
   valorile variabilelor x1, x2: valorile asteptate pt partea reala si partea imaginara

Testarea claselor (1)

- **Testarea unei clase presupune:**
  - testarea separata a fiecarei functii membru
  - testarea comportarii in timp a obiectelor clasei:

    - se poate folosi o diagrama de stari UML care reda toate starile obiectelor clasei de la creare pana la distrugere

Testarea "Alpha-Omega" – cazurile de test se aleg a.î. un obiect al clasei testate sa treaca prin toate starile sale, de la creare pana la distrugere.

- Fiecare executie de test corespunde unei căi de la nodul de intrare la nodul de iesire in graful starilor.
- Suita de teste trebuie sa acopere executia tuturor metodelor clasei, fiecare cel putin odata.
- Este o testare minimala a unei clase: este urmata de alte teste, bazate pe specificatia functionala a clasei sau pe acoperirea codului.

Testarea claselor (2)

- **Ordinea de apel a metodelor unei clase la o executie de test depinde de tipul clasei**:
  - "Non-modal" – clasa care accepta orice mesaj (apel) 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.
    - 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.
    - Prin suita de teste trebuie sa se asigure trecerea unui obiect prin toate starile si toate tranzitiile ("acoperirea" tuturor starilor si a tranzitiilor).

Platforme de testare automata a claselor:

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

Testarea de integrare (1)

Scopul : integrarea modulelor (unitati functionale/clase) dezvoltate si testate independent, intr-un sistem functional.

- Se verifica interactiunile dintre module, grupuri de module, subsisteme, pana la nivel de sistem.
- Clasele sunt testate independent, prin testele unitare. Prin testele de integrare se verifica interactiunea dintre functiile claselor prin diferite suite de test.
- Testele de integrare presupun, ca si testele unitare, realizarea de module "ciot " si module "driver". Numarul de module "driver " şi de module "ciot" necesare in testele de integrare depinde de ordinea in care sunt integrate modulele.
- Testele de integrare necesita instrumente de gestiune a versiunilor si a configuratiilor.

Metoda "big-bang“

- Sunt integrate intr-un program executabil toate modulele existente la un moment dat. Modulele "driver " şi "ciot" necesare sunt de asemenea integrate. Metoda este periculoasa caci toate defectele existente se manifesta in acelasi timp si localizarea lor este dificila.

Testarea de integrare (2)

Integrarea progresiva

- Integrarea modulelor existente in mai multi pasi
- In fiecare pas se adauga ansamblului de module integrate numai un singur modul sau 2-3 module corelate. Astfel, defectele care apar la o executie de test sunt provocate de ultimul modul integrat.

Integrarea ascendenta ("de jos in sus")

- Se incepe prin testarea modulelor simple, care nu apeleaza alte module, apoi se adauga progresiv module care apeleaza numai modulele deja integrate, pana cand este asamblat intregul sistem.
- Metoda necesita implementarea unui modul "driver" pentru fiecare modul/grup de module corelate (si nici un modul "ciot").

Avantajele integrarii de jos in sus

- Nu sunt necesare module "ciot". Modulele "driver" se implementeaza mult mai usor decat modulele "ciot". Exista chiar instrumente care produc automat module "driver".
- De regula modulele de nivel coborat sunt apelate de multe alte module:→ este mai avantajos sa fie implementate decat sa fie simulate prin module ciot.

Testele de integrare(3)

Dezavantajele integrarii de jos in sus

- Modulele principale ale programului, care controleaza fluxul aplicatiei, sunt testate la sfarsit. Descoperirea unor defecte in testarea lor, in general, conduce la reproiectare si reimplementarea unor module deja integrate.
- Un prototip timpuriu al aplicatiei nu este posibil.

Integrarea descendenta ("de sus in jos")

- Se incepe prin testarea modulului principal, care controleaza fluxul controlului in aplicatie, in pasii urmatori se adauga modulele apelate direct de el, si asa mai departe.
- **Metoda presupune implementarea unui singur modul "driver" (care apeleaza modulul principal) si a unui modul "ciot" pentru fiecare alt modul al programului.**

Testarea de integrare(4)

- Integrarea descendenta se efectueaza intr-un proces de dezvoltare descendenta.
- Fiecare modul este testat imediat dupa ce a fost implementat, moment in care nu au fost inca implementate modulele apelate de el.
- In fiecare pas este inlocuit un singur modul "ciot" cu cel real.

<u>Avantajele integrarii descendente</u>

- **Erorile de proiectare sunt descoperite timpuriu**, la inceputul procesului de integrare, atunci când sunt testate modulele principale ale programului→ se evita reproiectarea si reimplementarea majoritatii componentelor de nivel mai coborât, asa cum se intâmpla cand erorile respective sunt descoperite la sfârsitul procesului de integrare.
- **Programul obtinut este mai fiabil** caci modulele principale sunt cel mai mult testate.
- Prin integrarea modulelor de nivel superior se poate considera **ca sistemul în ansamblul său exista dintr-o faza timpurie a dezvoltarii** si deci se poate exersa cu el in vederea validarii.

Testele de integrare(5)

Dezavantajele integrarii descendente

- Este necesar sa se implementeze cate un modul "ciot" pentru fiecare modul al programului, cu exceptia modulului principal.
- Este dificil de simulat prin module "ciot" componente complexe si componente care contin in interfață structuri de date.
- In testarea componentelor principale, 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.

Testele de integrare(6)

Tehnici de integrare hibride

- Integrarea nu trebuie sa fie strict descendenta sau strict ascendenta.
- Dezavantajele integrarii descendente pot fi reduse aplicand tehnici hibride, de exemplu, folosind in locul unor module "ciot", modulele reale testate.
- Experienta arata ca este foarte util sa se inceapa prin integrarea modulelor de interfata utilizator. Aceasta permite continuarea integrarii in conditii mai bune de observare a comportarii programului.

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.

Testarea de sistem (1)

Obiectivul: se verifică dacă produsul obtinut (hardware+software) satisface specificația cerințelor; dacă este o versiune intermediară, se verifică satisfacerea cerințelor acelei versiuni.

- Sunt teste ale sistemului de programe şi echipamente. Poate fi necesar ca sistemul să fie instalat şi testat în mediul său real de operare ; dacă nu este posibil, se simuleaza mediul real.
- Adesea, testele de sistem ocupă cel mai mult timp din întreaga perioadă de testare.
- Toate testele sunt de tip «cutie neagră» (cazurile de test sunt alese pe baza specificației, fără cunoaşterea codului).
- Este efectuată de o echipă a companiei dezvoltatoare.

Testele de sistem: derivate din cerințele funcționale şi nefuncționale (fiabilitate, portabilitate, adaptabilitate, performanţă, disponibilitate, securitate, siguranta in functionare)

Teste funcționale - derivate din cerințele funcționale

- Cazurile de test se definesc plecand de la cazurile de utilizare
- Includ şi testele de interfață utilizator
- In funcție de tipul sistemului, testele funcționale pot fi combinate cu alte tipuri de teste, derivate din cerințele nefuncționale.

Testarea de sistem(2)

Teste de performanța

- Se verifica daca performanta sistemului corespunde celei cerute:
  - In conditii de operare normale (performanta nominala)
  - In conditiile cele mai defavorabile

  Parametrii de performanta masurati difera de la sistem la sistem. Exemple:

  - “timpul de raspuns trebuie sa fie sub o milisecunda, 90% din timpul operarii”
  - “timpul de raspuns la o tranzactie on-line sa fie sub o secunda”

Teste de comunicare (interfete externe)

- Se verifica fluxul datelor si al controlului prin interfetele externe
- Protocoalele de comunicatie

Teste ale utilizarii resurselor

- Utilizare CPU, memorie interna, spatiu pe disc, utilizarea retelei

Testarea de sistem(3)

Teste de fiabilitate

- Se verifica satisfacerea cerintelor de fiabilitate, de ex. MTBF (Mean Time Between Failures) – perioada de timp medie intre 2 caderi (functionari neconforme cu specificatia)
- **Se folosesc teste statistice operationale**: datele de test sunt generate aleator, conform unei legi de probabilitate care corespunde profilului operational (de utilizare) al sistemului.
- Testarea de fiabilitate se termina atunci cand rata caderilor ajunge la valoarea ceruta.

Teste de securitate

- Se verifica daca sistemul este protejat impotriva atacurilor:
  - Acces neautorizat la resursele sau functiile sistemului
  - Incercare de acces la fisierele altor utilizatori
  - Oprirea executiei proceselor rulate de alti utilizatori
  - Securitatea comunicarii in retea

Testarea de sistem(4)

Teste de portabilitate

- Se ruleaza teste selective pe toate platformele prevazute

Teste de siguranta in functionare

- Se verifica daca sistemul protejeaza utilizatorii sai impotriva daunelor provocate de caderi ale sistemului:
  - Nu se pierd fisiere deschise
  - Nu se pierd informatii din baza de date
  - Nu se pierd informatii de care depinde buna functionare a sistemului
- Se fac teste cauzand probleme, cum ar fi: intreruperea curentului electric, inchiderea calculatorului in timpul unor operatii, etc.

Testarea de sistem(5)

Teste de stres

- Sistemul este pus in situatii de operare exceptionale: rata foarte mare a intrarilor, numar de utilizatori foarte mare conectati simultan la sistem, consum maxim de resurse, trafic maximal in retea, etc.
- Se urmareste evaluarea limitelor de performanta ale sistemului.
- **Sunt teste de robustete**: sistemul nu trebuie sa se degradeze, ci sa-si mentina starea de functionare in conditii neprevazute la crearea sa.

Testarea de acceptare (1)

- **Scopul: verificarea conformitatii produsului obtinut cu produsul solicitat, conform contractului cu clientul** (Specificatia cerintelor utilizator).
- Clientul şi utilizatorii finali participă la efectuarea acestor teste.
- Uneori testele de acceptare sunt conduse de client.
- Se efectuează în mediul operațional al produsului.
- Pentru unele produse software, testarea de acceptare a versiunii finale are loc în două etape:

  Testarea alfa: se efectuează folosindu-se specificatia cerintelor utilizator, până cand clientul şi dezvoltatorul cad de acord că programul este o reprezentare satisfăcătoare a cerințelor.

  Testarea beta: programul este distribuit unor utilizatori selecționați, realizandu-se astfel testarea lui în condiții reale de utilizare → se reduce riscul căderilor software în timpul operării şi se asigură o calitate crescută a produsului software prin validarea de către utilizatorii finali.

Testarea de acceptare (2)

- Criteriul de acceptare: produsul trebuie să aibă funcționalitătile şi calitățile asteptate de client şi utilizatorii țintă.

- De ce este necesară testarea de acceptare după ce sistemul a fost deja testat pentru conformitatea cu specificatia cerintelor?

1. Documentul de specificație a cerințelor cuprinde o descriere detaliata a cerințelor viitorului produs, realizata de dezvoltator dupa extragerea cerințelor utilizator, pe baza acestor cerințe (care pot fi incluse în acelasi document).

Unele cerințe utilizator pot fi înțelese de dezvoltator în mod diferit de fața de client.

→ prin testarea de acceptare se verifică faptul că cerințele utilizator au fost corect întelese de dezvoltator iar produsul obținut satisface asteptările clientului şi ale utilizatorilor finali.

2. Este posibil ca unele schimbari ale cerințelor pe parcursul dezvoltarii să nu fi fost comunicate dezvoltatorului.

Testarea de acceptare(3)

Tipuri de teste:

- Functionalitatea: produsul trebuie sa ofere toate funcțiile prevazute în contract şi să le execute corect.
- Acuratetea: precizia rezultatelor.
- Integritatea datelor: datele dintr-o baza de date nu sunt alterate prin operatii de interogare, actualizare, restaurare.
- Recupererea datelor: se verifica daca datele pot fi recuperate dupa o cadere
  - Sunt toate datele recuperate?
  - Sunt consistente datele recuperate?

Testarea de acceptare (4)

Tipuri de teste(cont):

- Usurinta de utilizare: usurinta de invatare, de operare, calitatea interfetei utilizator, prezenta help-ului on-line, s.a.
- Usurinta de instalare si configurare
- Performanta: se verifica parametrii de performanta specificati.
- Fiabilitatea: rata caderilor in timpul testelor de acceptare
- Usurinta de intretinere si disponibilitatea sistemului: timpul mediu necesar repararii defectelor (MTTR) in urma caderilor din timpul testelor de acceptare
- Interoperabilitatea

si altele

Testele regresive

- Se numesc astfel testele executate dupa corectarea defectelor, pentru a se verifica daca in cursul corectarii nu au fost introduse alte defecte.

- Aceste teste sunt efectuate atat in timpul dezvoltarii, pe parcursul testelor, cat si in timpul perioadei de operare a sistemului.

- Pentru facilitarea lor este necesar sa se arhiveze toate cazurile de test executate in timpul dezvoltarii programului, ceea ce permite, în plus, verificarea automata a rezultatelor testelor regresive.

Lecturi suplimentare

1. Kshirasager Naik, Priyadarshi Tripathy, “Software Testing and Quality Assurance – Theory and Practice”, Publishe by Wiley John Wiley & Sons, New Jersey, 2008.
2. https://www.guru99.com/software-testing-introduction-importance.html
3. https://www.guru99.com/software-testing.html
4. https://www.tutorialspoint.com/software_testing/software_testing_types.htm

Testarea claselor în JUNIT

Prof. univ. dr. ing. Florica Moldoveanu

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

Platforme de testare unitară

- JUnit: platforma (framework) de testare unitara 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

Platformele de testare unitară reduc efortul de testare permițând:

- Definirea cazurilor de test si organizarea lor în suite de teste
- Setarea contextului de execuție al cazurilor de test (operatii înainte şi după execuția cazurilor de test)
- Execuția suitelor de teste
- Inregistrarea rezultatelor executiei suitelor de teste
- Analiza rezultatelor testelor
- Reutilizarea suitelor de teste după modificarea codului – teste de regresie

Testarea claselor în JUNIT (1)

Pentru testarea unei clase C în Junit:

- Se implementează o **clasă de test**, TestC

  - Pentru fiecare metodă M a clasei C se implementează o metodă de test, TestM, în clasa TestC

    - Metoda TestM implementează unul sau mai multe cazuri de test ale metodei M:

      - apelează metoda M cu valori alese pentru parametrii de intrare
      - compară rezultatele întoarse de metoda M cu cele aşteptate, folosind aserțiuni

- Clasa **Assert** din mediul Junit furnizează metode de definire a aserțiunilor asupra rezultatelor aşteptate la execuția unei metode a clasei testate.

- Execuția unei metode aserțiune se termină fie normal, fie printr-o excepție, numită **AssertionFailedError**. **Exemplu de metodă aserțiune:**

```
static public void assertTrue(Boolean conditie) {
    if (!conditie)
            throw new AssertionFailedError();
}
```

Testarea claselor în JUNIT (2)

- O clasa de test in Junit 3 extinde clasa **TestCase** a mediului Junit.
- Functiile din clasa de test care sunt metode de test au nume prefixate cu “test”
- In Junit 4 metodele de test sunt adnotate cu @Test (https://www.guru99.com/junit-annotations-api.html)

```
public class Calcule {
   public static int Suma(int a, int b) {
      return (a+b);
   }
}
```

Junit 3

```
import junit.framework.*;// TestCase şi Assert
import Calcule;

public class TestCalcule extends TestCase {
   public void testSuma() {
      assertEquals(15, Calcule.Suma(5, 10));
   }
}
```

Junit 4

```
import org.junit.*;
import static org.junit.Assert.*;
import Calcule;

public class TestCalcule {
   @Test
   public void CalculSuma() {
     assertEquals(15, Calcule.Suma(5, 10));
   }
}
```

Definirea claselor de test în JUNIT 4 (1)

Exemplu:

Clasa de testat:

```
public class Find {
   public static int findMax(int arr[]){
    int max=0;
     for(int i=1; i < arr.length; i++){
       if(max<arr[i])  max=arr[i];
      }
      return max;
}
    public static int findMin(int arr[]){
      int min=0;
       for(int i=1; i <arr.length; i++){
         if(min>arr[i])  min=arr[i];
        }
       return min;
     }
 }
```

Clasa de test:

```
import org.junit.*;
import static org.junit.Assert.*;
import Find;
//2 cazuri de test pentru fiecare functie membru
public class TestFind {
   @Test
   public void testFindMax(){
     assertEquals (4, Find.findMax(new int[]{1,3,4,2}));
     assertEquals (-1,
          Find.findMax(new int[]{-12,-1,-3,-4,-2}));  }
   @Test
   public void testFindMin(){
     assertEquals (1, Find.findMin(new int[]{1,3,4,2}));
     assertEquals (-12,
          Find.findMin(new int[]{-12,-1,-3,-4,-2}));  }
}
```

Executia cazurilor de test în JUNIT 4 (1)

- Se creaza o clasa **TestRunner** si se apeleaza metoda runClasses() din clasa JUnitCore cu parametru numele clasei de test. Functia intoarce un obiect de tip *Result*, care poate fi utilizat pentru a obtine rezultatele testelor. Functia getFailures din clasa Result intoarce lista testelor care au cazut (public List<Failure> **getFailures**()).

```
import org.JUnit.runner.JUnitCore;
import org.JUnit.runner.Result;
import org.JUnit.runner.notification.Failure;
public class TestRunner {
  public static void main(String[] args) {
    Result result = JUnitCore.runClasses(TestFind.class);
    System.out.println("Caderi: " + result.getFailureCount());
    for (Failure failure: result.getFailures()) //pentru fiecare cadere
            { System.out.println(failure.toString()); }
 }
}
```

Executia cazurilor de test în JUNIT 4 (2)

- Se execută programul de test cu clasa **TestRunner:** se vor executa cazurile de test definite in clasa TestFind.
- Rezultatele afisate:

  Caderi: 2

  AssertionFailedError: expected: <-1> but was:<0>

  AssertionFailedError: expected: <1> but was:<0>

  Prima excepție este produsa de assertEquals (-1, Find.findMax(new int[]{-12,-1,-3,-4,-2}));

  A 2-a excepție este produsa de assertEquals (1, Find.findMin(new int[]{1,3,4,2}));

- Se înlocuiesc inițializările:

  max=0; cu max=arr[0];

  min=0; cu min=arr[0];
- Se ruleaza din nou **TestRunner**.
- Rezultatul afisat:

  Caderi: 0

Setarea contextului de executie al cazurilor de test în JUNIT 4 (1) - test fixture -

- Poate fi necesar ca anumite operatii sa fie executate înaintea unui caz de test/ după execuția cazului de test sau înaintea tuturor cazurilor de test definite în clasa de test/după execuția tuturor cazurilor de test.
- De exemplu, daca funcția testată lucrează cu un fişier care trebuie să existe atunci când este apelată. Crearea şi distrugerea fişierului nu fac parte din funcție.
  - ➢ Sunt 2 posibilități:
  - Crearea şi distrugerea fişierului să fie incluse în funcția de test (care implementează cazul de test al funcției) – nu se recomandă, deoarece aceste operații nu fac parte din funcția testată.
  - Crearea şi distrugerea fişierului să fie implementate separat de cazul de test, prin funcții speciale de setare a contextului de execuție a cazului de test.

Setarea contextului de executie al cazurilor de test în JUNIT 4 (2)

Alte exemple:

- Crearea unei conexiuni la o bază de date/ deconectarea după execuția cazului de test
- Pornirea unui server înaintea execuției tuturor cazurilor de test/ închiderea conexiunii după. Este ineficient ca pornirea şi oprirea serverului să fie implementate în fiecare caz de test.

- Astfel de operații pot fi definite în funcții membru ale clasei de test, adnotate cu:

  @Before: operații executate înaintea fiecărui caz de test

  @After: operații executate după fiecare caz de test

  @BeforeClass: operații executate înaintea tuturor cazurilor de test

  @AfterClass: operații executate după executia tuturor cazurilor de test

Setarea contextului de executie al cazurilor de test în JUNIT 4 (3)

Exemplu:

```
public class OutputFileTest { //clasa de test
     private File output;
    @BeforeClass
      public static void alocari(){ …aloca resurse pentru toate testele…}
    @AfterClass
      public static void dealocari(){ …dealoca resursele folosite pentru toate testele…}
    @Before
      public void createOutputFile(){ output = new File (…); }
    @After
      public void deleteOutputFile() { output.delete(); }
    @Test
       public void testFile1() { ….cazul de test 1 pt lucrul cu fisierul output…}
    @Test
       public void testFile2() { ….cazul de test 2 pt lucrul cu fisierul output…}
}
```

Setarea contextului de executie al cazurilor de test în JUNIT 4 (4)

Ordinea la executie a metodelor clasei OutputFileTest:

```
alocari();
createOutputFile();
testFile1();
deleteOutputFile();
createOutputFile();
testFile2();
deleteOutputFile();
dealocari();
```

Rularea unei suite de teste în JUNIT 4

Testele (cazurile de test) definite în mai multe clase de test pot forma o suita de teste.

- Se creaza o clasa adnotata cu **@RunWith** şi **@Suite** si se adauga clasele de test la suita.
- Clasa astfel adnotata nu are cod – este doar un suport pentru adnotari.

```
import org.JUnit.runner.*;

@Runwith(Suite.class)

@Suite.SuiteClasses({test1.class, test2.class……})
 public class TestSuiteExample { } //clasa nu are cod

 public class TestRunner { //rularea testelor definite in clasele din suita (test1, test2,..)

   public static void main(String[] args) {

    Result result = JUnitCore.runClasses(TestSuiteExample.class);

    System.out.println("Caderi: " + result.getFailureCount());

    for (Failure failure: result.getFailures()) { System.out.println(failure.toString()); }
 }
}
```

Functii din clasa Assert

- ***assertEquals(Object asteptat, Object actual):*** executia metodei se termina normal daca obiectul asteptat si cel actual sunt egale conform metodei **equals()**: asteptat**.equals(** actual **) ==** true
- ***assertEquals(int asteptat, int actual):*** se termina normal daca *asteptat* **==** *actual*

  Exista o asertiune similara pentru fiecare tip primitiv: int, float, double, char, byte, long, short, Boolean şi String.
- ***assertEquals(double asteptat, double actual, double toleranţa*)**: metoda se termina normal daca valoarea absoluta a diferentei dintre valoarea asteptata si cea actuala este <= cu valoarea de toleranta; exista o asertiune similara pentru intrari float.
- ***assertSame(Object asteptat, Object actual)***: metoda se termina normal daca parametrii *asteptat* si *actual* sunt referinte la acelasi obiect din memorie.
- ***assertNull(Object testobject):*** se termina normal daca *testobject* este *null*
- ***assertFalse(Boolean conditie):*** se termina normal daca expresia *conditie* are valoarea FALSE.
- ***assertTrue(Boolean conditie):*** se termina normal daca expresia *conditie* are valoarea TRUE.

Lecturi suplimentare

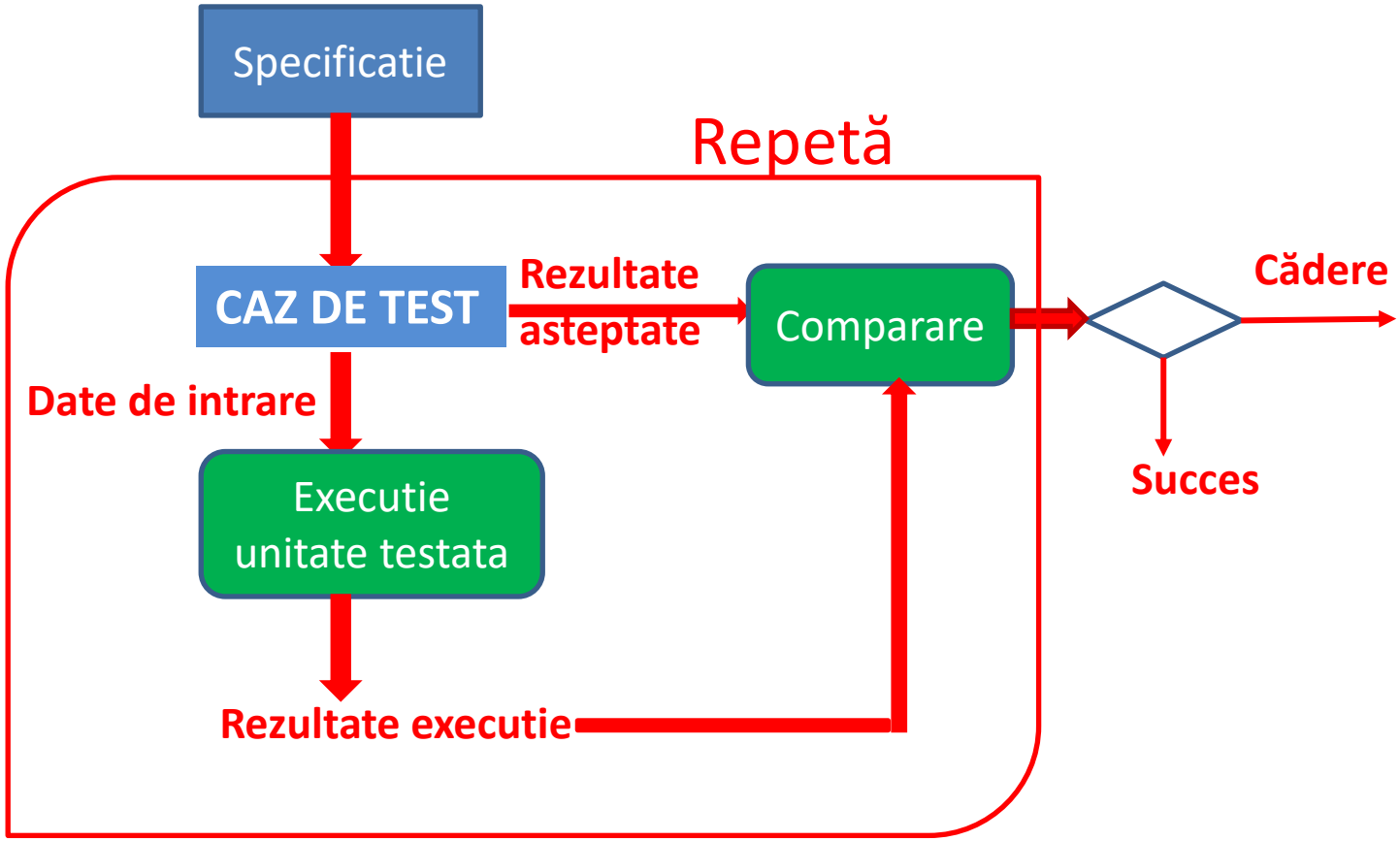
1. https://www.guru99.com/junit-annotations-api.html
2. https://junit.org/junit4/javadoc/latest/org/junit/runner/JUnitCore.html#runClasses(java.lang.Class...)
3. https://junit.org/junit4/javadoc/latest/org/junit/runner/Result.html
4. https://www.javatips.net/api/org.junit.runner.result

Alegerea cazurilor de test în testarea functionala -1

Prof. univ. dr. ing. Florica Moldoveanu

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

Testare funcțională: testare black box



Date de intrare = vector de test:

- date prin care se verifica functionarea conform specificatiei
- date prin care se pot descoperi defecte de implementare

Alegerea cazurilor de test in testarea functionala

- Bazata pe specificatia unitatii testate
- Focalizata pe verificarea comportarii sale externe
- Unitatea testata este tratata ca o «cutie neagra»: **testare «black box»**
- Este utilizata la toate nivelurile de testare: unitara, de integrare, de sistem si de acceptare

Un caz de test cuprinde un set de valori pentru variabilele de intrare ale unitatii testate impreuna cu rezultatele pe care unitatea testata trebuie sa le produca (comportarea asteptata a programului).

- Cazurile de test se aleg pentru a verifica functionarea unitatii testate conform specificatiei, dar şi pentru a descoperi defecte în implementare.

In testarea unei functii dintr-o clasa setul variabilelor de intrare/iesire include: parametrii functiei si variabilele membru ale clasei.

Setul de date de intrare folosit intr-un caz de test îl numim **vector de test**.

Tipuri de teste functionale

- **Testarea "ad hoc":** programul este executat (in mod repetat) pentru intrari alese arbitrar observandu-se comportarea sa – este principala metoda de detectare a problemelor raportate de utilizatorii finali.
- **Testarea folosind "valori speciale" din domeniile variabilelor de intrare sau de iesire:**
  - Un vector de test contine o combinatie de **valori speciale** ale variabilelor de intrare
  - **Pairwise testing** – reduce numarul de combinatii intre valorile speciale
- **Testarea bazata pe clasele de echivalenta ale variabilelor de intrare/iesire:**

  vectorii de test se aleg pe baza predicatelor definite asupra variabilelor de intrare sau iesire.
- **Teste statistice**
- **Testarea bazata pe specificatii formale**: specificatii de intrare-iesire (pre si post conditii), automate cu stari finite, s.a.

Testarea folosind “valori speciale” din domeniile variabilelor de intrare sau de iesire

Concepte cheie:

- Se identifica domeniul de valori al fiecarei variabie de intrare si de iesire a unitatii testate, analizand specificatia cerintelor si documentele de proiectare.
- Se selecteaza **valori speciale** (valori cu proprietati *importante)* din domeniul fiecărei variabile de intrare sau de iesire.
- Se proiecteaza cazuri de test considerand diferite combinatii de **valori speciale din domeniile variabilelor de intrare.**
- Se proiecteaza cazuri de test continand combinatii de valori de intrare care produc **valori speciale din domeniile variabilelor de iesire.**

Selectia valorilor *speciale* din domeniul unei variabile de intrare sau de iesire se bazeaza pe:

- Domeniul (tipul) variabilei
- Modul in care este utilizata: variabila de intrare, de iesire, de intrare-iesire

Alegerea valorilor de test (1)

- **Criterii de selectie a valorilor speciale pentru variabile de intrare**

1. **Domeniul este un set discret redus de valori** (ex. {3, -5, 10}, {True, False}, {P, Q, R})→ se efectueaza teste folosind fiecare valoare din set si valori din afara setului (intrari nevalide)

2. **Domeniul este compus din unul sau mai multe segmente continue de valori**:

Exemple: [0..1000]; [-1, 1] si [10, 100]; [val. minima- val. maxima a unui intreg pe 16 biti];

→ datele de test (**valorile speciale**) se selecteaza astfel:

- Valoarea minima a fiecarui segment
- Valoarea maxima a fiecarui segment
- O valoare reprezentativa din interiorul fiecarui segment
- Valori cu proprietati matematice speciale; ex: 0, 1, cea mai mica/ mare valoare reala
- Valori din afara fiecarui segment (intrari nevalide)

3. **Variabila de tip String** – se fac teste cu: şir vid, şir cu un singur caracter, şir cu numarul maxim de caractere, şir cu un numar intermediar de caractere, şir cu numar de caractere mai mare decat cel admis. Șirurile de test vor contine caractere speciale.

Alegerea valorilor de test (2)

- **Criterii de selectie a valorilor speciale pentru variabile de iesire**

  1. **Domeniul este un set discret de valori**

  → **set redus:** se efectueaza teste care conduc la obtinerea fiecarei valori din set;

  → **set mare**: se executa un numar de teste care conduc la obtinerea unui numar mare de valori din domeniul de iesire

  2. **Domeniul este compus din unul sau mai multe segmente continue de valori** → cazurile de test se proiecteaza selectand:

  – Valori de intrare care produc valorile minime ale segmentelor
  – Valori de intrare care produc valorile maxime ale segmentelor
  – Valori de intrare care produc cateva date din interiorul fiecarui segment

- **Criterii de selectie pentru variabile de intrare- iesire** (variabile de intrare in care se regasesc iesiri): **teste suplimentare**

  – Un caz de test care produce o valoare de iesire a variabilei diferita de valoarea sa de intrare
  – Un caz de test in care valoarea de iesire a variabilei este identica cu valoarea sa de intrare

Alegerea valorilor de test (3)

Variabilă de intrare tablou

- Un tablou poate avea una sau mai multe dimensiuni. Toate elementele unui tablou sunt de acelasi tip.
- Cazurile de test trebuie sa includa teste cu diferite dimensiuni ale tabloului (daca dimensiunea este variabila de intrare), in diferite combinatii daca este multi-dimensional: tablou vid, tablou cu dimensiunea minima, tablou cu dimensiunea maxima, tablou cu o dimensiune intermediara, tablou cu prima dimensiune minima si a 2-a dimensiune maxima, s.a.
- Cazurile de test trebuie sa includa tablouri cu valori speciale, alese in functie de tipul elementelor sale.
- Aceste criterii generale pot fi combinate cu unele euristici, ținand cont de felul in care este folosit tabloul.
- In unele cazuri, sunt necesare teste pe substructuri ale unui tablou multi-dimensional (pt. un tablou 2D: elementele diagonalei, matricea triunghiulara inferioara si cea superioara, randuri, coloane).

Alegerea valorilor de test (4)

Variabila de intrare tablou – exemplu.

Fie o functie C de cautare a unui numar intreg intr-un tablou de numere intregi, specificata astfel

int cauta (int X[], int n, int y);

Functia cauta numarul y in tabloul X de n elemente; daca este gasit intoarce indexul primei sale aparitii in tablou altfel intoarce -1.

Setul minim de cazuri de test trebuie sa verifice functia in urmatoarele situatii:

1) tablou vid (n=0) ; functia trebuie sa intoarca -1;

2) tablou cu un singur element (n=1): cel mai mic intreg, cel mai mare intreg, un numar intreg oarecare

3) tablou cu numar maxim de elemente (daca este specificat): valori diverse

4) tablou cu un numar intermediar de elemente: valori diverse

Alegerea valorilor de test (5)

Pentru cazurile (2), (3) si (4) se vor face teste in care:

- numarul y exista in tablou; functia intoarce indexul sau in tablou;
- numarul y nu exista in tablou; functia intoarce -1;

Dupa fiecare apel al functiei se va afisa tabloul X pentru a se verifica faptul ca nu a fost modificat.

➢ Setul de cazuri de test se imbogateste cu teste alese pe **criterii euristice**

Euristici de testare pentru programe de cautare:

- programele de cautare produc erori atunci cand elementul cautat este primul sau ultimul in lista;
- de multe ori programatorii neglijeaza situatiile in care colectia prelucrata in program are un numar de elemente neobisnuit, de exemplu zero sau unu;
- uneori, programele de cautare se comporta diferit in cazurile: numar de elemente din colectie par, respectiv numar de elemente din colectie impar; de aceea trebuie testate ambele situatii

Alegerea valorilor de test (6)

Tinand cont de aceste euristici, cazurile de test trebuie sa verifice functia in urmatoarele situatii:

1) tablou vid (n=0)

2) tablou cu un singur element (n=1)

   a) valoarea parametrului y exista in tablou

   b) valoarea parametrului y nu exista in tablou

3) tablou cu numar par de elemente (n este un numar par)

   a) Valoarea lui y este prima in tablou (indexul 0)

   b) Valoarea lui y este ultima in tablou (indexul (n-1))

   c) Valoarea lui y este intr-o pozitie oarecare a tabloului

   d) Valoarea lui y nu exista in tablou

4) tablou cu numar impar de elemente (n este un numar impar)

   a,b,c,d ca la punctul 3

In teste se vor folosi numere intregi cu valori extreme si valori normale.

Alegerea valorilor de test (7)

Exemplu de suita de test:

1) intrari: n=0; orice tablou; y orice numar
   rezultate : valoarea functiei = -1, tabloul nemodificat

2) intrari: n=1; x[0] = 10; y = 10
   rezultate : valoarea functiei = 0, tabloul nemodificat: x[0]= 10

3) intrari: n=1; x[0]= 10; y = 15
   rezultate : valoarea functiei = -1, tabloul nemodificat: x[0] = 10

4) intrari: n=2; x[0] = 10, x[1] = 20; y= 10
   rezultate: valoarea functiei = 0, tabloul nemodificat: x[0] = 10, x[1] = 20

5) intrari: n=2; x[0] = 10, x[1] = 20; y= 20
   rezultate: valoarea functiei = 1, tabloul nemodificat: x[0] = 10, x[1] = 20

Alegerea valorilor de test (8)

6) intrari: n=2; x[0] = 10, x[1] = 20; y= -1

rezultate: valoarea functiei = - 1, tabloul nemodificat: x[0] = 10, x[1] = 20

7) intrari: n=4; x[0] = -32677, x[1] = 0; x[2] = 32677, x[3] = -1, y= -1

rezultate: valoarea functiei = 3, tabloul nemodificat

8) intrari: n=6; x[0] = 32677, x[1] = -70; x[2] = 32677, x[3] = -1, x[4] =0; x[5] = 100, y= 32677

rezultate: valoarea functiei = 0, tabloul nemodificat

In mod similar se vor acoperi cazurile: n impar mare, cu valori diverse si elementul cautat in prima, ultima si o pozitie intermediara.

Alegerea valorilor de test (9)

Concluzii:

Alegerea cazurilor de test prin analiza domeniilor variabilelor de intrare:

- Conduce la prea multe cazuri de test: **vectorii de test** sunt combinatii ale valorilor selectate ale variabilelor de intrare (numarul de CT = nr de combinatii alese).
- Generarea iesirii asteptate pentru un vector de test este relativ simpla: proiectantul testului calculeaza iesirea asteptata analizand specificatia programului.

Alegerea cazurilor de test prin analiza domeniilor variabilelor de iesire:

- Numarul de cazuri de test obtinute este mai mic – nu este necesar sa se considere combinatii intre valorile speciale ale variabilelor de iesire
- Generarea unui vector de intrare necesar pentru a produce o valoare de iesire/combinatie de valori de iesire necesita analiza specificatiei in directia inversa.

Numarul de cazuri de test determinate prin analiza domeniilor variabilelor de intrare si de iesire este prea mare → **poate fi redus prin diferite metode**.

Pairwise testing(1)

Metoda produce un numar de CT mult mai mic decat in cazul folosirii tuturor combinatiilor intre valorile selectate.

Asigura că fiecare combinatie posibila intre valorile selectate pentru fiecare pereche de variabile de intrare este acoperita prin cel putin un test.

Studii empirice au aratat ca metoda poate conduce la detectarea a circa 70% din defecte folosind cazurile de test generate

Exemplu:

Fie un sistem cu 3 intrari, X, Y, Z, pentru care s-au selectat valorile:

X= {True, False}, Y= {1,100}, Z = {Q,R}

Numarul total de combinatii (vectori de intrare) este: 2*2*2 = 8. Numarul de CT generate prin Pairwise Testing este 4.

Caz de test	Intrare X	Intrare Y	Intrare Z
CT1	True	1	Q
CT2	True	100	R
CT3	False	100	Q
CT4	False	1	R

Pairwise testing(2) – tablouri ortogonale

Notatie: L_{Runs} ($Levels^{Factors}$)

Factors: numarul de variabile de intrare;
Levels: nr maxim de valori distincte ale fiecarei variabile
Runs: numarul de cazuri de test

$L_4(2^3)$ **Variabile de intrare**

Runs	1	2	3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

Tabloul ortogonal pentru 3 variabile, fiecare cu 2 valori posibile

X= {true, false},
Y= {1,100},
Z = {Q,R}

Maparea valorilor variabilelor pe niveluri →

Cazuri de test	X	Y	Z
1	true	1	Q
2	true	100	R
3	false	1	R
4	false	100	Q

➢ 4 cazuri de test in loc de 2x2x2 =8!

- Valorile distincte ale variabilelor de intrare se mapeaza pe niveluri: [1-levels].

Pairwise testing(3) – tablouri ortogonale(cont)

$L_9(3^4)$ **Factors**

Runs	1	2	3	4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Tabloul ortogonal pentru 4 variabile, fiecare cu cel mult 3 valori posibile

A= { -1, 100, 200}
B = {a, b, c}
C = {-0.1, 0.1, 0}
D= { true, false}

Maparea valorilor variabilelor pe niveluri →

Cazuri de test	A	B	C	D
1	-1	a	-0.1	true
2	-1	b	0.1	false
3	-1	c	0	true
4	100	a	0.1	false
5	100	b	0	true
6	100	c	-0.1	false
7	200	a	0	false
8	200	b	-0.1	true
9	200	c	0.1	true

- 9 cazuri de test in loc de 3x3x3x3 = 81

- Daca o variabila are un numar de valori distincte <levels, valorile sale vor fi mapate pe mai multe niveluri.

Pairwise testing(4)

Tablourile se numesc "ortogonale" (termen imprumutat din matematica) deoarece:

- fiecare combinatie posibila intre valorile a doua variabile apare o singura data;
- fiecare vector de test (linie a tabloului) este diferit de ceilalti – furnizeaza o informatie unica;
- fiecare vector de test este independent statistic de ceilalti (aparitia unuia nu influenteaza aparitia celorlalti) – corelatia dintre ei este nula.

Metoda "pairwise testing" este eficienta:

- Cazurile de test se obtin simplu:
  - tablourile ortogonale se genereaza automat
  - maparea valorilor variabilelor pe niveluri se poate face automat
- Este o metoda statistica sistematica de testare a interactiunilor intre perechi de variabile
- Furnizeaza o acoperire uniforma reprezentativa a tuturor combinatiilor intre perechi de variabile
- S-a dovedit foarte utila in testele de integrare, testele de interfata utilizator, testarea de sistem.

Lecturi suplimentare

1. https://www.softwaretestinghelp.com/what-is-pairwise-testing/
2. http://www.testingeducation.org/wtst5/PairwisePNSQC2004.pdf
3. https://blog.issart.com/real-life-examples-of-how-to-use-pairwise-technique-in-test-design/

Alegerea cazurilor de test în testarea functionala -2

Prof. univ. dr. ing. Florica Moldoveanu

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

Testarea bazata pe clase de echivalență (1)

- O clasă de echivalență a domeniului de intrare al unității testate este alcatuită din toți vectorii de test care produc aceeaşi comportare a unității testate (funcție/program).
- Clasele de echivalență rezultă din predicatele definite asupra valorilor variabilelor de intrare.

Testarea bazata pe clase de echivalență (Equivalence class testing):

- Pentru testarea minimală este suficient sa se folosească un singur caz de test pentru fiecare clasă de echivalență → Numarul de cazuri de test este mai mic decat în cazul folosirii tuturor combinatiilor între valorile selectate din domeniile variabilelor de intrare.

Testarea valorilor de frontieră (**Boundary Value Testing)**

- Se aleg cazuri de test cu valori la frontierele claselor de echivalență ale variabilelor de intrare: pe frontiere, langa frontiere în afara clasei de echivalență şi în interiorul clasei de echivalenta.

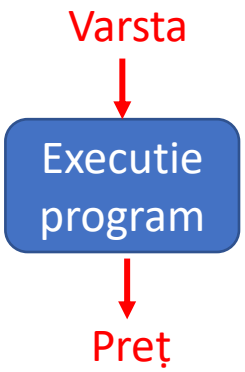
- Permite detectarea cazurilor de implementare eronata a conditiilor de frontiera (predicate gresite)

- Considerarea CT cu valori de frontiera creşte numărul de CT pentru fiecare clasă de echivalență.

Testarea bazata pe clase de echivalenta (2)

1. Program/unitate program cu o singura variabila de intrare

Unitatea testata calculeaza pretul unei calatorii cu autobuzul astfel:
0 lei pentru copii cu varsta sub 7 ani
2 lei pentru tineri cu varsta intre 7 si 18 ani
5 lei pentru adulti (19-64)
3 lei pentru cei cu varsta peste 64



Clasele de echivalenta ale domeniului de intrare:
CE1. 0<= varsta < 7
CE2. 7<= varsta < 19
CE3. 19<= varsta < 65
CE4. 65<= varsta < 101

Cazuri de test – testarea minimala (normala):

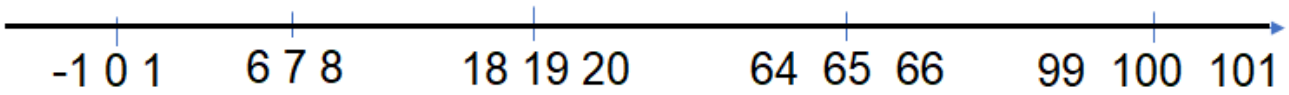
CT1. varsta=4; RA (rezultat asteptat): pret = 0
CT2. varsta=10; RA: pret = 2
CT3. varsta=30; RA: pret = 5
CT4. varsta=70; RA: pret = 3

Cazuri de test – testarea minimala robusta:

Se adauga cazuri de test cu valori de intrare nevalide.
CT5. varsta=-1; RA: mesaj “varsta nevalida”
CT6. varsta=110; RA: mesaj “varsta nevalida”

Testarea bazata pe clase de echivalenta (3)

Cazuri de test – testarea cu valori la frontierele claselor de echivalenta:



CTF1. varsta=0; RA: pret = 0 CTF2. varsta = -1; RA: “varsta nevalida” CTF3. varsta=1; RA: pret = 0

CTF4. varsta=6, RA: pret = 0 CTF5. varsta=7; RA: pret = 2 CTF6: varsta=8, RA: pret = 2

CTF7. varsta=18; RA: pret = 2 CTF8: varsta=19; RA: pret = 5 CTF9: varsta=20; RA: pret = 5

CTF10. varsta=64; RA: pret = 5 CTF11: varsta=65; RA: pret = 3 CTF12: varsta=66; RA: pret = 3

CTF13. varsta=99; RA: pret = 3 CTF14. varsta=100; RA: pret = 3

CTF15. varsta = 101; RA: “varsta nevalida”

Testarea bazata pe clase de echivalenta (4)

2. Program cu doua variabile de intrare

10<= i1 <= 100, cu subintervalele: [10, 20), [20, 70), [70, 100] → 3 clase de echivalenta

20<= i2 <= 50, cu subintervalele: [20, 30) şi [30, 50] → 2 clase de echivalenta

- Comportarea programului depinde de clasele de echivalență din care fac parte valorile variabilelor de intrare. Clasele de echivalență ale domeniului de intrare (DI) al programului:

CE1: 10<=i1<=20 şi 20<=i2<=30; CE2: 10<=i1<=20 şi 30<=i2<=50;

CE3: 20<=i1<=70 şi 20<=i2<=30; ……

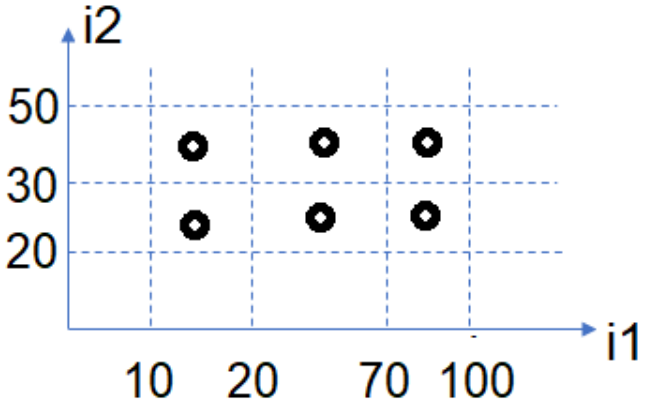
Cazuri de test – testarea minimală:

Un caz de test pentru fiecare clasă de echivalență a DI al programului:

CT1. i1=15, i2=25 CT2. i1=17, i2=40

CT3. i1 = 50, i2 = 23 CT4. i1=60, i2= 45

CT5. i1 = 80, i2 = 26 CT6. i1=90, i2=35



Testarea bazata pe clase de echivalenta (5)

Cazuri de test – testarea minimala robustă:

Se adaugă cazuri de test cu valori din afara domeniilor variabilelor de intrare

CT1. i1=15, i2=25 CT2. i1=17, i2=40

CT3. i1 = 50, i2 = 23 CT4. i1=60, i2= 45

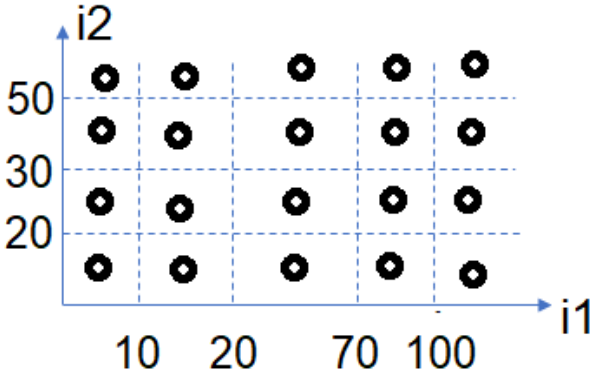
CT5. i1 = 80, i2 = 26 CT6. i1=90, i2=35

CT7. i1 = 5, i2 = 10 CT8. i1= 3, i2 = 24

CT9. i1 = 4, i2 = 36 CT10. i1= 1, i2 = 55

……

CT19. i1=110, i2 = 3 CT20. i1=120, i2 = 60



Cazuri de test – testarea extinsă

- Mai multe cazuri de test pentru fiecare clasă de echivalență

Testarea bazata pe clase de echivalenta (6)

3. Triunghiul

Unitatea testata primeste 3 numere, a, b, c, si verifica daca ele pot reprezenta lungimile

laturilor unui triunghi; a,b,c au valori in intervalul (0,200]. Rezultatul este unul dintre mesajele:

- triunghi echilateral/ isoscel/oarecare, nu este un triunghi

De asemenea, se afiseaza mesaje specifice atunci cand valorile variabilelor de intrare sunt in afara domeniului.

Clasele de echivalenta ale DI al programului sunt determinate de urmatoarele predicate:

P1: a>0 && b>0 && c>0 && a<=200 && b<=200 && c<=200 - domeniul variabilelor de intrare
P2: a+b >c && a+c > b && b+c > a - conditia ca a,b,c sa reprezinte laturile unui triunghi
P3: a==b && b==c – triunghi echilateral
P4: a==b && b!=c || a==c && c!= b || b==c && c!=a - triunghi isoscel
P5: a!=b && a!=c && b!=c - triunghi oarecare

Clasele de echivalenta - toate combinatiile de valori a,b,c care satisfac unul dintre predicatele

CE1. P1 && P3: triunghi echilateral
CE2. P1 && P2 && P4: triunghi isoscel
CE3. P1 && P2 && P5: triunghi oarecare
CE4. P1 && !P2: nu este un triunghi

Testarea bazata pe clase de echivalenta (7)

Cazuri de test – testarea minimala: un caz de test pentru fiecare clasa de echivalenta

CT	a	b	c	Rezultat asteptat
1	10	10	10	Triunghi echilateral
2	5	5	2	Triunghi isoscel
3	7	6	9	Triunghi oarecare
4	4	2	1	Nu este un triunghi

Cazuri de test – testarea minimala robusta:
Se adauga cazuri de test cu valori in afara domeniilor variabilelor a, b, c:

CT	a	b	c	Rezultat asteptat
5	-1	10	10	Valoare nevalida pentru a
6	5	-1	2	Valoare nevalida pentru b
7	7	6	0	Valoare nevalida pentru c
8	201	4	1	Valoare nevalida pentru a
9	12	201	10	Valoare nevalida pentru b

Testarea bazata pe clase de echivalenta (8)

Cazuri de test – testarea minimala robusta:

CT	a	b	c	Rezultat asteptat
10	1	40	201	Valoare nevalida pentru c
11	0	0	2	Valoare nevalida pentru a si b
12	-1	6	0	Valoare nevalida pentru a si c
13	10	-1	-1	Valoare nevalida pentru b si c
14	0	0	0	Valoare nevalida pentru a, b si c

Cazuri de test – testarea extinsa: mai multe cazuri de test pentru fiecare clasa de echivalenta

CT	a	b	c	Rezultat asteptat
1	1	1	1	Triunghi echilateral
2	100	100	100	Triunghi echilateral
3	200	200	200	Triunghi echilateral

Testarea bazata pe clase de echivalenta (9)

Cazuri de test – testarea extinsa:

CT	a	B	c	Rezultat asteptat
4	50	50	100	Triunghi isoscel
5	70	80	70	Triunghi isoscel
6	10	40	40	Triunghi isoscel
7	2	3	4	Triunghi oarecare
8	60	20	50	Triunghi oarecare
9	199	200	198	Triunghi oarecare
10	24	1	25	Nu este un triunghi
11	2	1	1	Nu este un triunghi
12	199	1	198	Nu este un triunghi

Cazuri de test – testarea extinsa robusta: se adauga cazuri de test cu valori nevalide pentru a, b, c.

Testarea bazata pe clase de echivalenta (10)

Metoda partitionarii in clase de echivalenta (CE) poate fi extinsa si asupra variabilelor de iesire:

pentru fiecare CE a domeniului de iesire se creaza un caz de test cu date de intrare care produc un rezultat apartinand unei CE de iesire.

Exemplu: Fie un program care trebuie sa produca intre 3 si 6 numere cuprinse intre 1000 si 2500.

Atunci, se vor alege intrari astfel incat iesirea programului sa fie:

- 3 numere egale cu 1000
- 3 numere egale cu 2500
- 6 numere egale cu 1000
- 6 numere egale cu 2500
- intre 3 si 6 numere cuprinse intre 1000 si 2500
- rezultat eronat: mai putin de 3 numere sau mai mult de 6 numere sau valori in afara intervalului [1000..2500]

In alegerea cazurilor de test trebuie sa se elimine redundanţele rezultate din considerarea atat a claselor de echivalenta de intrare cat si a celor de iesire.

Testarea bazata pe clase de echivalenta (11)

4. Testarea unei functii pentru rezolvarea ecuatiei de gradul 2

Specificația funcției:

void ec_grad2(float a, float b, float c)

Funcția rezolvă ecuația de gradul 2: $a \cdot x^2 + b \cdot x + c = 0$

1) $a \neq 0$ && $b \neq 0$ && $c \neq 0$

2) Dacă $b^2 - 4ac > 0$ atunci funcția afişează mesajul “rădăcini reale” şi valorile celor 2 rădăcini reale, sub forma: “x1=” valoare , “x2=” valoare.

3) Dacă $b^2 - 4ac = 0$ atunci funcția afişează mesajul “rădăcini confundate” şi valoarea celor 2 rădăcini reale, sub forma: “x1 = x2=“ valoare.

4) Dacă $b^2 - 4ac < 0$ atunci funcția afişează mesajul “rădăcini complexe” şi “partea reala =” valoarea părții reale, “partea imaginara =” valoarea părții imaginare.

Testarea bazata pe clase de echivalenta (12)

P1: $a \neq 0$ && $b \neq 0$ && $c \neq 0$

P2: $b^2 - 4ac > 0$

P3: $b^2 - 4ac = 0$

P4: $b^2 - 4ac < 0$

Clasele de echivalență ale domeniului de intrare:

CE1. P1&&P2

CE2. P1&&P3

CE3. P1&&P4

Cazuri de test – testarea minimala: un caz de test pentru fiecare clasa de echivalenta

CT	a	b	c	Rezultat asteptat
1	2	5	-3	Radacini reale: x1=0.5; x2=-3
2	5	20	32	Radacini complexe; partea reala=-2; partea imaginara= 1.54919
3	1	4	4	Radacini confundate; x1=x2= -2

Testarea bazata pe clase de echivalenta (13)

Cazuri de test – testarea minimala robusta:
Se adauga cazuri de test cu valori in afara domeniilor variabilelor a, b, c:

CT	a	b	c	Rezultat asteptat
1	0	10	-1	Valoare nevalida pentru a
2	2	0	3	Valoare nevalida pentru b
3	5	4	0	Valoare nevalida pentru c
4	0	0	5	Valoare nevalida pentru a si b
5	0	10	0	Valoare nevalida pentru a si c
6	1	0	0	Valoare nevalida pentru b si c

Testarea bazata pe clase de echivalenta (14)

Avantajele testarii bazate pe clase de echivalenta:

- Numarul de cazuri de test necesare pentru acoperirea unui domeniu larg de intrari este in general mic.
- Este o metoda de testare sistematica.
- Probabilitatea de descoperire a defectelor este mai mare decat in cazul alegerii aleatoare a unei suite de teste de aceeasi dimensiune.

Testele statistice (1)

❖ **Se efectueaza în timpul testării de sistem**

❖ Sunt, ca şi testele funcționale, teste “black box”.

❖ **Sunt teste în care datele de intrare se aleg aleator**, după o lege de probabilitate care poate fi:

- uniformă pe domeniul datelor de intrare al programului testat - aceste teste se mai numesc teste statistice uniforme;
- similară cu distribuţia datelor de intrare estimate pentru exploatarea programului – aceste teste se mai numesc teste statistice operaţionale.

- **Rezultatele experimentale ale testelor statistice uniforme sunt inegale**. Pentru unele programe s-au dovedit foarte eficiente, ele conducând la descoperirea unor defecte importante. Pentru altele s-au dovedit ineficiente.

Testele statistice(2)

Testele statistice uniforme

- Asigură o bună acoperire a cailor de executie ale programului daca domeniile de intrare ale căilor de execuţie au o probabilitate asemănătoare.
- Nu acoperă căile care corespund tratării excepţiilor. De aceea trebuie completate cu teste folosind date de intrare în afara domeniului intrărilor.

Testele statistice operaţionale sunt în general <u>teste de fiabilitate</u>.

- Prin ele se urmăreşte nu numai descoperirea de defecte ci şi comportarea programului în timp.
- Căderile observate în timpul acestor teste permit estimarea măsurilor de fiabilitate, cum ar fi MTBF (Mean Time Between Failures).

Lecturi suplimentare

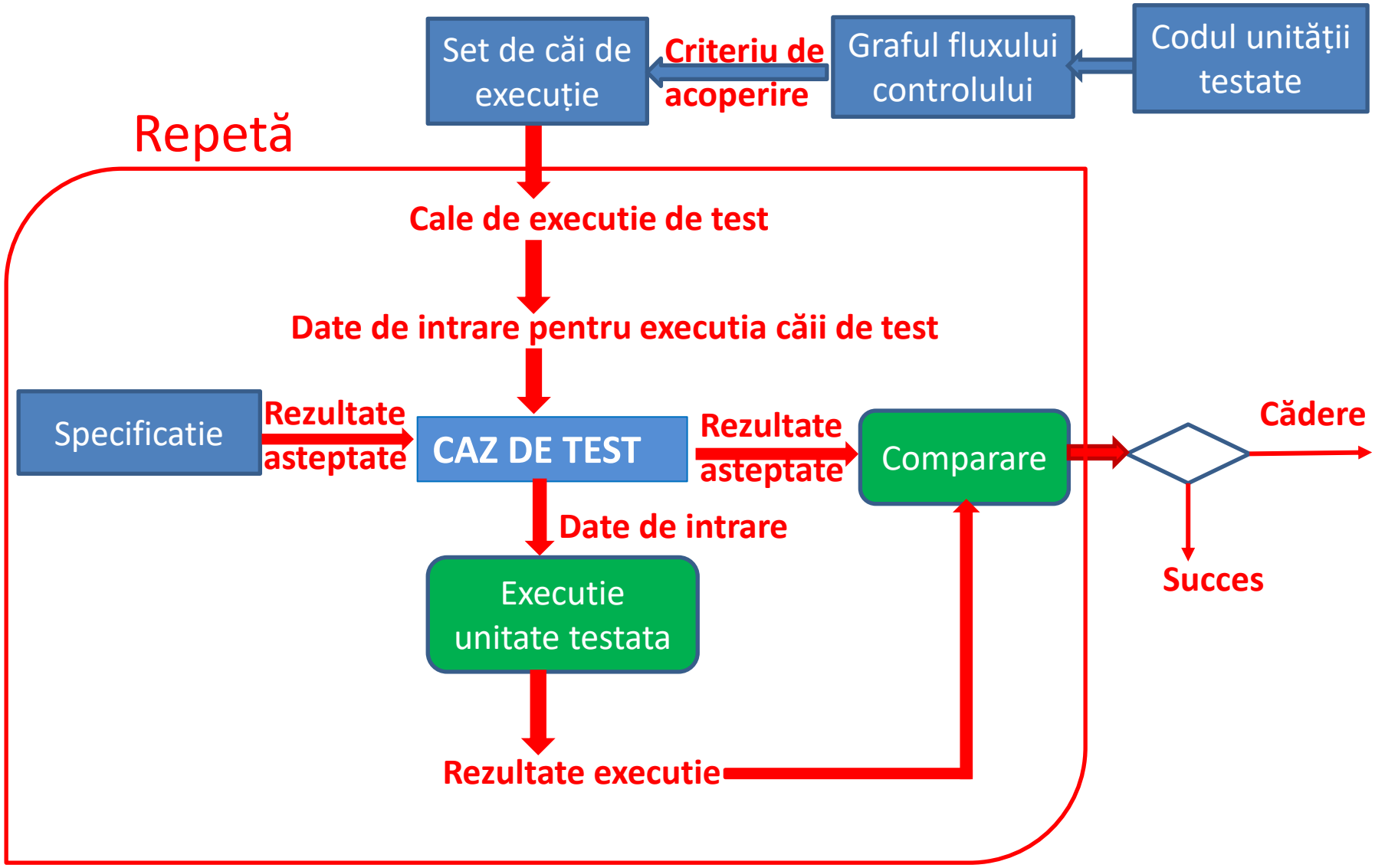
1. https://www.guru99.com/equivalence-partitioning-boundary-value-analysis.html
2. https://www.geeksforgeeks.org/equivalence-partitioning-method/
3. https://hackernoon.com/equivalence-class-partitioning-and-boundary-value-analysis-in-black-box-testing-znh3umt

Alegerea cazurilor de test în testarea structurala

Prof. univ. dr. ing. Florica Moldoveanu

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

Testare structurala: testare white box



Testarea structurală
- white box testing -

- Focalizată pe implementarea unității testate – datele de test se aleg pe baza codului

  Unitatea testată este tratata ca o “cutie transparenta”(white box).

- Se foloseşte în testarea unitară, în testarea de integrare şi în testarea programelor mici.

- Vectorii de test se aleg a.î. prin executiile de test sa fie traversate caile executabile ale unitatii testate: testarea structurala permite evidenţierea următoarelor tipuri de defecte:

1. Căi absente in graful fluxului controlului - ca urmare a ignorării unor condiţii (de exemplu: test de împărţire la zero);
2. Căi definite incorect sau incomplet (datorita unor conditii de ramificare gresite)
3. Acţiuni incorecte sau absente; de exemplu: calculul unei valori cu o metodă incorectă, neatribuirea unei valori unei anumite variabile, apelul unei proceduri cu o listă de argumente incorectă, etc.

Dintre acestea, cel mai simplu de depistat sunt greselile de tip 3. Pentru descoperirea lor este suficient să se asigure execuţia tuturor instrucţiunilor componentei testate.

Graful Fluxului Controlului

Graful fluxului controlului - GFC (Control Flow Graph), pe scurt **Graful controlului** unitatii testate este principalul instrument în testarea structurală.

- Se determina pe baza codului unitatii testate.
- Are trei **tipuri de noduri** :
  - **Noduri de prelucrare**, reprezentand "**blocuri de instrucţiuni indivizibile maximale"** : porţiuni liniare maximale de instrucţiuni care se execută întotdeauna în aceeaşi secvenţă
  - **Noduri de decizie**, care corespund instructiunilor de decizie (if, conditia de repetare sau iesire dintr-un ciclu).
  - **Noduri de jonctiune** a mai multor ramificatii (noduri ne-etichetate)
- **Arcele** reprezintă transferul controlului între noduri.
- **Graful controlului are un nod unic de intrare şi un nod unic de ieşire**. Nodul de intrare are gradul de intrare zero, iar cel de ieşire are gradul de ieşire zero.
- Daca în unitatea testata exista mai multe puncte de iesire, acestea se reunesc prin arce intr-un nod de jonctiune din care se ajunge direct in nodul de iesire.

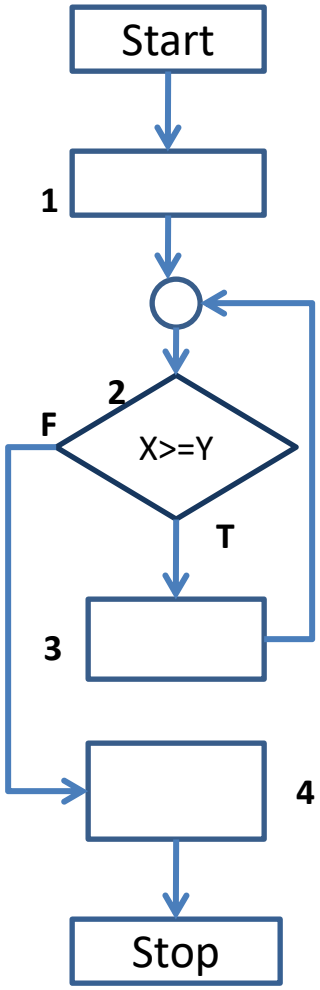
Graful fluxului controlului unei unitati program

EXEMPLU

```
f1=fopen(….);
f2=fopen(….);
fscanf(f1, ”%f”, x);
fscanf(f1, “%f”, y);
z=0;
while(x>=y)
{ x-=y;
  z++;
}
fprintf(f2, “%d”, z);
fclose(f1);
fclose(f2);
```

Codul este decupat în următoarele blocuri:

```
1. f1=fopen(….);
   f2=fopen(….);
   fscanf(f1, ”%f, x);
   fscanf(f1, “%f, y);
   z=0
2. while(x>=y)
3. x-=y;
   z++;
4. fprintf(f2, “%d”, z);
   fclose(f1);
   fclose(f2);
```



Graful controlului

Căi în graful fluxului controlului

- **O cale în graful controlului** este o secventa de noduri de prelucrare si de decizie, care începe cu nodul de intrare şi se termină cu nodul de ieşire.

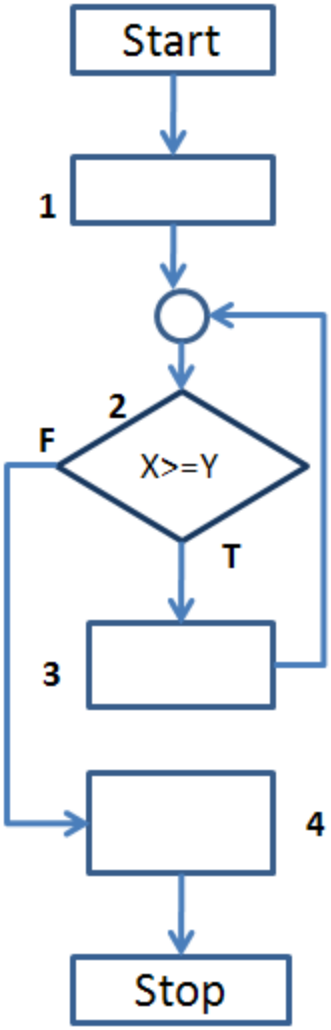
  Exemple:

  (1, 2(F), 4); (1, 2(T), 3, 2(F), 4); (1, 2(T), 3, 2(T), 3, 2(F), 4); etc.

- **Cale executabilă**: cale executată cel putin pentru un vector de intrare (Vector de intrare: asignare de valori variabilelor de intrare).

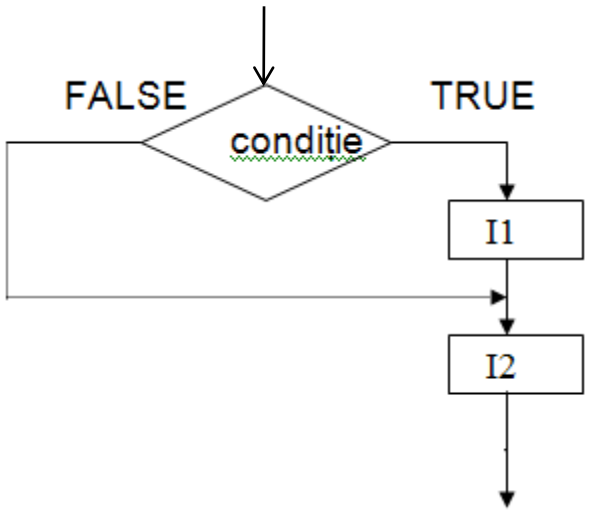
- **Scopul testarii structurale** : ideal→ parcurgerea tuturor cailor executabile prin executiile de test → in general, imposibil din cauza prezenţei ciclurilor.

- **Alegerea căilor de testat**: pe baza unor **criterii de acoperire a grafului controlului, care asigură descoperirea tipurilor de defecte mentionate.**



Acoperirea tuturor nodurilor grafului controlului

Acoperirea tuturor nodurilor= executia tuturor instructiunilor



- Conform criteriului este suficient sa se testeze calea care contine nodul de decizie si cele 2 noduri de prelucrare.
- Criteriul nu impune testarea caii care contine ramificatia FALSE a conditiei.

→Programatorul nu va verifica rezultatul programului in acest caz.

Se recomanda alegerea mai multor cai de test, chiar daca toate nodurile grafului pot fi incluse intr-o singura cale:

- Cea mai scurta cale
- Alte câteva cai, de lungime crescândă

- **Este un criteriu de testare minimal. Nu este satisfăcător, în cele mai multe cazuri.**

Acoperirea tuturor ramificatiilor nodurilor de decizie ale grafului controlului (1)

- Criteriul impune ca prin executiile de test sa se traverseze toate ramificatiile nodurilor de decizie.

- Conform acestui criteriu se va putea verifica dacă transferul controlului este corect pentru valoarea FALSE a condiţiei, în exemplul anterior.

➢ Este posibil ca o singura cale executabila sa acopere toate arcele. Exemplu:

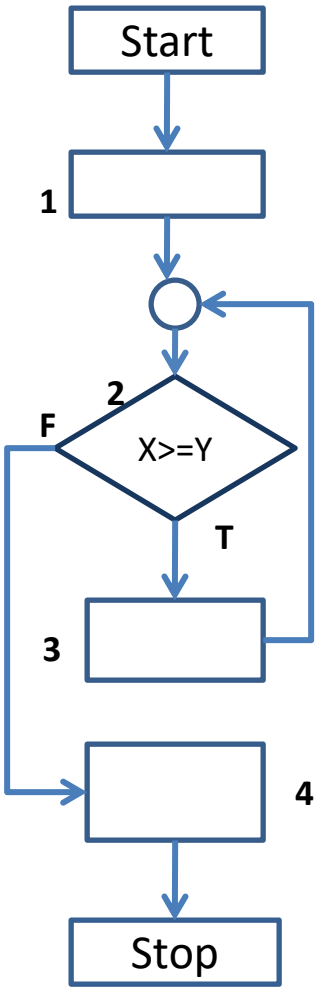
(1, 2(T), 3, 2(F), 4) → corespunde unei singure iteratii a ciclului

➢ In prezenta ciclurilor:

Cate cai se vor alege?

Cat de lungi?

Exemple: (1, 2(T), 3, 2(F), 4); (1, 2(T), 3, 2(T), 3, 2(F), 4), etc.



Acoperirea tuturor ramificatiilor nodurilor de decizie ale grafului controlului (2)

Testarea ciclurilor: recomandari empirice

- **Se alege o cale care nu traverseaza ciclul (zero iteratii)**. Testul poate evidentia greseli de initializare a ciclului, sau in calcule efectuate la iesirea din ciclu care se bazeaza pe executia ciclului.
- **Se alege o cale care contine o singura iteratie a ciclului.** Testul poate evidentia greseli de initializare a ciclului.
- **Se alege o cale care contine 2 iteratii ale ciclului**. Testul poate evidentia greseli care impiedica repetarea ciclului.

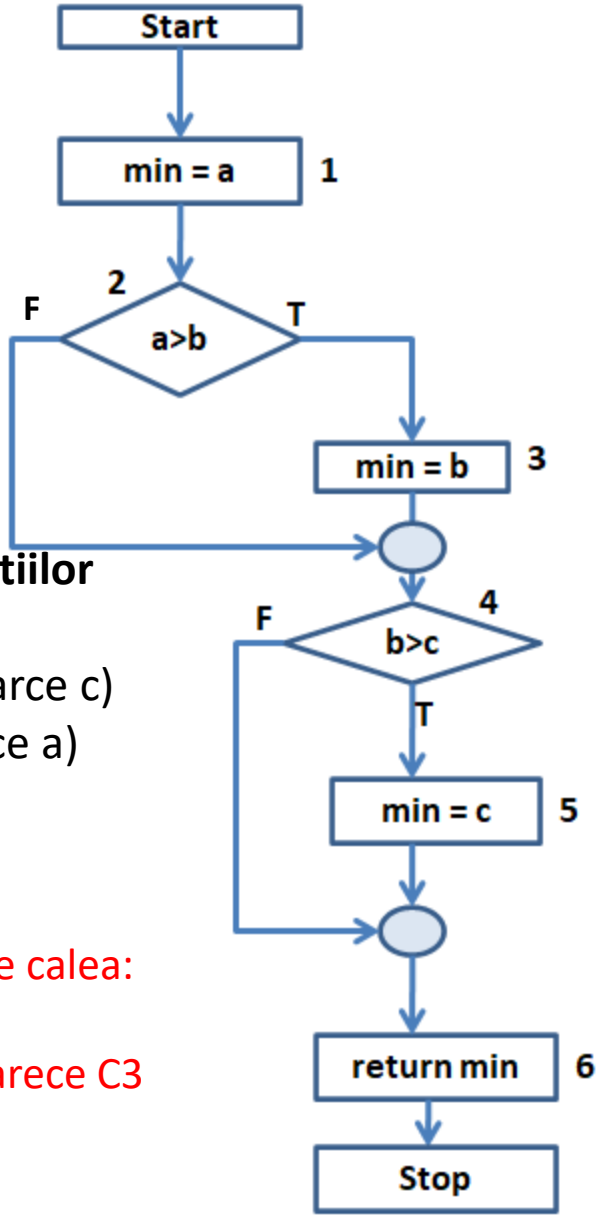
CONCLUZIE: în alegerea setului de cai de testat pe baza criteriului acoperirii tuturor ramificatiilor, se va tine cont de aceste recomandari.

Acoperirea tuturor ramificatiilor nodurilor de decizie ale grafului controlului (3)

Fie o functie care trebuie sa intoarca cea mai mica valoare dintre trei valori primite ca parametri.

Consideram urmatoarea implementare a functiei:

```
int minim (int a, int b, int c)
{ int min = a;
  if(a>b) min = b;
  if(b>c) min = c;
  return min;
}
```



Caile de test alese pe baza criteriului acoperirii tuturor ramificatiilor pot fi:

C1: (1, 2(T), 3, 4(T), 5, 6) → se testeaza cazurile: a>b si b>c (intoarce c)

C2: (1, 2(F), 4(F), 6) → se testeaza cazurile: a<=b si b<=c (intoarce a)

Cele 2 executii de test pot fi:

a = 50, b=40, c = 30 ; a=10, b=30, c = 40

Fie vectorul de intrare: a=15, b=30, c=20. Programul se va executa pe calea: C3: 1, 2(F), 4(T), 5, 6

Valoarea intoarsa este 20: greseala din cod nu va fi descoperita deoarece C3 nu a fost selectata pentru executiile de test.

Conditia din nodul 4 este gresita; trebuie sa fie: min>c

Acoperirea tuturor ramificatiilor nodurilor de decizie ale grafului controlului (4)

Exemple de cazuri de test pentru functia **minim**

Nr.	a	b	c	Relatia	Calea	Valoarea intoarsa	Valoare corecta
1.	1	1	1	a=b=c	1,2F,4F,6	1	1
2.	1	5	5	a<b si b=c	1,2F,4F,6	1	1
3.	5	5	3	a= b si b>c	1,2F,4T,5,6	3	3
4.	10	5	5	a>b si b= c	1,2T,3,4F,6	5	5
5.	10	5	20	a>b si b<c	1,2T,3,4F,6	5	5
6.	20	10	1	a>b si b>c	1,2T,3,4T,5,6	1	1
7.	**1**	**20**	**10**	**a<b si b>c si a<c**	**1,2F,4T,5,6**	**10**	**1**
8.	17	20	15	a<b si b>c si a>c	1,2F,4T,5,6	15	15

Cazurile de test 1, 2 si 6 corespund cailor C1 si C2 alese **pe baza criteriului acoperirii tuturor ramificatiilor.** Cf criteriului sunt suficiente 2 cazuri de test, care asigura executia pe caile C1, respectiv C2. Pentru orice caz de test care acopera una dintre cele 2 cai programul furnizeaza raspuns corect!

In cazurile (a<=b si b>c si a<c) programul furnizeaza raspuns gresit; exemplu: cazul de test 7.

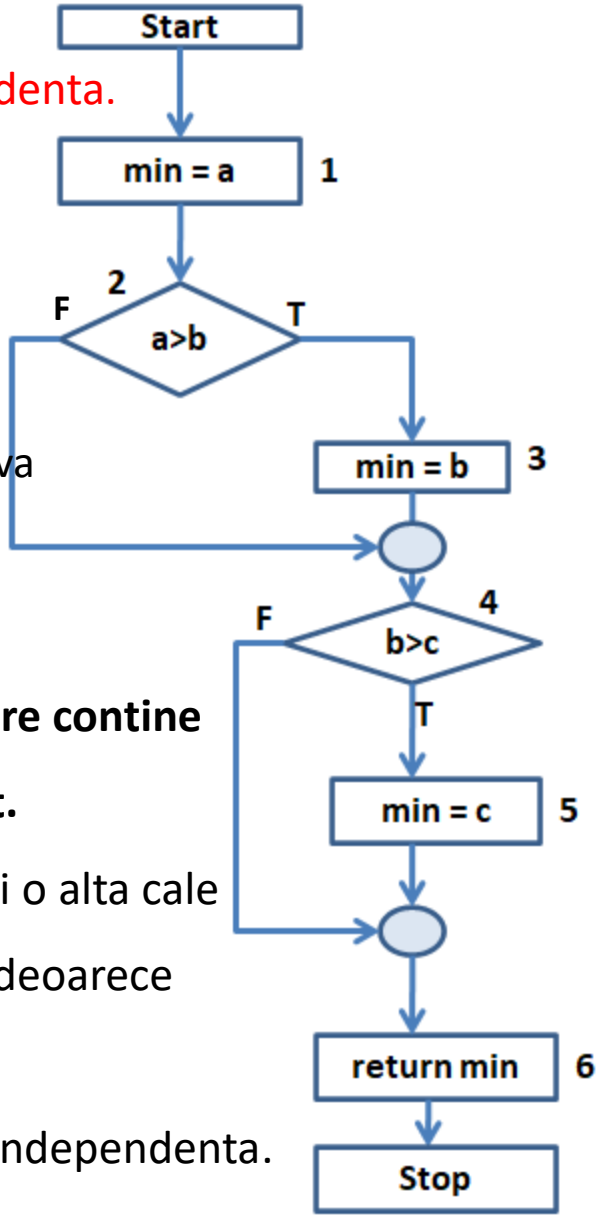
Acoperirea setului de căi de bază ale grafului controlului (1)

Criteriul acoperirii setului de cai de baza: fiecare ramificatie a unui nod de decizie trebuie testata pe o cale liniar independenta.

Pentru graful din figură se aleg căile:

1) TT - C1: (1, 2(T), 3, 4(T), 5, 6)
2) FT – C2: (1, 2(F), 4 (T), 5, 6) → acopera cazul a<b si b>c
3) FF – C3: (1, 2(F), 4(F), 6)

La executia caii C2 folosind cazul de test 7 din tabelul anterior se va descoperi ca programul produce un rezultat incorect!



➢ **Setul cailor de baza este un set de căi liniar independente**

- **Se construieste iterativ, adaugand de fiecare data o cale care contine cel putin un arc care nu este inclus in nici o alta cale din set.**
- Daca o cale introduce un nod nou (care nu este inclus in nici o alta cale liniar independenta) atunci calea este liniar independenta, deoarece introduce automat si un arc nou.
- O cale care este subcale a unei alte cai nu este o cale liniar independenta.

Acoperirea setului de căi de bază ale grafului controlului (2)

Complexitatea ciclomatica a unui program (McCabe, 1982):

v = e - n + 2

e este numarul de arce(edges) ale grafului controlului

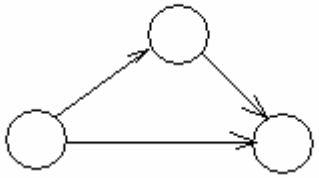
n este numarul de noduri(nodes)

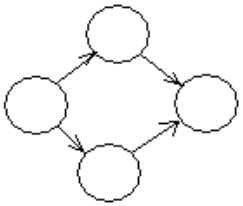
Complexitatea ciclomatică indică numarul de căi liniar independente prin codul sursă =

numarul de cazuri de test pe baza criteriului acoperirii setului de căi de baza.

Secventa e=0, n=1, v=1

➢Un singur caz de test este suficient pentru testarea unei secvente

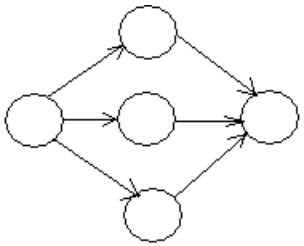
if ... then..  e=3, n=3, v=2

if..then..else..endif  e=4, n=4, v=2

Fiecare nod de decizie creşte complexitatea ciclomatica cu 1: 1(secventa) +1

Acoperirea setului de căi de bază ale grafului controlului (3)

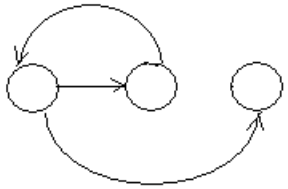
case



e=6, n=5, v=3

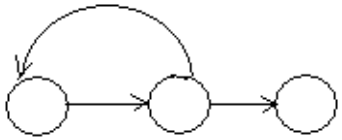
3 căi liniar independente: 3 cazuri de test

while..do



e=3, n=3, v=2

repeat ..until



e=3, n=3, v=2

Un nod de decizie introduce 2 căi liniar independente (2 cazuri de test)

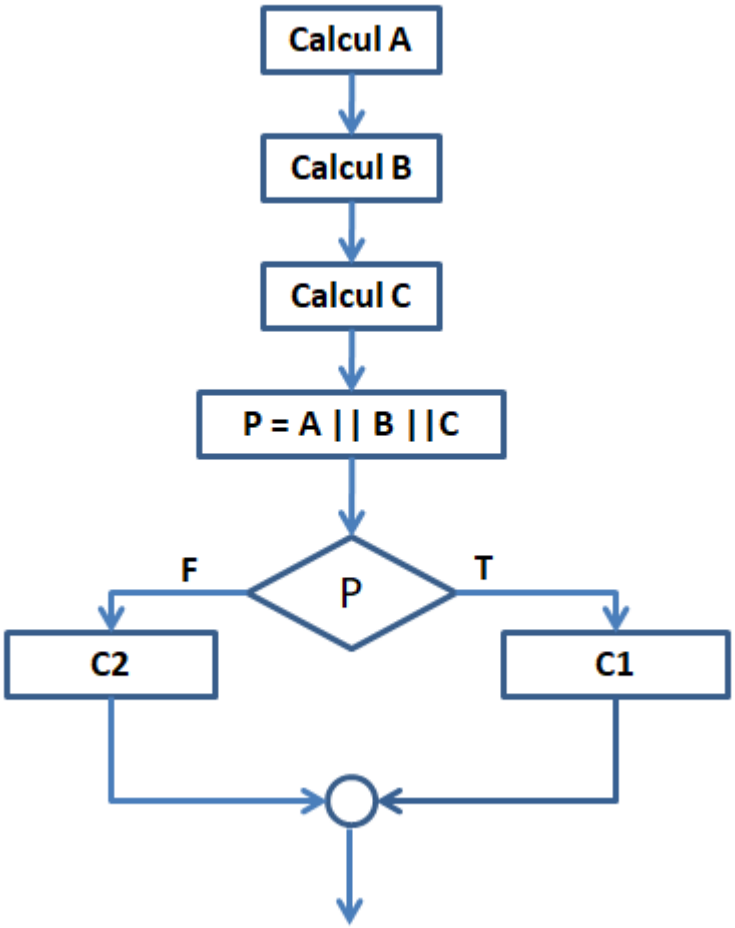
Pentru functia ***minim*** din exemplul anterior:

e = 7, n= 6, v = 7-6+2 = 3

- setul cailor de baza este compus din 3 cai liniar independente
- minim 3 executii de test

Acoperirea tuturor predicatelor

Criteriile bazate pe acoperirea tuturor ramificațiilor sunt insuficiente atunci cand nodurile de decizie conțin predicate compuse: de ex. (A && B && C), (A|| B || C). De aceea, în astfel de cazuri, setul de căi determinat pe baza acoperirii tuturor ramificațiilor/setului de căi de bază trebuie să fie completat cu teste pentru acoperirea predicatelor.



Caz	A	B	C	P
1	T	F	F	T
2	F	F	F	F

2 cazuri de test selectate pe baza criteriilor de acoperire a tuturor ramificatiilor. Nu sunt testate cazurile in care B=T sau C=T.

Caz	A	B	C	P
1	T	T	T	T
2	T	F	T	T
3	T	T	F	T
4	T	F	F	T
5	F	T	T	T
6	F	F	T	T
7	F	T	F	T
8	F	F	F	F

8 cazuri de test trebuie sa fie selectate pe baza criteriului acoperirii predicatelor: pot fi descoperite erori in calculul predicatelor B si C.

Alegerea cazurilor de test (1)

- **După stabilirea setului căilor de test, C = {c}, trebuie alese cazurile de test: datele de intrare folosite în teste (vectorii de test) si rezultatele asteptate la fiecare test.**

1. Se determină predicatul fiecărei căi, $R_c(I)$, $c \in C$, unde

$I = (i_1, i_2, \ldots, i_n)$ este vectorul variabilelor de intrare;

2. Pentru fiecare $R_c(I)$, $c \in C$, se determină un set de atribuiri pentru variabilele de intrare, X_c care satisface predicatul căii:

$R_c(X_c) = \text{true}$ **X_c** → vectorul de intrare care asigură parcurgerea căii

Atunci <u>setul vectorilor de intrare folosit în teste</u> este:

$VT = \{ X_c \mid c \in C, X_c \in D_I, R_c(X_c) = \text{true} \}$,

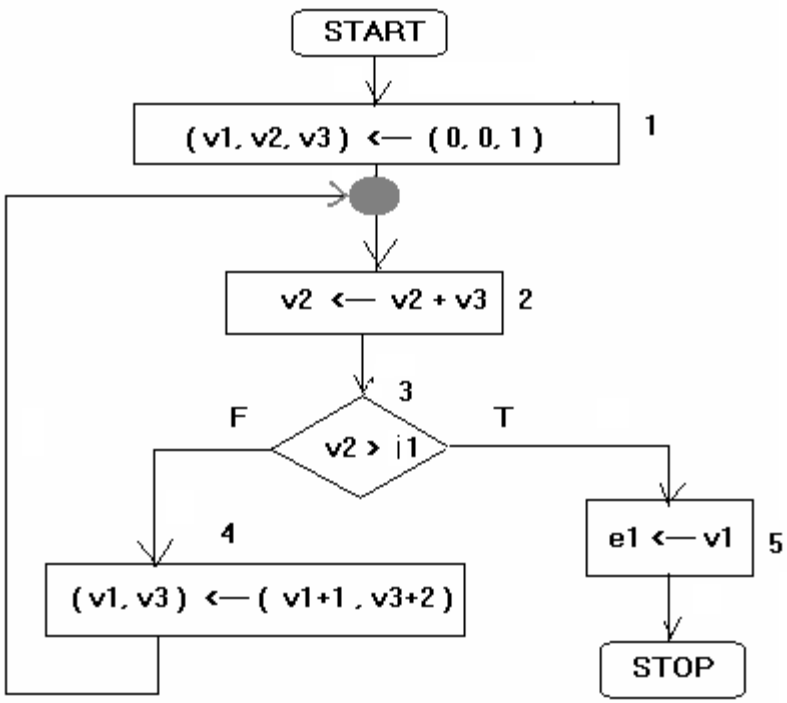
unde $D_I = X\ D_{ik}$ (produs cartezian), iar D_{ik} este domeniul de valori al variabilei de intrare i_k.

3. Pentru fiecare $X_c \in VT$, se determina valorile corecte ale variabilelor de iesire.

Alegerea cazurilor de test (2)

Fie urmatorul graf al controlului, pentru care s-au ales caile de test:

c1 = (1, 2, 3(T), 5)
c2 = (1, 2, 3(F), 4, 2, 3(T) , 5)



- Astfel, cele 2 ramificatii ale nodului de decizie sunt testate pe 2 căi liniar independente.

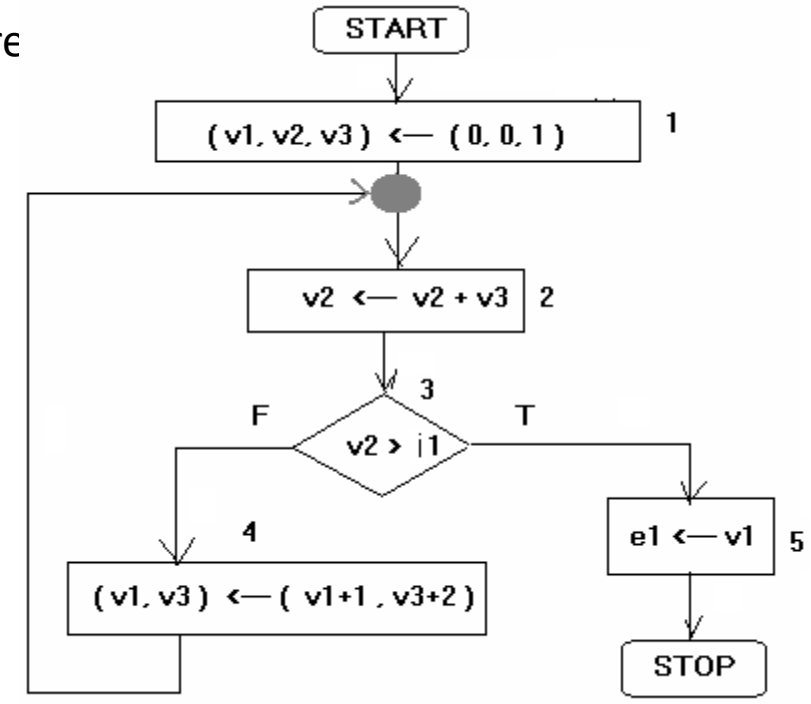
- **Pentru determinarea predicatului unei căi**, se parcurge calea de la nodul de iesire pana la nodul de intrare , concatenând conditiile de ramificare pe masura ce sunt intalnite.
- La intalnirea unei atribuiri, y←E, daca y face parte din expresia curenta a predicatului, se substituie y cu E.

Determinarea predicatului unei căi (1)

Exemplu:

c2 = (1, 2, 3(F), 4, 2, 3(T) , 5)

i1 este variabila de intrare, e1 este variabila de iesire



3T: Rc2 = v2> i1

2: Rc2 = (v2+v3)>i1

4: Rc2 = (v2 + v3 +2) > i1

3F: Rc2 = (v2 + v3 +2) > i1 Λ NOT(v2 > i1)

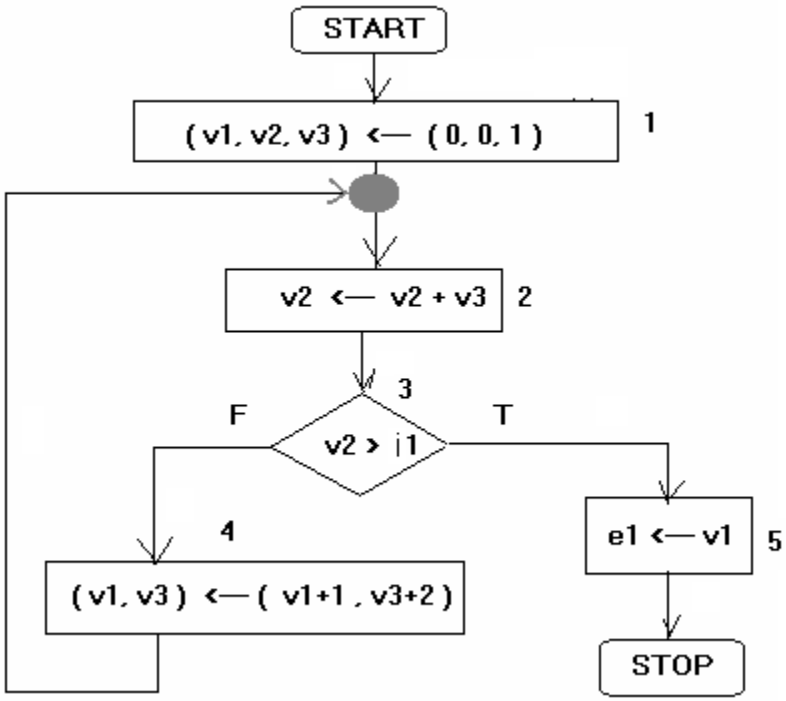
2: Rc2 = (v2 + v3 + v3 + 2) >i1 Λ NOT(v2+v3 >i1)

1: Rc2 = 4> i1 Λ NOT(1 >i1)

Rc2 = 1 <= i1 < 4

- Calea c2 va fi executata pentru orice valoare a lui i1 care satisface acest predicat: 1, 2, 3.

Determinarea predicatului unei căi (3)



c3 = (1, 2, 3(F), 4, 2, 3(F), 4, 2, 3(T), 5)

3T: Rc3 = v2> i1

2: Rc3 = (v2+v3)>i1

4: Rc3 = (v2 + v3 +2) > i1

3F: Rc3 = (v2 + v3 +2) > i1 Λ NOT(v2 > i1)

2: Rc3 = (v2 + v3 + v3 + 2) >i1 Λ NOT(v2+v3 >i1)

4: Rc3 = (v2 + v3 + 2+ v3+2 + 2) >i1

Λ NOT(v2+v3+ 2>i1) → Rc3 = (v2+2v3+4) > i1 Λ NOT(v2+v3+2 >i1)

3F: Rc3 = (v2+2v3+4) > i1 Λ NOT(v2+v3+2 >i1) Λ NOT(v2>i1)

2: Rc3 = (v2+v3 + 2v3 + 4) >i1 Λ NOT(v2+v3 + v3+2 >i1) Λ NOT(v2+v3>i1)

1: Rc3 = 7>i1 Λ NOT(4 >i1) Λ NOT(1>i1) → **Rc3 = 4 <= i1 < 7**

Stabilirea rezultatului unei executii de test

Stabilirea rezultatului corect al unei executii de test presupune cunoasterea specificatiei programului.

- **Programul din exemplu calculeaza numarul maxim de termeni ai sumei 1+3+5+... ..+ (2j+1), ∀ j >= 0, care pot fi adunati a.î. suma lor să fie <= i1.**

Rc2 = 1 <= i1 < 4

- Executand programul pentru orice 1 <= i1 < 4, rezultatul corect este e1 = 1.

Rc3 = 4 <= i1 < 7

- Executand programul pentru orice 4 <= i1 < 7, rezultatul corect este e1 = 2.

➢ **Criteriile de acoperire a grafului controlului pot conduce la alegerea unor căi netraversabile.**

Cale netraversabila = **cale neexecutabilă**: cale care nu va fi executata niciodata (nici o combinatie de valori ale variabilelor de intrare nu satisface predicatul căii)!

Avantajele/dezavantajele testării structurale

Avantaje

Pot fi generate automat:

- Graful controlului, pornind de la codul sursa
- Predicatele de cale, pe baza grafului controlului
- Vectorii de test, alesi aleator din domeniul datelor de intrare a.î. sa fie satisfacute predicatele de cale.

Este asistată de instrumente.

Dezavantaje

- Cazurile de test depind de codul programului, trebuie recalculate la orice modificare → nu pot fi reutilizate dupa o modificare a codului
- Cazurile de test acopera functionalitatile implementate, nu le descopera pe cele absente!

CONCLUZIE:

- Testele structurale trebuie sa fie completate cu cele functionale.
- Testele structurale pot evidentia unele erori care nu sunt descoperite prin testele functionale, care folosesc esantioane ale datelor de intrare → se începe cu testarea functionala si se continua cu testarea structurala.

Exercitii

1) Fie o functie implementata in C, care calculeaza $z= x^y$, pentru x, y, intregi pozitivi.

- Functia este declarata astfel:

  int putere(int x, int y)
- Sa se determine un set de cazuri de test pentru functie, prin:
  - Metoda testarii functionale
  - Metoda testarii structurale

Pentru testarea structurala, se va considera urmatorul cod:

```
int putere( int x, int y)
{  int z;
   z=1;
   while(y!= 0)
   { if(y mod %2 ==1) z = z *x;
     y = y/2; x = x*x;
   }
   return z
   }
```

Exercitii

2) Fie o functie implementata in C, care calculeaza numarul de termeni pozitivi dintr-un tablou de n numere intregi, X, si suma lor.

- Functia este declarata astfel:

  void f (int *X, int n, int &s, int &k)

  k – numarul de termeni pozitivi

  s – suma termenilor pozitivi

- Sa se determine un set de cazuri de test pentru functie, prin:
  - Metoda testarii functionale
  - Metoda testarii structurale

Lecturi suplimentare

1. https://sttp.site/chapters/testing-techniques/structural-testing.html
2. http://www.cs.toronto.edu/~chechik/courses18/csc410/Ch12-13-StructuralAndDataflowTesting.pdf
3. https://www.testbytes.net/blog/structural-testing-in-software-testing/

Test driven development (TDD)

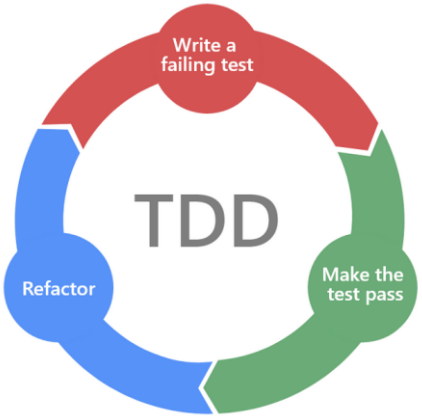
Prof. univ. dr. ing. Florica Moldoveanu

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

Ce este TDD?

❖ Este o practică în dezvoltarea de software, cu originile în metodologia agilă XP (*Extreme programming),* folosită în prezent nu numai în contextul unei dezvoltări agile.

Test driven development = Test first development + Refactoring



[2]

Test-first development:

- *Programatorul scrie testele unitare înainte de a scrie codul testat*
- Testele sunt dezvoltate incremental pornind de la scenarii.
- Fiecare test corespunde unei cerințe.
- Reprezentantul utilizatorilor/clientului este implicat în dezvoltarea şi validarea testelor.
- Sunt folosite instrumente de testare unitară automată.
- Atunci când este adăugat/modificat cod sunt rulate toate testele, atât cele anterioare

cât şi noul test – se verifică astfel ca noul cod nu a introdus erori.

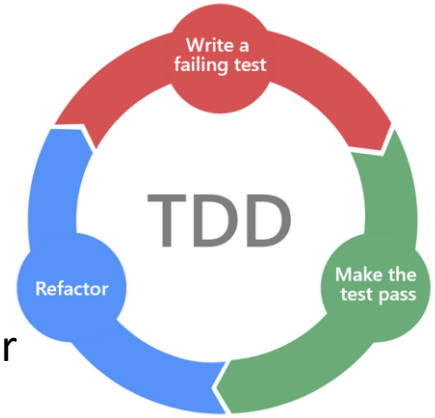
Ce este TDD?

Refactorizare (îmbunătățirea codului)

▪ *Rescrierea codului, fără a modifica funcționalitatea sa externă, pentru îmbunătățirea unor atribute ale codului, cum ar fi: eficiența, claritatea, complexitatea, uşurința de întreținere.*

➢ Codul este mai uşor de înțeles, reduce necesitatea documentației.

➢ Modificările se fac mai uşor deoarece codul este bine structurat şi clar.



[2]

Rosu.

- Se scrie un test unitar, înainte de a scrie codul testat. Rezultatul execuției testului este o cădere (failure), caci codul testat nu exista - eroarea de compilare este o cădere.

Verde.

- Se scrie într-un timp scurt codul care trebuie să satisfacă testul, chiar dacă nu este o soluție de implementare bună. Rezultatul execuției testului trebuie sa fie “success” (passed).

Albastru

- Refactorizarea se aplică atât codului testat cât şi codului de test, şi nu trebuie să modifice comportarea externă a programului.
  Se reexecuta toate testele. Se fac ajustări până când toate testele trec.

TDD – paşii (1)

Presupunem că noua funcționalitate se va implementa printr-o **functie f, dintr-o clasă C**, şi folosim Junit pentru testarea unitară.

Pentru testarea funcției f din **clasa C** se implementeaza o functie **test_f**, în **clasa de test CT**, parcurgându-se următorii paşi:

repetă

scrie un test în functia test_f, în care se apelează funcția **f**, folosind o aserțiune

repetă

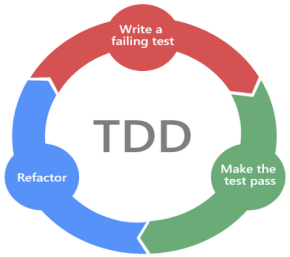
- rulează programul de test (cu toate testele definite în functia test_f)
- dacă rezultatul rulării este "failed" **adauga (după prima rulare)/modifică funcția f**

până când rezultatul rulării este "passed"

repetă

- refactorizează codul funcției **f** şi al funcției **test_f**
- rulează programul de test (cu toate testele definite în functia **test_f**)

până când toate testele definite în **test_f** trec (nici un test nu a produs cadere)



până când nu mai sunt teste noi pentru funcția f

TDD – caracteristici

- Se acordă aceeaşi importanță codului de test ca şi codului testat.
- **Testele trebuie să fie clare: uşor de înțeles**
- **Un (caz de) test trebuie să corespundă unei cerinte şi să conțină o singură aserțiune**.
- Testele trebuie sa fie independente intre ele.
- **Testele trebuie sa fie repetabile: reproductibile în orice context.**
- Fiecare test trebuie să aibă un rezultat binar: cădere sau success.
- Un test trebuie să fie scris imediat înainte de scrierea codului pe care-l validează.
- **Testele reprezintă specificații executabile şi fac parte din documentația aplicației.**
- Aplicarea TDD necesită o perioadă de învățare - este o shimbare de paradigmă – dar este o investiție în viitor → conduce la creşterea productivității activității de dezvoltare.
- Aplicarea TDD necesită instrumente bune de testare unitară automată.

TDD – beneficii (1)

- **Conduce la dezvoltarea de software cu grad înalt de uzabilitate (usability):** uşurința de înțelegere, de învățare, de operare, plăcerea de a utiliza produsul, de către utilizatorii țintă

  - abordarea TDD îi forțează pe dezvoltatori să se focalizeze pe problemele de uzabilitate pe tot parcursul dezvoltarii;

  - uzabilitatea este asigurată prin testele unitare: reprezentantul utilizatorilor este implicat în dezvoltarea şi validarea testelor.

- **Reduce numărul de defecte din cod:** sunt eliminate in mare parte prin tesele unitare

- **Evidențiază problemele de funcționalitate în fazele incipiente** → reduce costul rezolvarii problemelor de implementare a funcționalitatilor cerute.

- **Simplifică codul**: solicită dezvoltatorului să scrie cod care trebuie să raspundă cerințelor de testare; se evită scrierea de cod care nu este necesar.

TDD – beneficii (2)

- **Conduce la cod extensibil**: dezvoltatorul este încurajat să construiască software alcătuit din unități mici de cod testabil.
- **Permite obtinerea de feedback rapid de la client/utilizatori.**
- **Documentarea actualizată a codului: testele sunt exemple de utilizare a codului.**
- Scurtează timpul de dezvoltare: abordarea “test-and-code” creşte productivitatea echipei de dezvoltare. Permite scrierea de cod funcțional într-un timp mai scurt decât alte practici de dezvoltare.

Resurse

1. https://hackernoon.com/introduction-to-test-driven-development-tdd-61a13bc92d92
2. https://marsner.com/blog/why-test-driven-development-tdd
3. Kent Beck, Test Driven Development: By Example, Addison Wesley, 2003
4. https://www.infoq.com/articles/test-driven-design-java/
5. Un demo cu aplicarea TDD în testarea unitară: *https://www.youtube.com/watch?v=ZHZY4lVm6co&feature=emb_title*

Dezvoltarea agilă

Prof. univ. dr. ing. Florica Moldoveanu

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

Agile Manifesto – cele 4 valori fundamentale

- Metodologiile agile au aparut din necesitatea de a elimina din dezvoltarea de software activitatile care conduc la intarzierea livrarii de software operational.
- Termenul “dezvoltare agila” - derivat din **Agile Manifesto**, publicat in 2001 de un grup din care au facut parte creatorii unor metodologii existente la acea vreme: Scrum, Extreme Programming (XP), Dynamic Systems Development Method (DSDM) si Crystal.
- Agile Manifesto a stabilit un set de valori și principii generale comune metodologiilor agile existente la momentul respectiv. Sunt definite **4 valori fundamentale** care permit echipelor de dezvoltare software sa fie performante:
  - **Indivizi si interactiuni** în loc de procese si instrumente
  - **Software functional** în loc de documentatie cuprinzatoare
  - **Colaborarea cu clientul** în loc de negociere contractuala
  - **Raspunsul la schimbari** în loc de urmărirea unui plan

Agile Manifesto – cele 12 principii (1)

1. Prioritatea maxima o are satisfacerea clientului prin furnizarea rapida si continua de software functional.
2. Schimbarile in cerinte sunt binevenite, chiar si tarziu in dezvoltare, in avantajul clientului.
3. Livrarea frecventa de software functional, la cateva saptamani sau luni, preferabil la intervale mai scurte.
4. Dezvoltatorii si expertii din domeniul aplicatiei trebuie sa lucreze impreuna zilnic pe toata durata proiectului.
5. Construieste proiecte în jurul unor persoane motivate. Dă-le mediul și sprijinul de care au nevoie, și ai încredere în ei pentru a obtine rezultatul.

   Echipele agile sunt alcatuite din oameni care doresc sa lucreze impreuna colaborativ si sa invete unul de la altul. Oamenii sunt principalul factor de succes in dezvoltarea de software.

6. Cea mai eficace metoda de transmitere a informatiilor este comunicarea directa in echipa.
7. Software-ul functional este principala masura a progresului.

Agile Manifesto – cele 12 principii (2)

8. Procesele agile promoveaza dezvoltarea durabila. Sponsorii, dezvoltatorii si utilizatorii trebuie sa fie capabili sa mentina un ritm constant pe timp nelimitat.

   Pentru a obtine rezultate de calitate, oamenii nu trebuie fortati sa munceasca 12 ore/zi in permanenta.

9. Atenția continua la excelenta tehnica si o buna proiectare maresc agilitatea.

   Practici: refactoring - mentinerea unui cod de calitate si test-driven development - software-ul este functional in permanenta. Se folosesc ghiduri de codare si uneori ghiduri de modelare.

10. Simplitatea – arta de a maximiza cantitatea de munca nefacuta – este esentiala.

    - Focalizare pe ceea ce trebuie livrat, evitandu-se munca inutila, timp consumat inutil, pentru a maximiza ROI

11. Cele mai bune arhitecturi, cerinte si proiectari rezulta din echipe care se auto-organizeaza.

    Agile Model Driven Development (AMDD) și Test Driven Development (TDD) sunt abordările principale din cadrul comunității agile, pentru producerea de arhitecturi eficiente, cerințe, și modele de calitate.

12. La intervale regulate echipa analizeaza cum poate sa devina mai eficienta apoi isi ajusteaza corespunzator comportamentul.

Metodologii de dezvoltare agilă

Metodologii: Extreme programming (XP), Scrum, Crystal, Dynamic Systems Development Method (DSDM), Feature Driven Development (FDD), Behavior Driven Development (BDD), Kanban, s.a.

PRINCIPII GENERALE

Implicarea clientului	Clientul trebuie să fie puternic implicat în procesul de dezvoltare. Rolul lui este de a furniza şi prioritiza noi cerințe şi de a evalua iterațiile dezvoltării.
Livrarea incrementală	Produsul software este dezvoltat în incremente, clientul specificand cerințele care trebuie sa fie implementate în fiecare increment.
Oameni, nu procese	Abilitațile echipei de dezvoltare trebuie sa fie recunoscute şi exploatate. Membrii echipei trebuie lasați să-şi dezvolte propriul stil de lucru, fără să li se impuna procese prescriptive.
Inglobează schimbarea	Se asteaptă schimbarea cerințelor, de aceea sistemul trebuie proiectat pentru a fi uşor adaptat la schimbari.
Menține simplitatea	Focalizare pe simplitate atât în produsul software dezvoltat cât şi în procesul de dezvoltare. Elimină complexitatea oricând este posibil.

Extreme programming (XP)

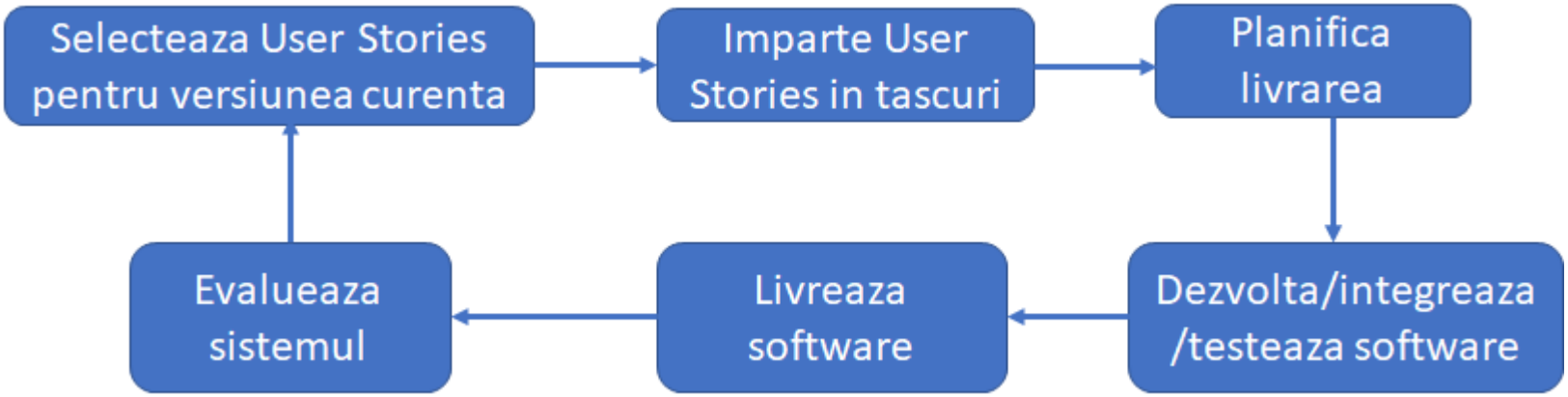
- Aparuta la sfarsitul anilor ’90
- **Abordare “extremă” în dezvoltarea iterativă:**
  - Versiuni noi pot fi construite de mai multe ori pe zi
  - Incrementele sunt livrate clientilor la intervale de 2 saptamani
  - Pentru fiecare versiune executabila trebuie rulate toate testele si versiunea este acceptata numai daca testele au trecut cu succes.

Practici cheie

- “User Stories” pentru specificare
- Test driven development
- Pair programming

XP – User stories

- In XP, **un reprezentant al utilizatorilor finali/clientului face parte din echipa de dezvoltare** si este **responsabil cu cerintele**.
- Cerintele utilizator sunt exprimate sub forma de **scenarii, numite "user stories".**
- User story trebuie sa fie o descriere scurta, fiind limitata la o pagina de hartie de dimensiuni mici (de obicei 3x5 inches – 1 inch =2.54 cm), impiedicand astfel cresterea descrierii.
- Reprezinta o modalitate simpla si rapida de a capta cerintele clientului.
- Pornind de la scenarii dezvoltatorii definesc tascuri de implementat.
- Pe baza tascurilor se fac estimarile de cost si timp.
- **Clientul alege cerintele de implementat in urmatorul livrabil pe baza prioritatii lor si a estimarilor de timp**.



User story

Kate is a doctor who wishes to prescribe medication for a patient attending a clinic. The patient record is already displayed on her computer so she clicks on the medication field and can select ‘current medication’, ‘new medication’ or ‘formulary’.

If she selects ‘current medication’, the system asks her to check the dose. If she wants to change the dose, she enters the dose and then confirms the prescription.

If she chooses ‘new medication’, the system assumes that she knows which medication to prescribe. She types the first few letters of the drug name. The system displays a list of possible drugs starting with these letters. She chooses the required medication and the system responds by asking her to check that the medication selected is correct. She enters the dose and then confirms the prescription.

If she chooses ‘formulary’, the system displays a search box for the approved formulary. She can then search for the drug required. She selects a drug and is asked to check that the medication is correct. She enters the dose and then confirms the prescription.

The system always checks that the dose is within the approved range. If it isn’t, Kate is asked to change the dose.

After Kate has confirmed the prescription, it will be displayed for checking. She either clicks ‘OK’ or ‘Change’. If she clicks ‘OK’, the prescription is recorded on the audit database. If she clicks on ‘Change’, she reenters the ‘Prescribing medication’ process.

http://iansommerville.com/software-engineering-book/

XP – Pair programming

- Dezvoltatorii lucreaza in perechi, creând codul impreuna.
- Este o revizie informala a codului – fiecare linie de cod este vazuta de 2 persoane
- Perechile sunt create dinamic, astfel încat toti membrii echipei lucreaza împreuna pe parcursul dezvoltarii
  - → ajuta la cunoasterea codului de toata echipa
  - → toata echipa este responsabilă pentru intregul cod (proprietate colectiva)
- Incurajeaza refactorizarea
- Dezvoltarea în echipă reduce riscul proiectului atunci când un membru paraseşte echipa.
- Dezvoltarea in perechi nu este ineficienta: exista dovezi care sugereaza ca este mai eficienta daca cei 2 ar lucra separat.

XP şi principiile agile

- **Dezvoltarea incrementala**: incremente mici livrate frecvent
- **Implicarea clientului**: angajarea permanenta a clientului in echipa.
- **Oameni nu procese**: pair programming, proprietate colectiva, evitarea programului de lucru prelungit.
- **Schimbarile** acceptate prin livrari regulate.
- **Mentinerea simplitatii** prin refactorizarea continua a codului.

XP - limitari

- Fara teste/proceduri de acceptare specificate de client “user stories” sunt supuse interpretarii, ceea ce le face dificil de utilizat ca baza contractuala.
- Solicita contactul permanent cu clientul pe tot parcursul proiectului, dificil in unele cazuri, sau timp suplimentar consumat.
- Dificil de scalat la proiecte mari.
- Presupune dezvoltatori competenti.
- XP este greu de integrat cu practicile de management din cele mai multe companii.
- Desi in dezvoltarea agila se folosesc practici din XP, metoda asa cum a fost definita initial nu este larg folosita.

Scrum

Conform definitiei creatorilor săi, Scrum este un framework care permite abordarea de probleme complexe, oferind în același timp productiv și creativ produse de cea mai înaltă valoare posibilă [1].

Este un framework de dezvoltare agila cu urmatoarele caracteristici esentiale:

❑ Echipa Scrum

- Se auto-organizeaza: nu exista un conducator al echipei care decide ce va face fiecare membru al echipei si cum se rezolva o anumita problema; acestea sunt hotarate de echipă ca un intreg.
- Este multifunctionala: este necesar ca fiecare membru al echipei să fie capabil sa trateze o problema/sarcina de la idee la implementare.
- Este sprijinită de 2 roluri specifice:

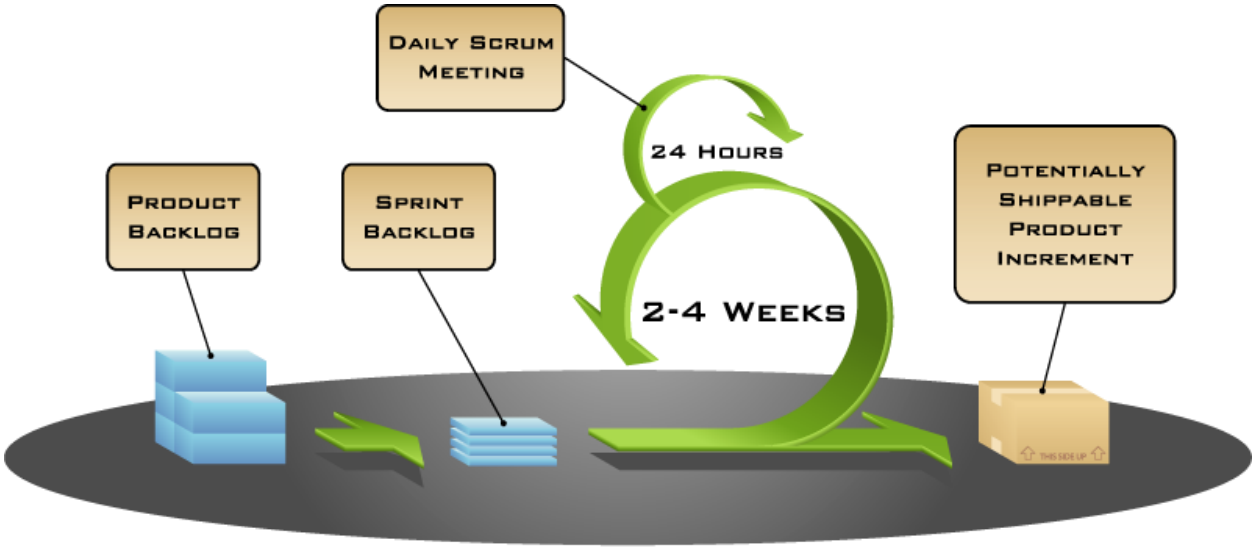
- **ScrumMaster**, care poate fi considerat un antrenor al echipei, ajutand-o sa foloseasca procesul Scrum pentru a performa la nivelul cel mai inalt.
- **Product Owner**, care reprezinta clientul/utilizatorii finali/partile interesate in proiect şi ghideaza echipa pentru construirea produsului aşteptat de acestia.

Caracteristici Scrum (2)

❑ Procesul de dezvoltare

- **Product owner** creaza lista de cerinte prioritizata, numita "**product backlog**"
- Dezvoltarea este organizata in iteratii scurte, de 2-4 saptamani, numite "sprint-uri".
- In cadrul unui sprint este proiectat, codat și testat un increment al produsului final, potential livrabil.
- In fiecare sprint sunt implementate o parte dintre cerintele din "product backlog", selectate de echipă, care formeaza "**sprint backlog**".
- In timpul unui sprint echipa se intalneste zilnic pentru a evalua progresul său (**Daily Scrum**)
- Nu se accepta schimbari in timpul unui sprint, de aceea durata sprint-urilor este planificata de echipă astfel încat sa nu fie necesare modificari in timpul sprint-urilor.
- Sprintul se termina cu un "**sprint review**" si o "**retrospectiva**" a sprint-ului

Procesul Scrum (2)



Sursa: [2]

Copyright © 2005, Mountain Goat Software

Scrum framework

Roluri

- Product owner
- ScrumMaster
- Echipa de dezvoltare

Activităţi

- Planificare sprint
- Daily Scrum
- Revizia sprint-ului
- Retrospectiva sprint-ului
- Rafinare product backlog

Artefacte

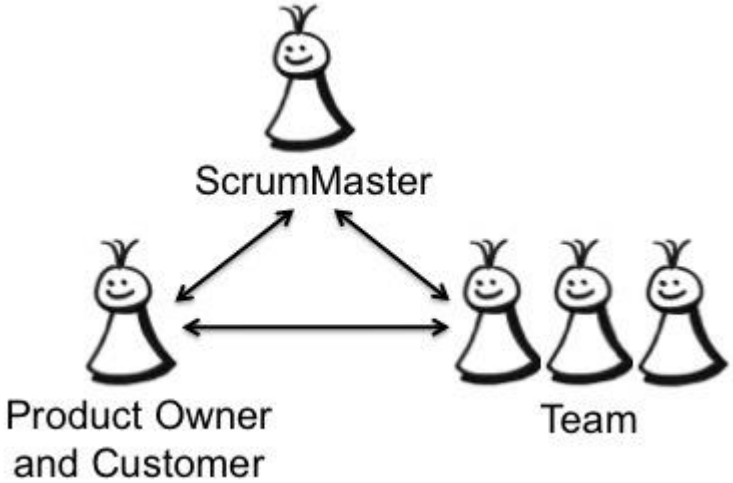
- Product backlog
- Sprint backlog
- Burndown charts
- Versiuni executabile ale produsului

Roluri - Product owner

- Este membrul echipei Scrum care **are sarcina de a maximiza valoarea muncii echipei**:
  - este responsabil pentru profitabilitatea produsului (ROI – Return On Investment)
- Detine viziunea asupra produsului, colaboreaza strans cu partile interesate in proiect (utilizatori, clienti), cultiva şi alimenteaza o comunitate in jurul produsului.
- **Faciliteaza comunicarea dintre echipă şi partile interesate, asigura construirea produsului potrivit (asteptat de clienti, utilizatori).**
- Decide continutul listei de cerinte (product backlog) şi ordoneaza articolele sale după prioritate astfel incat sa se obtina valoarea maxima.
- Decide când va fi lansat produsul și ce funcționalitați va avea.
- Modifică lista de cerințe înaintea fiecarei iterații, după cum e necesar.
- Acceptă sau respinge ceea ce s-a produs.
- Decide impreuna cu echipa care articole din lista de cerințe vor trebui considerate pentru sprintul curent.

Roluri - ScrumMaster

- **Antreneaza echipa**: ajuta intreaga echipa sa obtina performanțe mai bune, sa aplice corect metodologia Scrum, sa aplice practici tehnice care o ajuta sa finalizeze sarcinile fiecarui sprint.
- **Responsabil pentru aplicarea valorilor și practicilor din Scrum**.
- Incurajeaza auto-organizarea echipei.
- Îndepărtează impedimentele.
- Se asigură că echipa este complet funcțională și productivă.
- Încurajează cooperarea strânsă între toate rolurile.
- Protejează echipa de interferențele externe.



Roluri – Echipa

- De obicei 5-9 persoane.
- **Multi-funcțională**:
  - Nu include roluri traditionale ca programator, tester, arhitect
  - Toti din echipa lucreaza împreuna pentru realizarea sarcinilor sprint-ului, pe care si le-au asumat in mod colectiv
- Membrii trebuie sa fie disponibili echipei si proiectului tot timpul.
- **Se auto-organizează**:
  - Product owner creaza o lista ordonata cu ceea ce este necesar sa fie facut
  - Membrii echipei evalueaza cât de mult se poate face într-un singur sprint şi se auto-organizeaza, hotărând ei singuri cine ce face astfel încât la sfarsitul sprintului să se obtina un nou increment al produsului.
- Membrii echipei ar trebui schimbați doar între sprinturi.

Artefacte - Product backlog

- O lista ordonata de idei si cerințe pentru produs, care pot proveni de la product owner, membrii echipei sau partile interesate.
- Fiecare articol din lista este insotit de o estimare a efortului de implementare in numar de ore.
- Lista este ordonata (prioritizata) de către product owner, astfel incat valoarea muncii echipei sa fie maximizata.
- Lista este reordonata la începutul fiecărui sprint.
- De obicei, la inceputul proiectului lista este scurta şi vagă, devine apoi din ce in ce mai lunga şi mai bine definita.
- **Product owner este responsabil pentru crearea şi rafinarea listei dar echipa îl ajuta în producerea si actualizarea listei.**

Un exemplu de product backlog

Articol in product backlog	Estimare
Permite unui oaspete să facă o rezervare	3
În calitate de oaspete, vreau să anulez o rezervare.	5
În calitate de oaspete, vreau să schimb datele rezervării.	3
Ca angajat al hotelului, pot să execut rapoartele RevPAR (revenue-per-available-room)	8
Îmbunătățește gestionarea excepțiilor	8
...	30

Sursa [2]

Activitati - Planificarea sprint-ului

- Are loc in cadrul unei intalniri cu durata limitata la care participa intreaga echipa.
- **Product owner:**

  - prezinta ce anume trebuie facut, utilizand articole din product backlog

  - raspunde la întrebarile echipei pentru a lamuri neînțelegerile asupra articolelor din product backlog
- **Echipa:**

  - decide cate articole sa ia din product backlog şi cum sa le realizeze

  - identifica taskurile şi estimeaza durata pentru fiecare (1-16 ore)

  - creaza sprint backlog-ul
- Durata recomandata pentru o intalnire de planificare sprint este de 2 ore (sau mai putin) x numarul de saptamani ale sprint-ului.
- Succesul intalnirii depinde de calitatea product backlog-ului, de aceea este importanta rafinarea sa.
- Adesea se defineste un scop al sprintului, care ajuta echipa sa se focalizeze pe realizarea sa.
- Se discuta design-ul de nivel înalt.

Artefacte - sprint backlog

- Este lista articolelor din product backlog detaliate, alese pentru implementare in sprintul curent, impreuna cu taskurile planuite de echipa pentru realizarea lor si estimarea efortului.
- Odata stabilit sprint backlog, echipa incepe sa lucreze la noul increment al produsului.

Exemplu de sprint backlog

Task -uri	L	Ma	Mi	J	V
Impl. interf. cu utilizatorul	8	4	8		
Impl. middle tier-ul	16	12	10	4	
Testează middle tier-ul	8	16	16	11	8
Scrie help-ul online	12				
Impl. clasa foo	8	8	8	8	8
Adaugă logarea erorilor			8	4	

Sursa [2]

Meeting-ul SCRUM zilnic

- Are loc la aceeasi ora si in acelasi loc.
- Dureaza maxim 15 minute. Se stă în picioare.
- Nu se rezolvă probleme
  - Este invitata toată lumea
  - Doar membrii echipei, ScrumMaster-ul si Product owner-ul pot vorbi
- Fiecare membru al echipei raspunde la 3 intrebari:
  - Ce ai facut ieri?
  - Ce vei face azi?
  - Exista ceva care te împiedică?
- Acestea NU reprezinta un raport pentru ScrumMaster, sunt angajamente in faţa unor egali.
- In functie de ceea ce s-a discutat la intalnirea zilnica echipa se poate reorganiza, pentru indeplinirea scopului sprintului si finalizarea incrementului.

Activitati - Sprint review

- Intalnire la finalul sprintului, cu durata limitata:
  recomandabil, o ora x nr. de spatamani ale sprintului.
- Informal
  - De regulă, 2 ore pentru pregătire
  - **Fără slide-uri**
- **Participanti**: product owner, echipa, ScrumMaster, conducerea companiei, clienti, dezvoltatori din alte proiecte.
- **Echipa prezintă ce a realizat în timpul sprintului**
- De obicei functionalitatile implementate in timpul sprint-ului sunt demonstrate prin scurte **aplicatii demo.**
- Se discuta ce au observat membrii echipei in timpul sprintului si eventual idei noi
- Se actualizeaza product backlog cu cerintele ramase

Activitati - Retrospectiva sprint-ului

- Intalnire scurta (maxim 3 ore), dupa Sprint review.
- Se discuta despre “ce merge și ce nu merge”.
- **Se identifica potentiale imbunatatiri si se planifica imbunatatirea procesului**.
- Participa:
  - toti membrii echipei
  - eventual clienți și alte persoane
- Scrum master trebuie sa asigure ca retrospectiva are loc si participanții înțeleg scopul ei.
- Incurajeaza membrii echipei sa participle la retrospective.

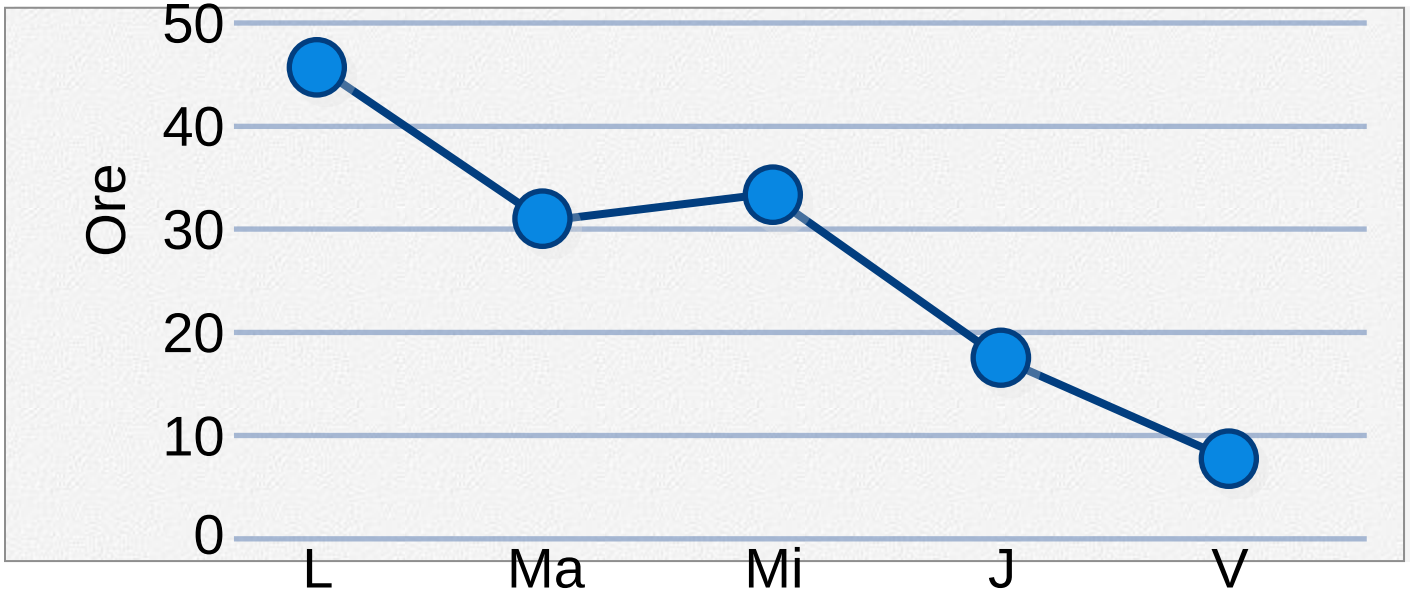
Gestionarea sprint backlog-ului

- Fiecare persoană își alege ce va lucra după propria dorință.
  - Taskurile nu sunt niciodată asignate de altcineva
- **Estimările din sprint backlog sunt actualizate zilnic.**
- Orice membru al echipei poate sa adauge, sa șteargă sau sa modifice articole din sprint backlog.
- Taskurile din cadrul sprint-ului sunt descoperite în mod natural.
- Durata lor este ajustata pe parcursul sprintului.
- **Se actualizează volumul de muncă rămas pe măsură ce se obțin mai multe informații.**

Artefacte - Sprint burndown

Task-uri	L	Ma	Mi	J	V
Impl. interf. cu utilizatorul	8	4	8		
Impl. middle tier-ul	16	12	10	7	
Testează middle tier-ul	8	16	16	11	8
Scrie help-ul online	12				

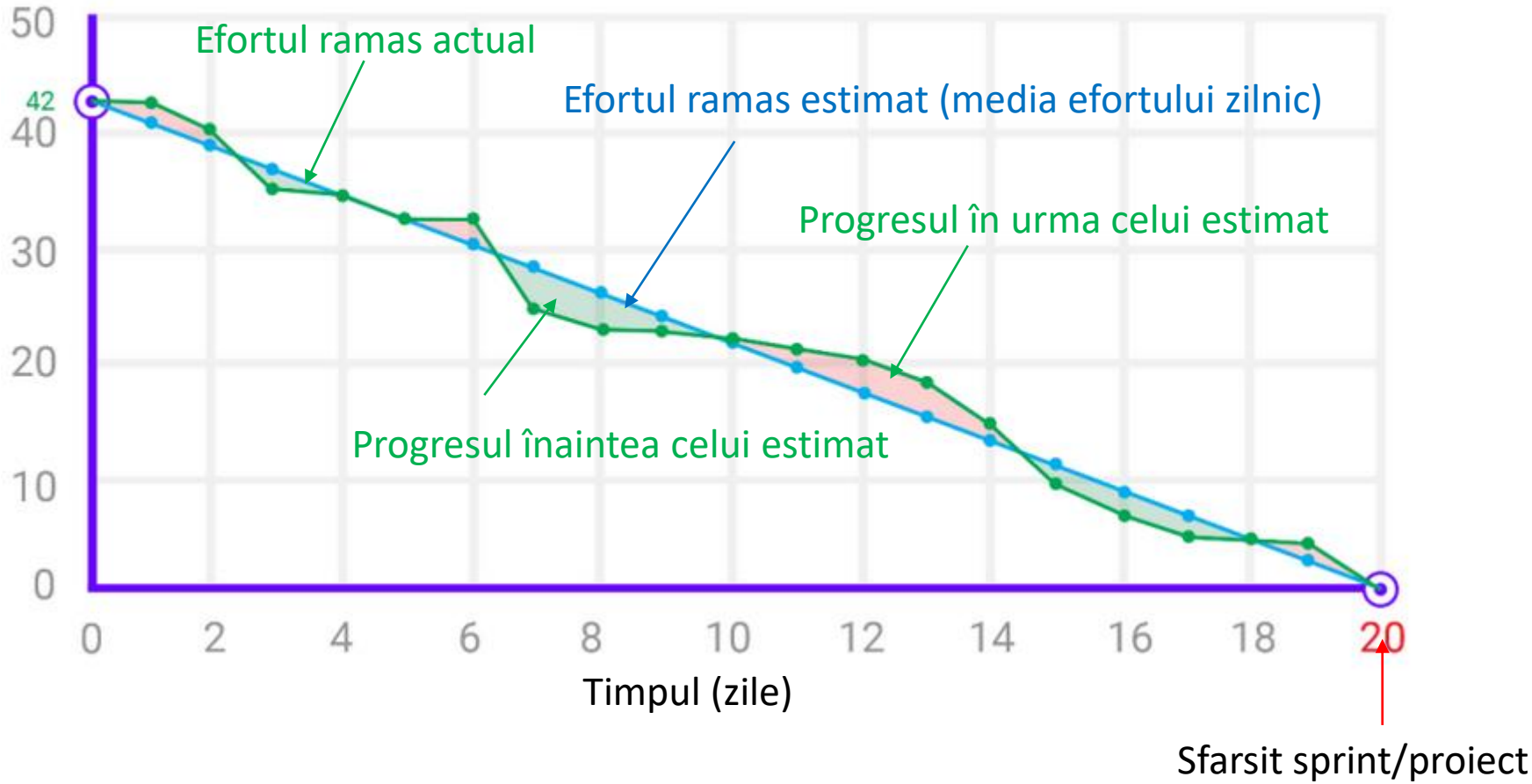
Diagrama sprint burndown: efortul ramas (ore) pana la sfarsitul sprintului



Sursa [2]

Diagrama sprint burndown

Efortul (ore/zile) ramas pana la terminarea sprintului/proiectului



Inaintea/în urma planificarii → ajustarea estimarilor

Rafinarea product backlog-ului

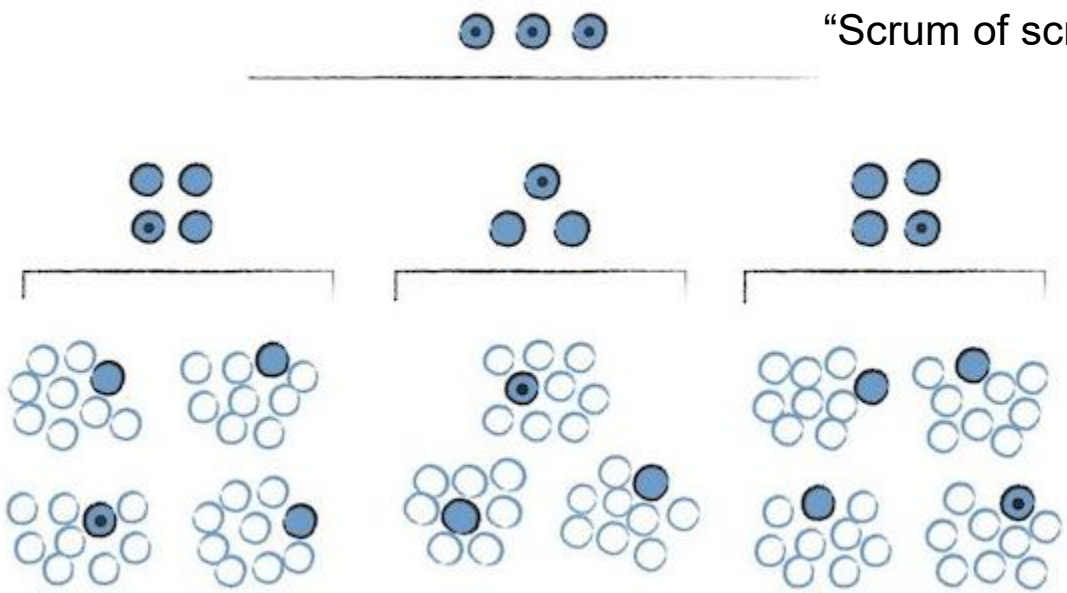
- **Activitate care se desfasoara pe toata durata unui proiect Scrum.**
- Este responsabilitatea intregii echipe.
- Consta din:

- Adaugarea de articole noi sau care au devenit mai importante
- Eliminarea articolelor care nu mai sunt importante
- Ordonarea articolelor
- Impartirea articolelor mari in articole mai mici, astfel incat sa poata fi implementate pe durata unui singur sprint
- Gruparea mai multor articole intr-unul singur
- Identificarea articolelor care ar trebui sa fie selectate in urmatorul sprint

Scalabilitate

- Echipa Scrum tipică e formată din 7 ± 2 persoane
  - Scalabilitatea se obține prin echipe formate la rândul lor din echipe: **scrums of scrums**
- Factori care pot influența scalarea
  - Tipul şi complexitatea aplicației
  - Numarul de persoane estimat pentru dezvoltare
  - Împrăștierea echipei
  - Durata proiectului
- Scrum-ul a fost folosit in mai multe proiecte la care au participat peste 500 persoane
- **Meeting-ul "Scrum of scrums"**
  - Are rolul de a coordona activitatea unui grup de echipe
  - Fiecare echipa din grup, la finalul intalnirii zilnice, desemneaza o persoana care va participa la meeting-ul "Scrum of scrums".
  - Meeting-ul "Scrum of scrums" este analog meeting-ului Scrum zilnic, dar nu are loc in mod necesar in fiecare zi; in multe organizatii are loc de 2-3 ori pe saptamana.

Scrums of scrums



Sursa: [4]

- Meeting-ul “Scrum of scrums” se desfasoara la fel ca o intalnire Scrum zilnica, fiecare participant raportand ce s-a facut, ce se va face si impedimentele, in numele echipei pe care o reprezinta.
- Impedimentele sunt focalizate pe problemele de coordonare intre echipe.
- Rezolvarea impedimentelor poate consta in: acordul asupra interfețelor dintre echipe, negocierea asupra responsabilitatilor, etc.

Metodologia Scrum a fost folosita de:

- Microsoft
- Yahoo
- Google
- Electronic Arts
- High Moon Studios
- Lockheed Martin
- Philips
- Siemens
- Nokia
- Capital One
- BBC
- Intuit
- Intuit
- Nielsen Media
- First American Real Estate
- BMC Software
- Ipswitch
- John Deere
- Lexis Nexis
- Sabre
- Salesforce.com
- Time Warner
- Turner Broadcasting
- Oce

Sursa [2]

Scrum a fost folosita pentru:

- Software comercial
- Aplicații in-house
- Aplicații la comandă
- Proiecte cu preț fix
- Aplicații financiare
- Aplicații certificate ISO 9001
- Sisteme încorporate (embedded)
- Sisteme cu cerințe de disponibilitate 99.999%, 24x7
- Programul Joint Strike Fighter
- Dezvoltarea de jocuri video
- Sisteme aprobate de FDA, life-critical
- Software de control al sateliților
- Site-uri web
- Software pentru dispozitive mobile
- Aplicații pentru rețele de calculatoare
- Unele din cele mai mari aplicații in uz

Sursa [2]

Mai multe informații

1. Ken Schwaber & Jeff Sutherland, The Scrum Guide, https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf#zoom=100
2. Mike Cohn, "O introducere in Scrum": www.mountaingoatsoftware.com/agile/scrum/a-reusable-scrum-presentation
3. www.mountaingoatsoftware.com/scrum
4. https://www.mountaingoatsoftware.com/agile/scrum/team
5. www.scrumalliance.org
6. www.controlchaos.com
7. Craig Larman, *Agile and Iterative Development: A Manager's Guide,* 2004, Pearson Eucation.
8. Mike Cohn, *Agile Estimating and Planning*, 2006, Prentice Hall.
9. Ken Schwaber, *Agile Project Management with Scrum,* 2004, Microsoft Press.

Abordari în dezvoltarea software
- o privire comparativa -

Prof. univ. dr. ing. Florica Moldoveanu

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

Modele şi metode/metodologii

Cascada	Iterative/Incrementale	Agile
<----------|	----------|	----------|>
Predictive	Exemplu: RUP	**Adaptive**
- Focalizate pe planificarea în detaliu a activităților pentru întreagul proces de dezvoltare. - Numai schimbările considerate importante sunt luate în considerare. - O bună planificare a resurselor umane. - Documentație completă.	- Dezvoltare în mai multe iteratii, cu livrarea unor versiuni intermediare. - Iterațiile, cu cerințele inițiale alocate, sunt planificate la începutul procesului de dezvoltare; o iterație: 1-3 luni. - Schimbările în cerințe şi feedbackul utilizatorilor sunt înglobate în fiecare nouă iterație. - O bună planificare a resurselor umane. - Documentație completă. -Implicarea moderată a clientului.	- Focalizate pe adaptarea rapidă la schimbări. - Perioadele iterațiilor sunt scurte şi fixe. - Livrări frecvente. - Implicare puternică a clientului. - Echipa lucrează într-o manieră colaborativă, prin comunicare directă. - Documentație minimală – focalizare pe cod funcțional. - Rezultatul procesului de dezvoltare este decis printr-un proces de negociere pe parcursul dezvoltării.

Waterfall sau agile pentru un proiect?

❖ **Cele mai importante caracteristici ale proiectelor în care procesul Waterfall este adecvat:**

- Cerințele clientului/utilizatorilor sunt clare de la început.
- Cerințele sunt stabile pe parcursul proiectului sau se schimbă foarte rar.
- Proiectul are o durată scurtă sau bine stabilită şi un buget constrâns.
- Mediul de dezvoltare este stabil.
- Personalul este instruit cu metodele şi instrumentele de dezvoltare.

❖ **Când sunt aplicate metodele agile în dezvoltarea de software?**

- Atunci când se doreşte dezvoltarea rapidă a unui produs software, fără clarificarea tuturor cerințelor de la început, acestea urmând să fie extrase şi implementate pe baza feedback-ului utilizatorilor la versiunile livrate pe parcurs.
- Atunci când se doreşte dezvoltarea personalizată a produsului software, cu angajarea clară a clientului de a deveni implicat în procesul de dezvoltare şi există puține reguli şi reglementări externe care să afecteze produsul software.

Care metodologie este mai potrivită pentru un proiect, dezvoltarea liniară dirijată de plan (Waterfall) sau dezvoltarea agilă?

Statistici Standish Group Chaos Report

(CHAOS: Comprehensive Human Appraisal for Originating Software)

2011-2015 [1]

PROJECT SUCCESS RATES

AGILE VS WATERFALL

METHOD	SUCCESSFUL	CHALLENGED	FAILED
AGILE	39%	52%	9%
WATERFALL	11%	60%	29%

2013-2017 [3]

PROJECT SUCCESS RATES

AGILE VS WATERFALL

METHOD	SUCCESSFUL	CHALLENGED	FAILED
AGILE	42%	50%	8%
WATERFALL	26%	53%	21%

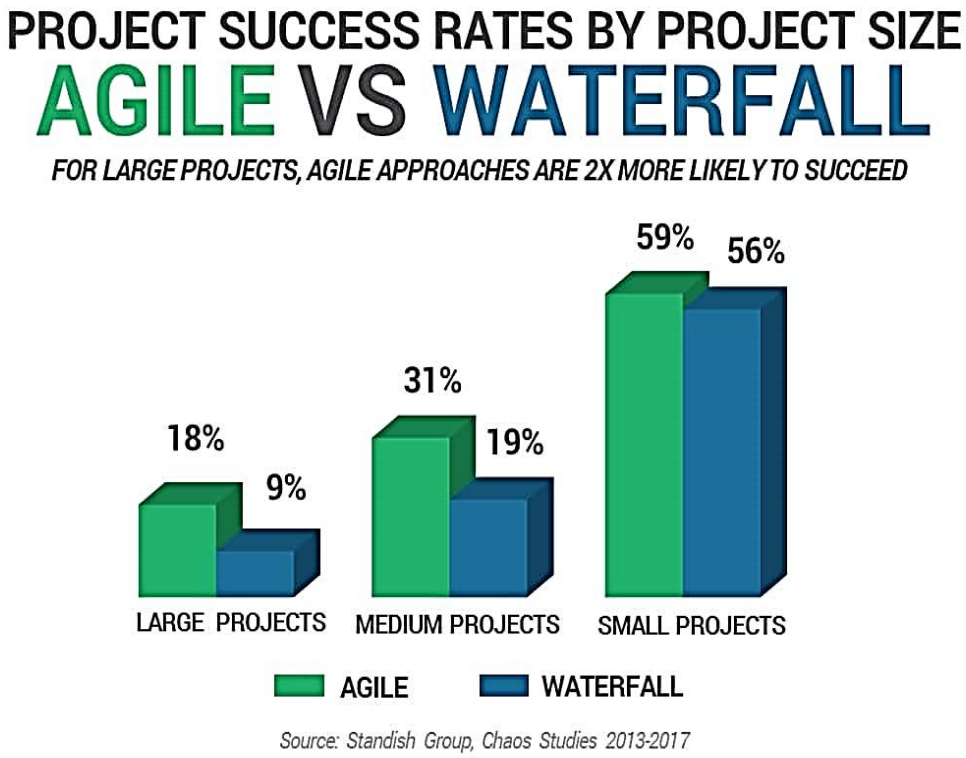
Successful – Proiectul s-a încheiat cu îndeplinirea celor 3 constrângeri: planificare, cost, scop (OnTime, OnBudget, OnTarget).

Challenged – Proiect încheiat şi aprobat, terminat cu întârziere sau cu depăşirea bugetului sau cu functionalități incomplete (una din cele 3 constrângeri neîndeplinită).

Failed – Proiectul a fost anulat înainte de a fi terminat sau a fost terminat dar rezultatul său nu este utilizat.

Observăm ca ratele de succes au crescut în ultimii ani (în special pentru Waterfall)

Statistici Standish Group Chaos Report



2013-2017

[3]

- Proiectele mari au o rată de succes mai mica.
- Pentru proiectele mari, rata de succes a proiectelor agile este de 2 ori mai mare decât a celor Waterfall.

De ce modelul Waterfall este utilizat în prezent?

- Raportul 2017 al Project Management Institute [5] arată că 51% din companii încă utilizează modelul Waterfall.
- Intre beneficiile utilizării modelului Waterfall sunt:
  - Este o abordare simplă
  - Managementul proiectului este simplu: fiecare etapă are o dată de început şi una de sfârşit, permițând o alocare clară în timp a resurselor umane pe activitățile specifice fiecarei etape.
  - Clienții care preferă date exacte de început şi de sfârşit apreciază abordarea Waterfall: le permite să ştie data la care vor avea produsul.
  - Costul dezvoltării poate fi determinat înainte de începerea proiectului.
  - Intregul proces poate fi controlat prin proceduri extrem de detaliate.
  - Este o soluție adecvată pentru clienții care preferă să nu fie implicați în procesul de dezvoltare, ci doar în faza inițială, apoi să primească produsul complet la final.
  - Ideal pentru proiecte mici, unde durata dezvoltării nu este de cea mai mare importanță.
  - Planul de proiect poate fi reutilizat în proiecte viitoare similare, utilizând documentația masivă elaborată anterior.

Dezavantaje ale dezvoltării agile

Predictibilitate scazută

- Pentru anumite livrabile, dezvoltatorul nu poate cuantifica exact efortul necesar, în special la începutul proiectelor mari, ceea ce crează frustrări echipelor începătoare în dezvoltarea agilă.

Mai mult timp şi angajare

- Dezvoltatorii, testerii şi clienții trebuie sa interacționeze direct (face to face) în mod constant pe parcursul dezvoltării.
- Utilizatorii trebuie sa fie disponibili pentru testarea rapidă, înainte de a se trece la următoarea etapă a dezvoltării
- Clientii trebuie sa fie instruiți pentru a ajuta în dezvoltarea produsului. Lipsa de participare a clientului va impacta calitatea produsului şi succesul proiectului.

Lipsa documentației necesare

- Documentația este minimală. Membrii noi în echipă nu cunosc detaliile anumitor functionalități şi cum trebuie realizate, ceea ce crează neînțelegeri şi dificultăți.

Proiectul poate cu uşurință să-şi piardă ținta

- Dezvoltarea agilă porneşte cu planificare minimală şi presupune că cerințele clientului se vor schimba continuu. Dacă feedback-ul utilizatorilor sau comunicarea nu este clară, dezvoltatorul se poate focaliza pe o direcție de dezvoltare greşită.

Concluzii

- Fiecare model de dezvoltare/metodologie are avantaje si dezavantaje.
- NU exista un model de dezvoltare perfect pentru toate tipurile de proiecte şi organizații.
- Alegerea şi adoptarea unui model sau metodologie trebuie făcută în funcție de proiect, echipă, organizație.
- Principalii factori în alegerea modelului de dezvoltare sunt scopul proiectului şi cerințele clientului.

Este necesară înțelegerea tuturor modelelor de dezvoltare pentru:

Alegerea modelului potrivit

Implementarea lui în organizație / proiect

Activitatea eficientă în cadrul unei dezvoltări

Statistici Standish Group Chaos Report
- evoluția ratei de success a proiectelor-

	2011	2012	2013	2014	2015
SUCCESSFUL	39%	37%	41%	36%	36%
CHALLENGED	39%	46%	40%	47%	45%
FAILED	22%	17%	19%	17%	19%

Clasificare pe baza criteriilor: OnTime, OnBudget, OnTarget

[1]

Pe baza unui număr de 25000 proiecte, (în medie 5000 pe an) din CHAOS database

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

Clasificare pe baza criteriilor: OnTime, OnBudget, with satisfactory results.

“Satisfactory results” include OnTarget (ținta management proiect) şi scopul/satisfacția clientului şi utilizatorilor finali.

Statistici Standish Group Chaos Report
- evoluția ratei de success a proiectelor-

Clasificare dupa mărimea proiectelor, pe baza criteriilor: OnTime, OnBudget, with satisfactory results, folosind CHAOS database 2011-2015.

	SUCCESSFUL	CHALLENGED	FAILED
Grand	2%	7%	17%
Large	6%	17%	24%
Medium	9%	26%	31%
Moderate	21%	32%	17%
Small	62%	16%	11%
TOTAL	100%	100%	100%

[1]

Resurse

1. https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf

2. https://www.infoq.com/articles/standish-chaos-2015/

3. https://hennyportman.wordpress.com/2020/01/03/review-chaos-report-2018/

4. https://vitalitychicago.com/blog/agile-projects-are-more-successful-traditional-projects/

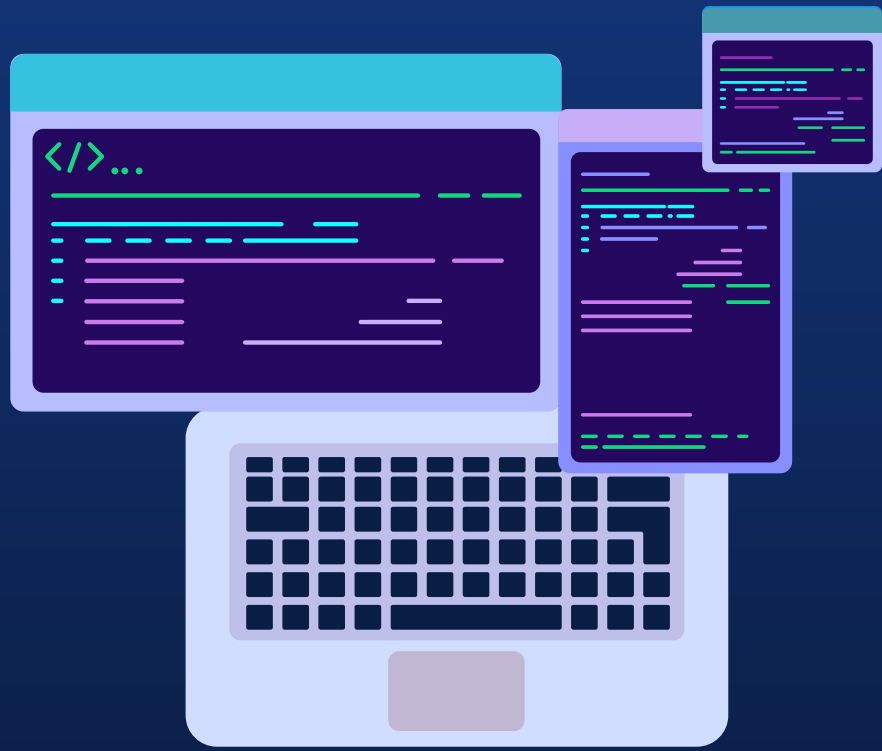
5.https://www.pmi.org/-/media/pmi/documents/public/pdf/learning/thought-leadership/pulse/pulse-of-the-profession-2017.pdf

6. https://www.seguetech.com/waterfall-vs-agile-methodology/

7.https://www.changeboard.com/article-details/17244/nutshell-how-to-pick-a-project-management-methodology-that-works-for-you/

METODOLOGIA AGILE

Filimon Adrian
Olaru Iulian
Slabu Nicusor



Cuprins

01 Istorie si ce este agile?

02 Cum functioneaza Agile?

03 De ce Agile?

04 Manifesto-ul Agile

05 Metodologii Agile

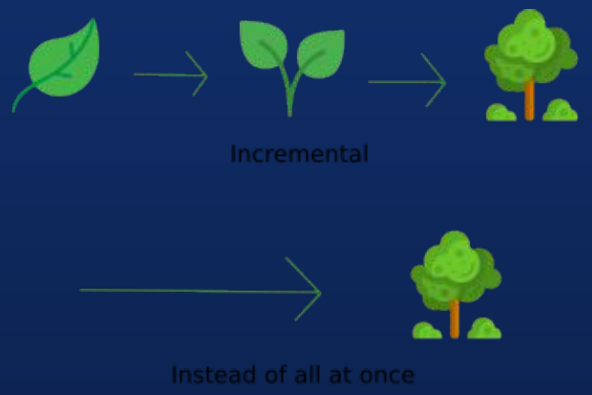
06 Scrum

01

Istorie si ce este Agile?

Agile a devenit o metodologie formala cand 17 pionieri s-au intalnit la statiunea de ski Snowbird din Utah si au emis Manifesto-ul Agile, in anul 2001.

Ce este agile?



Agile este o abordare iterativa de a dezvolta si livra software in mod incremental, de la inceputul proiectului, in defavoarea livrarii produsului complet la finalul ciclului de dezvoltare.

Functioneaza divizand prioectele in bucati mici, asa numitele “user stories”. Aceste bucati sunt prioritizate si livrate constant, la intervale mici.

02

Cum functioneaza?

1. Intocmirea unei liste
   - Discutand cu clientul, se stabilesc functionalitatile dorite. Acestea vor deveni “user stories”
2. Estimarea lucrurilor
   - Se estimeaza cat de mult va dura fiecare “story” si dimensiunea acesteia. Aceste “user stories” trebuie raportate tot timpul una la cealalalta.
3. Stabilirea prioritatilor
   - In cazul in care vor fi mai multe prioritati decat timpul permite, clientul este rugat sa prioritizeze functionalitatile cele mai importante, lasandu-le pe celelalte la final.

4. Inceperea executiei

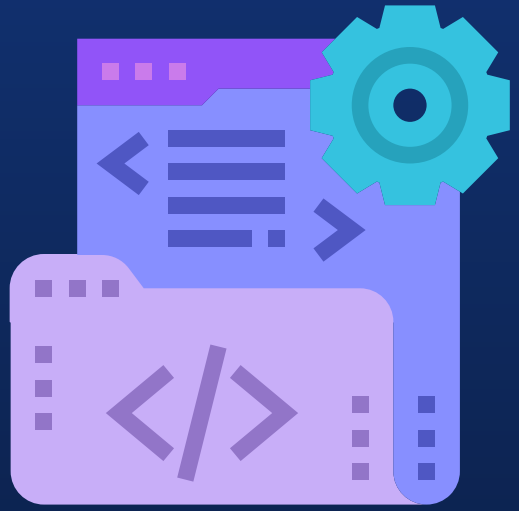
- Livrezi produsul clientului, cerand constant feedback, intr-o maniera top to bottom.

5. Actualizarea planului pe parcurs

- Pe parcursul procesului, daca timpii estimati au fost destul de buni, nu vor aparea probleme. Daca nu este suficient timp pentru atingerea obiectivelor, este recomandat sa reduci amploarea proiectului. Se poate cere si extinderea scadentei cu compensatii financiare mai mari.

03

De ce Agile?



Imbunatatire implicare client

Release-uri facilitate

Crestere in calitate

Reduce riscul

Avantajele modelului Agile

Satisfacerea clientului, prin livrari rapide si constante.

Interactiunea sporita intre membrii echipei si gradul sporit de comunicare

Software functional livrat la cateva saptamani

Cooperare zilnica intre bussiness side si developer side

Atentie sporita asupra design-ului si a calitatii

Adaptare Regulata

Flexibilitate la schimbari de ultima ora

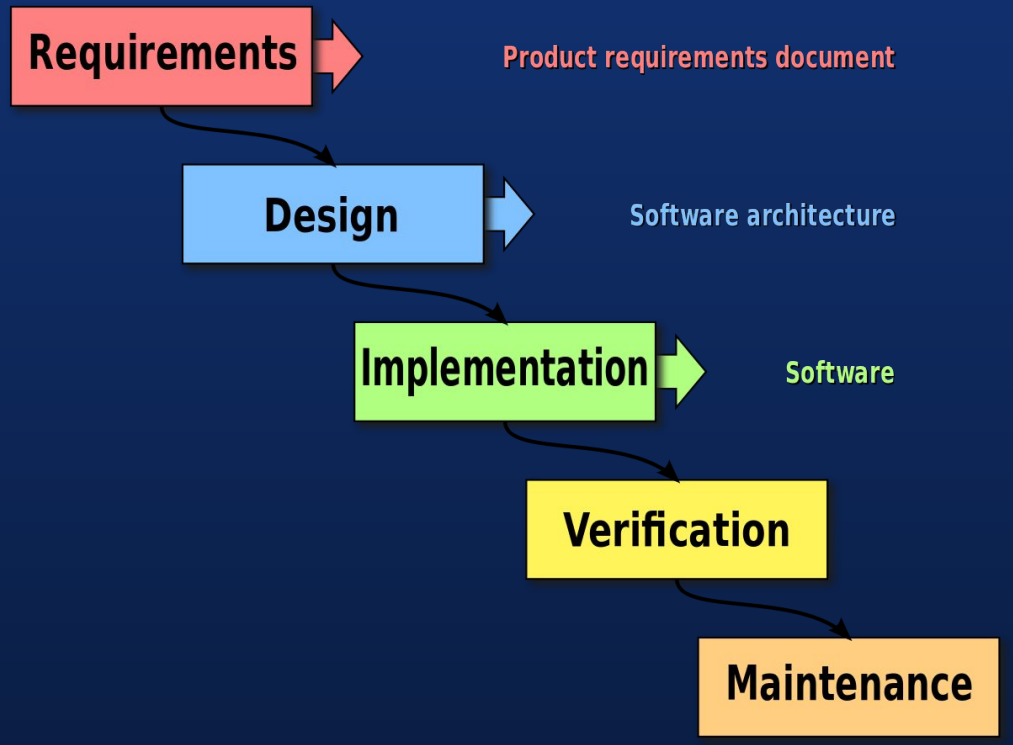
Dezavantajele modelului Agile

Este dificil de estimat efortul necesar pentru dezvoltarea livrabilelor in cadrul unor componente mari.

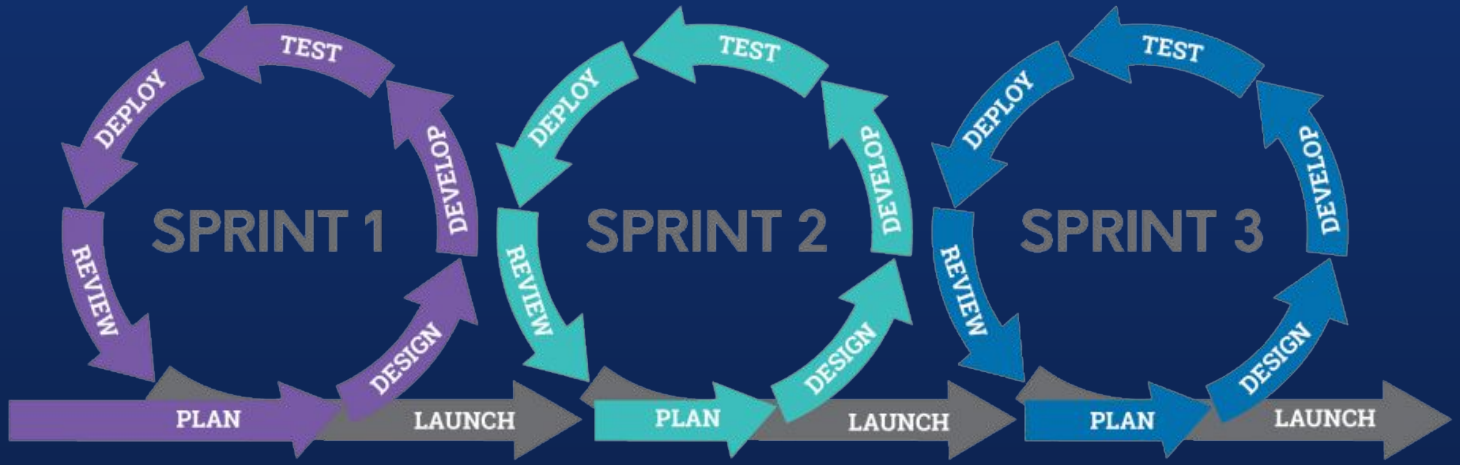
Este nevoie de programatori cu experienta pentru a fi capabili sa ia deciziile corecte pe carcursul dezvoltarii.

Proiectul poate deraia usor daca feedback-ul de la client este neclar.

WATERFALL

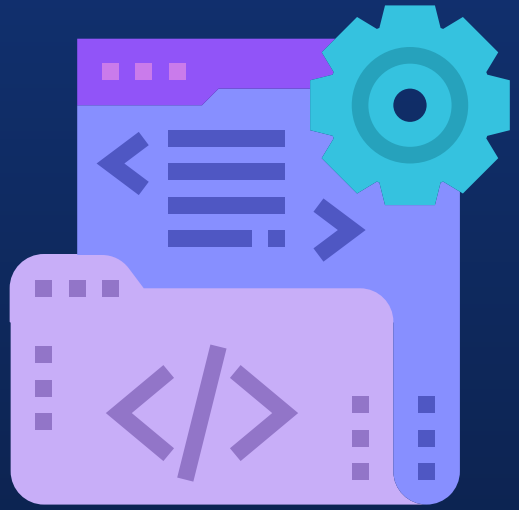

Requirements
Product requirements document
Design
Software architecture
Implementation
Software
Verification
Maintenance

AGILE


SPRINT 1
TEST
DEVELOP
DESIGN
LAUNCH
PLAN
REVIEW
DEPLOY
SPRINT 2
TEST
DEVELOP
DESIGN
LAUNCH
PLAN
REVIEW
DEPLOY
SPRINT 3
TEST
DEVELOP
DESIGN
LAUNCH
PLAN
REVIEW
DEPLOY

04

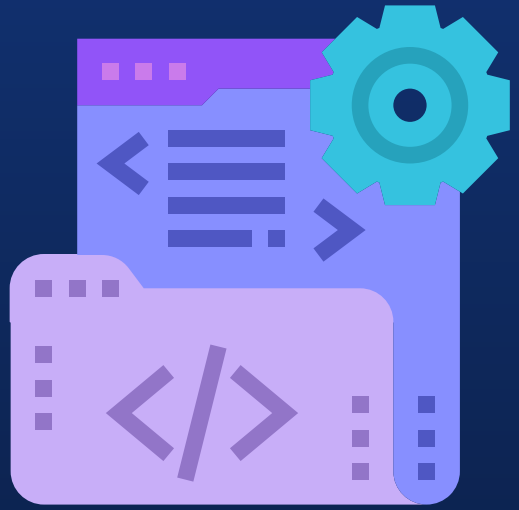
Manifesto-ul Agile

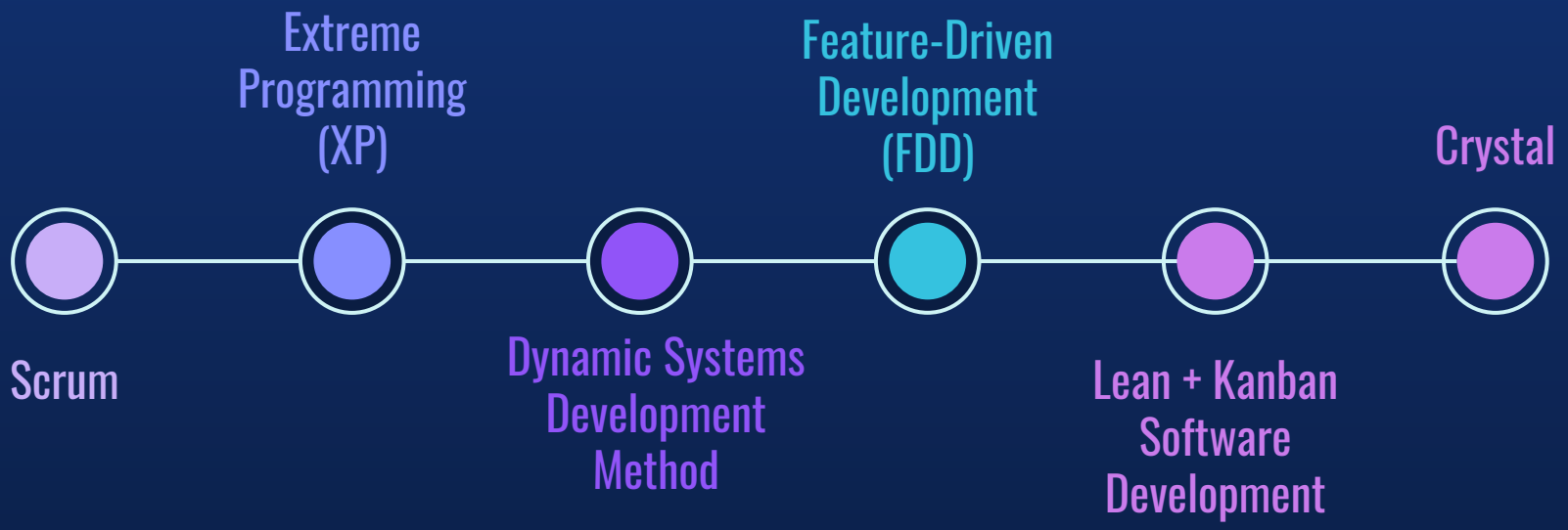


Individual and Interactions	⟷	Over Processed and Tools
Working Software	⟷	Over Comprehensive Documentation
Customer Collaboration	⟷	Over Contract Negotiation
Responding to Change	⟷	Over Following a Plan

05

Metodologii Agile

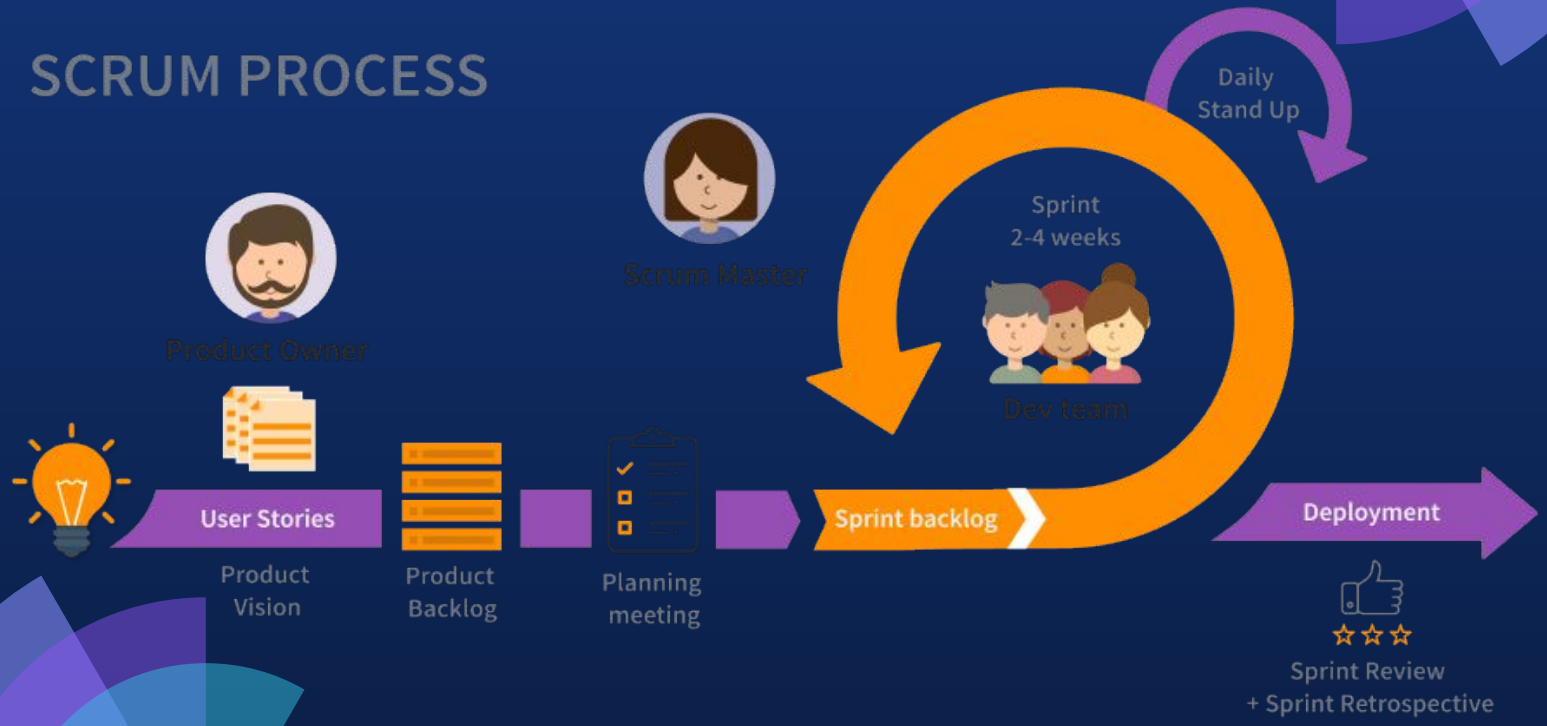


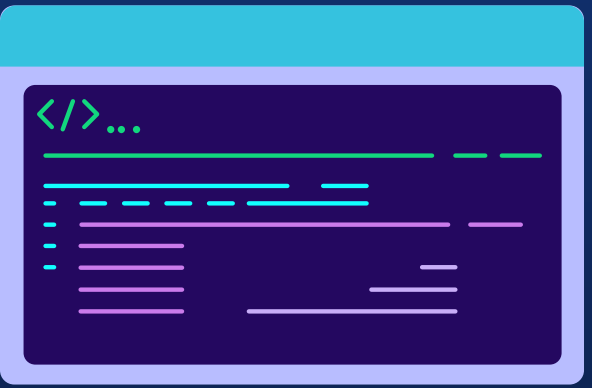

Extreme Programming (XP)
Feature-Driven Development (FDD)
Crystal
Scrum
Dynamic Systems Development Method
Lean + Kanban Software Development

06

Scrum

SCRUM PROCESS





MULTUMIM!

Calitatea produselor software (Software Quality)

Prof. univ. dr. ing. Florica Moldoveanu

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

Calitatea produselor software (1)

- **Calitatea unui produs** este uneori definita ca “ totalitatea caracteristicilor prin care el satisface o serie de necesitati definite sau impuse”.
- **Calitatea unui produs software** este data de «capacitatea sa de a putea fi utilizat eficient, efectiv si confortabil, de catre un set de utilizatori, pentru un set de scopuri, în conditii specificate».
- Caracteristicile de calitate ale unui produs software sunt proprietati ale produsului la care utilizatorii sunt sensibili. De exemplu : uşurinta de utilizare, fiabilitatea, timpul de raspuns, s. a.
- Exista diferite **modele de clasificare a caracteristicilor (atributelor) de calitate** ale produselor software. Modelele includ adesea si **masuri cantitative (metrici)** pe baza carora se stabileste gradul in care produsul intruneste fiecare atribut de calitate.

Calitatea produselor software(2)

- **Definita diferit in raport cu rolurile si asteptarile diferitelor persoane**
- **2 vederi/ grupuri de persoane:**

- **Vederea externa (comportarea externa, observata): black box view**

Consumatori ai produselor sau serviciilor software:

- Clienti – responsabili pentru achizitia produselor software
- Utilizatori – cei care utilizeaza produsele software, intregul mediu operational al produsului software

- **Vederea interna (caracteristicile interne ale produsului): white box view**

Producatori software: orice persoana implicata in dezvoltare, management, intretinere, marketing, certificare, verificare si validare, etc.

Calitatea produselor software(3)

Calitatea software din perspectiva consumatorului - calitatea externa

- **Functionalitatea:** produsul indeplineste functiile specificate, care satisfac necesitatile utilizatorilor.
- **Fiabilitatea**: produsul indeplineste aceste functii corect, in utilizari repetate sau in decursul unei perioade lungi de timp - indeplineste functiile in mod fiabil.
- **Uzabilitatea, usurinta de instalare, interoperabilitatea cu alte sisteme, adaptabilitatea, portabilitatea**, s.a.

Calitatea software din perspectiva producatorului - calitatea interna

- Principala calitate: **produsul software este conform cu specificatia -** furnizeaza servicii conform contractului.
- Pentru manageri: alegerea unui proces de dezvoltare adecvat, a metodologiilor, limbajelor si instrumentelor adecvate, alti factori legati de calitate.
- Pentru cei implicati in intretinere: produsul să aibă o arhitectură usor de modificat si testat, etc.

Atribute de calitate interne

Vederea interna asupra calitatii software include:

- Atribute de calitate ale produsului (calitatea arhitecturii, calitatea codului)
- Atribute de calitate ale procesului de dezvoltare

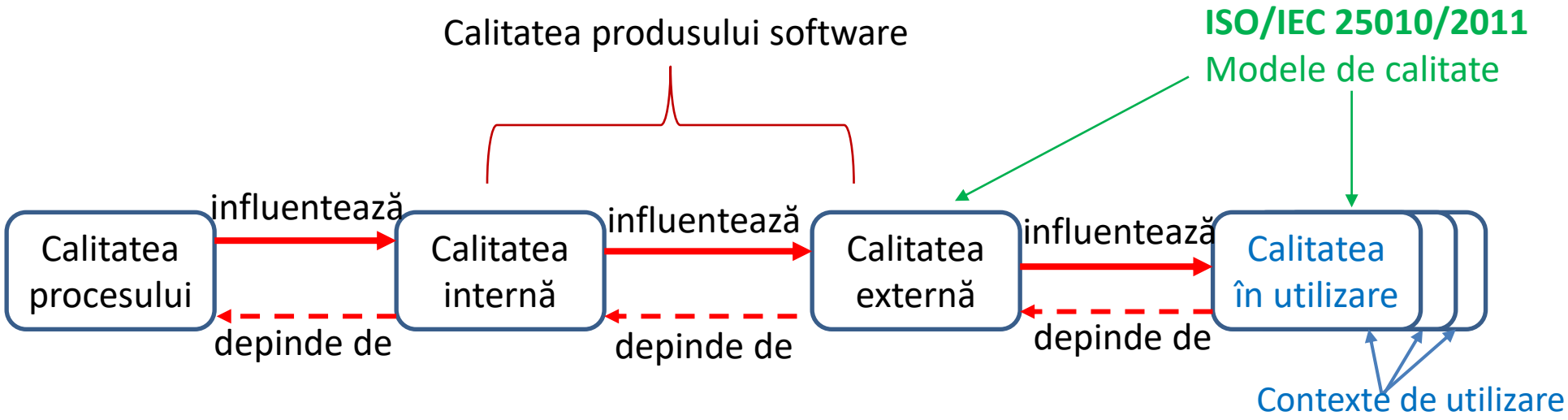
Atribute de calitate ale produsului:

- Rezistenta arhitecturii la modificari
- Usurinta testarii: subsisteme cat mai independente, modularizare
- Usurinta de modificare: cat de usor este sa se adauge cod sau sa se modifice codul a.i. sa nu se introduca defecte
- Usurinta de intelegere a codului
- Eficienta codului, atunci cand exista constrangeri de resurse
- Securitatea: sunt incluse mecanisme pentru prevenirea atacurilor, a accesului neautorizat la resurse, etc.

Atribute de calitate ale procesului

- Respectarea termenelor de livrare
- Respectarea costurilor estimate
- Repetabilitatea in obtinerea de produse de calitate

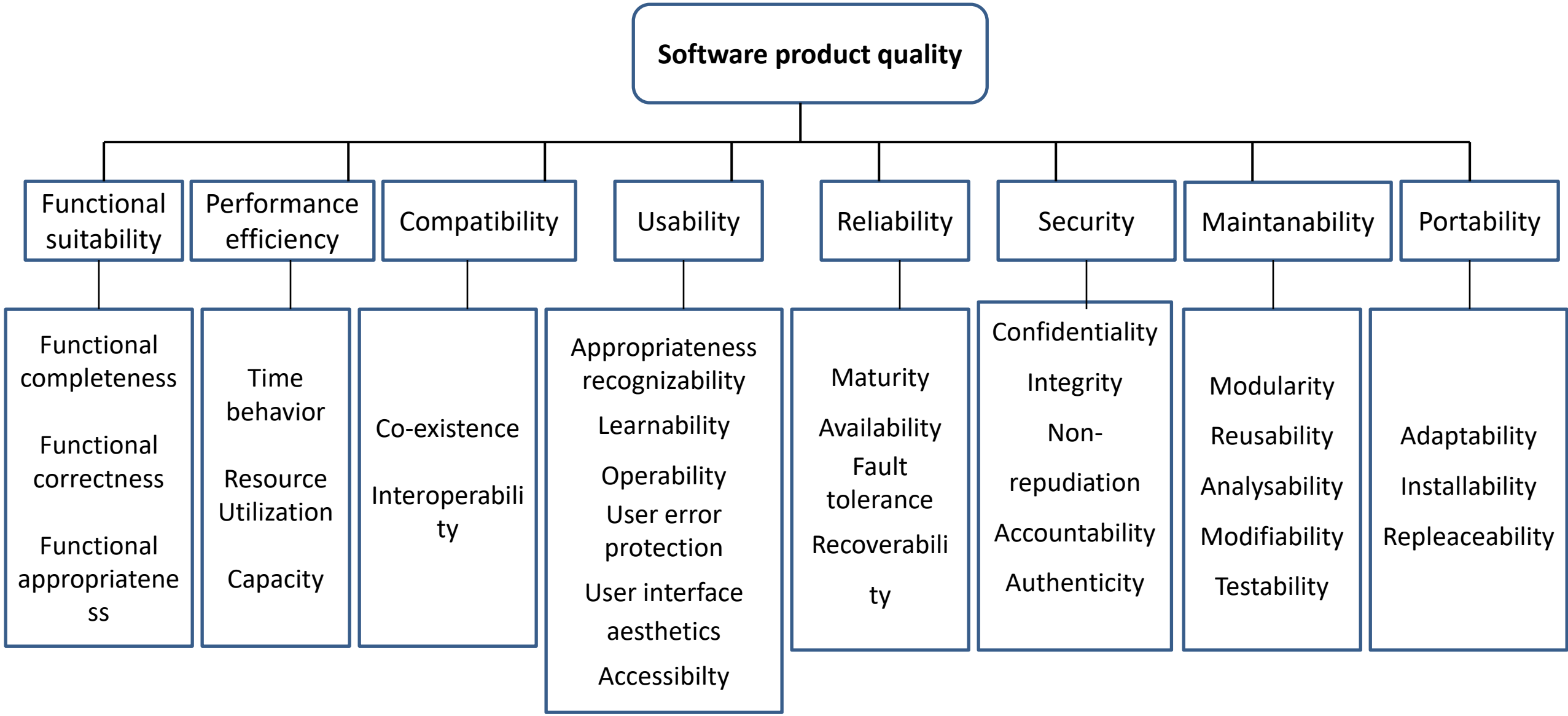
Calitatea software



Calitatea in utilizare: calitatea perceputa de utilizatori atunci cand produsul este folosit în viata reală, în diferite contexte de utilizare. Exemple: eficacitatea, satisfactia, lipsa riscului in utilizarea, acoperirea contextelor de utilizare.

- Calitatea interna influenteaza dar nu determina calitatea externa:
  - calitatea arhitecturii influenteaza mentenabilitatea
  - usurinta testarii influenteaza fiabilitatea
  - calitatea codului nu determina calitati externe ca: uzabilitatea, securitatea, portabilitatea
- Calitatea externa depinde de calitatea interna:
  - Fiabilitatea depinde de usurinta testarii
  - Mentenabilitatea depinde de calitatea arhitecturii

ISO/IEC 25010/2011 - Modelul de calitate al produsului software -1



Modelul de calitate ISO/IEC 25010/2011 -2

Standardul ISO/IEC 25010/2011 [2] defineste un model de calitate al produselor software alcatuit din 8 caracteristici de calitate externe de nivel înalt şi 31 subcaracteristici.

Organizat în doua niveluri ierarhice. Pe primul nivel sunt cele 8 caracteristici de calitate de nivel înalt: Potentialitate functională, Eficienta performantei, Compatibilitatea, Uzabilitatea, Fiabilitatea, Securitatea, Uşurinta de întretinere, Portabilitatea. Fiecare caracteristică are un set propriu de sub-caracteristici.

- Fiecare caracteristica de calitate de nivel inalt se evalueaza pe baza gradului de îndeplinire a subcaracteristicilor sale.

1. **Potentialitate funcțională**: Capacitatea produsului de a furniza functii care îndeplinesc nevoile declarate și implicite atunci când sunt utilizate în condiții specificate.

Subcaracteristici:

- **Completitudine funcțională** : gradul în care produsul furnizează toate funcțiile specificate de utilizator
- **Corectitudinea funcțională**: gradul în care produsul furnizează rezultate corecte, cu precizia dorita de utilizator
- **Adecvarea funcțională**: gradul în care produsul furnizează numai functiile necesare pentru ca sistemul sa îndeplineasca sarcinile si obiectivele specificate de utilizator

Modelul de calitate ISO/IEC 25010/2011 -3

2. Eficiența performanței: performanța în raport cu cantitatea de resurse utilizate în condițiile specificate.

Resursele pot include alte produse software din configurația software - hardware a sistemului și materiale (de exemplu, hârtie de imprimare, suport de stocare).

Subcaracteristici:

- **Comportamentul în timp**: gradul în care produsul (sistemul) îndeplineşte cerințele privind timpii de răspuns și de procesare și ratele de prelucrare (throughput), atunci când își realizează funcțiile.
- **Utilizarea resurselor:** gradul în care produsul (sistemul) îndeplineşte cerințele privind cantitățile și tipurile de resurse utilizate, atunci când își realizează funcțiile.
- **Capacitatea:** gradul în care produsul (sistemul) îndeplineşte cerințele privind limitele maxime ale unor parametri.

Parametrii pot include: numărul de articole care pot fi stocate, numărul de utilizatori concurenți, lățimea de bandă a comunicației, randamentul tranzacțiilor și dimensiunea bazei de date.

Modelul de calitate ISO/IEC 25010/2011 - 4

3. Compatibilitatea: gradul în care produsul, sistemul sau o componentă poate schimba informații cu alte produse, sisteme sau componente și / sau își poate îndeplini funcțiile necesare, partajând același mediu hardware sau software cu ele.

Subcaracteristici:

- **Coexistenţa**: gradul în care produsul își poate îndeplini funcțiile necesare în mod eficient, partajând în același timp un mediu și resurse comune cu alte produse, fără impact negativ asupra oricărui alt produs.
- I**nteroperabilitatea**: gradul în care produsul/sistemul poate schimba informații cu alte produse/sisteme și poate utiliza informațiile care au fost schimbate.

Modelul de calitate ISO/IEC 25010/2011 - 5

4. Uzabilitatea: gradul în care produsul poate fi utilizat de către utilizatori specificați pentru a atinge obiectivele specificate, cu eficacitate, eficiență și satisfacție într-un context specific de utilizare.

Uzabilitatea poate fi specificată ca o caracteristică a calității produsului şi măsurată prin subcaracteristicile sale, sau specificată şi măsurată ca un subset de caracteristici ale calității în utilizare.

Subcaracteristici:

- **Recunoașterea adecvării:** gradul în care utilizatorii pot recunoaște dacă produsul este adecvat nevoilor lor.
  - depinde de impresiile inițiale ale produsului, și / sau orice documentație, demonstratie, tutoriale
- **Uşurința de învățare**: gradul în care produsul poate fi utilizat de către utilizatori specificați pentru a atinge obiective specifice de învățare a utilizării produsului cu eficacitate, eficiență, lipsa de risc și satisfacție într-un context specific de utilizare.
- **Operabilitate**: gradul în care un produs sau sistem are atribute care îl fac ușor de operat și de controlat
- **Protecția împotriva erorilor utilizatorului**: gradul în care un sistem protejează utilizatorii împotriva erorilor

Modelul de calitate ISO/IEC 25010/2011 - 6

Subcaracteristici uzabilitate – continuare

- **Estetica interfeței cu utilizatorul:** gradul în care o interfață cu utilizatorul permite interacțiuni plăcute și satisfăcătoare pentru utilizator
- **Accesibilitatea:** gradul în care produsul poate fi utilizat de persoane cu cea mai largă gamă de caracteristici și capacități pentru a atinge un obiectiv specificat într-un context specific de utilizare
  - Accesibilitatea pentru persoanele cu dizabilități poate fi specificată sau măsurată fie în măsura în care produsul poate fi utilizat de utilizatorii cu dizabilități specificate, pentru a atinge obiectivele specificate cu eficacitate, eficiență, lipsa de risc și satisfacție într-un anumit domeniu de utilizare specificat, fie prin prezența proprietăților produsului care permit accesibilitatea.

Modelul de calitate ISO/IEC 25010/2011 - 7

5. Fiabilitatea: gradul în care produsul îndeplinește funcții specificate în condiții specificate pentru o perioadă de timp specificată

Subcaracteristici:

- **Maturitatea:** gradul în care produsul satisface nevoile de fiabilitate în condiții normale de funcționare.
- **Toleranța la caderi:** gradul în care produsul funcționează conform intenției, în ciuda prezenței defecțiunilor hardware sau software
- **Disponibilitatea:** gradul în care produsul este operațional și accesibil atunci când este necesar pentru utilizare
  - disponibilitatea poate fi evaluată prin proporția timpului total în care produsul se află într-o stare activa, intr-o perioada de timp specifcata.
  - depinde de maturitate (frecvența caderilor), toleranța la caderi și recuperabilitate (durata timpului de nefuncționare după fiecare cadere).
- **Recuperabilitate:** gradul în care, în caz de întrerupere sau cadere, produsul/sistemul poate recupera datele direct afectate și restabili starea dorită a sistemului
  - În urma unei defecțiuni, sistemul va fi uneori oprit pentru o perioadă de timp, a cărei lungime este determinată de recuperabilitatea sa.

Modelul de calitate ISO/IEC 25010/2011 - 8

6. Securitatea: gradul în care produsul/sistemul protejează informațiile și datele, astfel încât persoanele sau alte produse sau sisteme să aibă gradul de acces la date adecvat tipurilor și nivelurilor lor de autorizare

- Securitatea se aplica nu numai datelor stocate în produs sau sistem, ci și datelor transmise.
- Securitatea contribuie la încredere.

Subcaracteristici:

- **Confidențialitatea:** gradul în care produsul/sistemul asigură că datele sunt accesibile numai celor autorizați să aibă access.
- **Integritatea:** gradul în care produsul/sistemul împiedică accesul neautorizat sau modificarea programelor sau datelor.
- **Non-repudierea (respingere)**: gradul în care se poate dovedi că au avut loc acțiuni sau evenimente, astfel încât evenimentele sau acțiunile nu pot fi respinse ulterior.
- **Responsabilitatea:** gradul în care acțiunile unei entități pot fi asociate în mod unic cu entitate a
- **Autenticitatea:** gradul în care identitatea unui subiect sau a unei resurse poate fi dovedită a fi cea revendicată

Modelul de calitate ISO/IEC 25010/2011 - 9

7. Mentenabilitatea: gradul de eficacitate și eficiență în care produsul/sistemul poate fi modificat.

- Modificările pot include corecții, îmbunătățiri sau adaptarea produsului la schimbările de mediu, la cerințe și specificații funcționale.
- Modificările includ cele efectuate de personalul de asistență specializat și cele efectuate de personalul operațional sau de utilizatorii finali.
- Mentenabilitatea include instalarea de actualizări și upgrade-uri.

Subcaracteristici:

– **Modularitatea:** gradul în care produsul este compus din componente discrete, astfel încât o modificare a unei componente are un impact minim asupra altor componente.

– **Reutilizarea:** gradul în care produsul sau componentele sale pot fi utilizate în mai multe sisteme sau în construirea altor sisteme.

– **Analizabilitatea**: uşurința de evaluare a impactului asupra produsului/sistemului a unei modificări intenționate, sau de diagnosticare a deficiențelor sau cauze ale defecțiunilor şi identificare a părților care trebuie să fie modificate.

Modelul de calitate ISO/IEC 25010/2011 - 10

Subcaracteristici mentenabilitate - continuare:

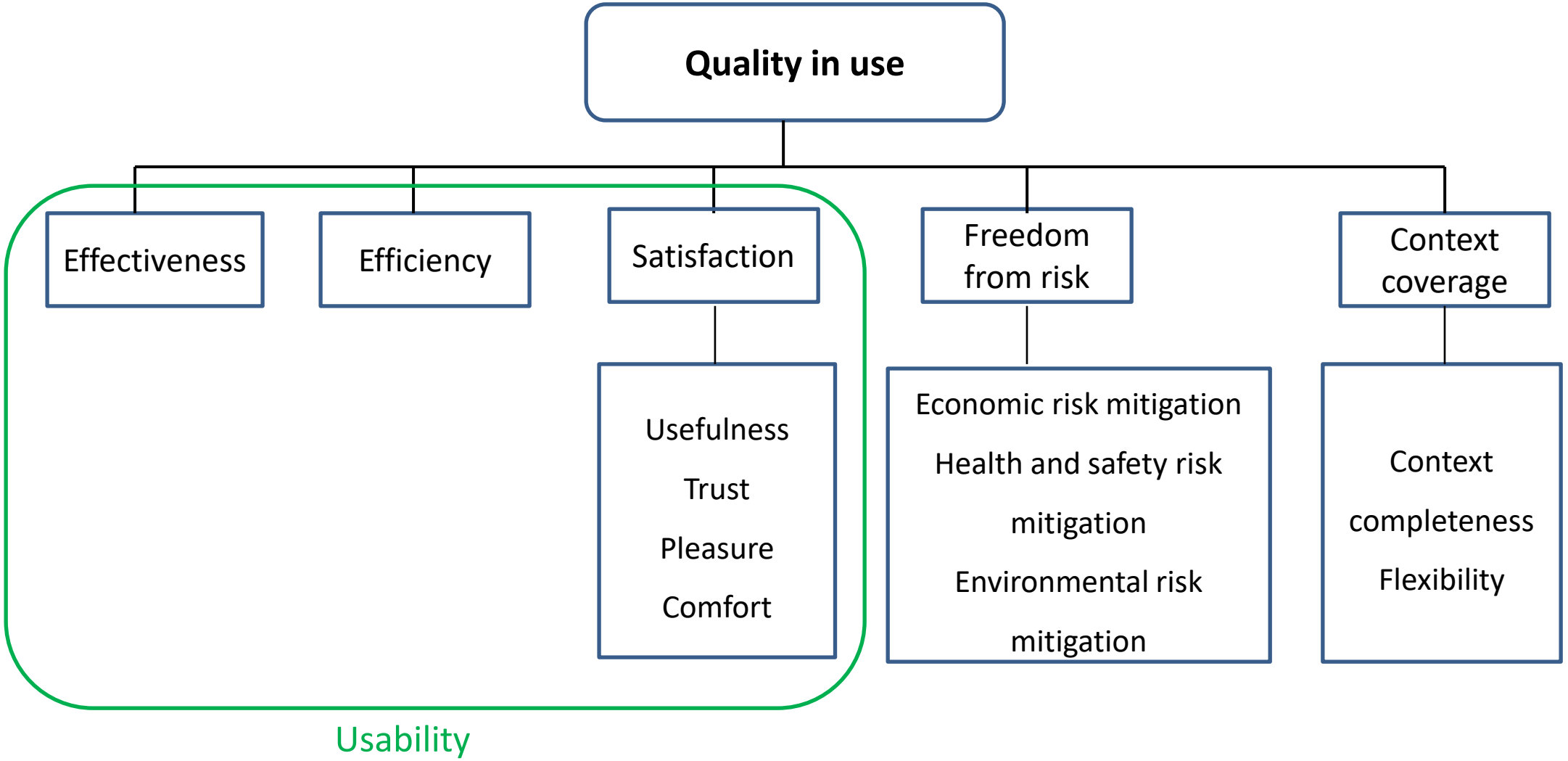
- **Modificabilitatea:** gradul în care un produs sau un sistem poate fi modificat în mod efectiv și eficient fără a introduce defecte sau a produce degradarea calității produsului existent.
- **Testabilitatea:** gradul de eficacitate și eficiență în care pot fi stabilite criteriile de testare pentru produs/sistem și pot fi efectuate teste pentru a determina dacă aceste criterii au fost îndeplinite.

8. Portabilitatea: gradul de eficacitate și eficiență în care produsul/sistemul poate fi transferat pe un alt hardware, software sau mediu operațional.

Subcaracteristici:

- **Adaptabilitatea:** gradul în care produsul/sistemul poate fi adaptat în mod efectiv și eficient pentru hardware, software sau alte medii operaționale sau de utilizare diferite sau în evoluție.
- **Instalabilitatea:** gradul de eficacitate și eficiență în care produsul/sistemul poate fi instalat și / sau dezinstalat cu succes într-un mediu specificat.
- **Înlocuirea**: gradul în care produsul poate înlocui un alt produs software specificat în același scop şi în același mediu, inclusiv o noua versiune a sa.

ISO/IEC 25010/2011 - Modelul “Calitatea in utilizare” - 1



ISO/IEC 25010/2011 - Modelul “Calitatea in utilizare” - 2

Calitatea in utilizare: gradul in care produsul software poate fi utilizat de utilizatori specifici pentru a-și atinge obiective specifice cu eficacitate, eficiență, lipsa de risc și satisfacție în contexte specifice de utilizare.

- Uzabilitatea este definita atat ca o caracteristica de calitate a produsului software cat si ca un subset de caracteristici de calitate in utilizare.

1. **Eficacitatea**: precizia și completitudinea in atingerea obiectivelor specificate, de catre utilizatori
2. **Eficiența**: resursele cheltuite în raport cu acuratețea și completitudinea in atingerea obiectivelor de catre utilizatori
3. **Satisfacția**: gradul in care produsul satisface necesitatile utilizatorilor atunci cand este utilizat intr-un context specificat

Subcaracteristici:

- **Utilitatea**: gradul în care un utilizator este mulțumit de realizarea percepută a obiectivelor practice, inclusiv rezultatele utilizării și consecințele utilizării
- **Încrederea**: gradul în care un utilizator sau un alt factor interesat are încredere că produsul se va comporta conform intenției.

ISO/IEC 25010/2011 - Modelul “Calitatea in utilizare” - 3

- **Plăcerea**: gradul în care un utilizator obține plăcerea de a-și îndeplini nevoile personale (ex. Nevoia de a capata noi cunostinte si abilitati, etc).
- **Confortul:** gradul în care utilizatorul este mulțumit de confortul fizic.

4. Lipsa riscului: gradul în care produsul reduce riscul potențial pentru statutul economic, viața umană, sănătatea sau mediul

Subcaracteristici:

- **Atenuarea riscului economic** : gradul în care produsul reduce riscul potențial pentru starea financiară, funcționarea eficientă, proprietatea comercială, reputația sau alte resurse în contextele de utilizare prevăzute.
- **Atenuarea riscului pentru sănătate și siguranță** : gradul în care produsul reduce riscul potențial pentru oameni în contextele de utilizare prevăzute.
- **Atenuarea riscului de mediu**: gradul în care produsul reduce riscul potențial pentru proprietate sau mediu în contextele de utilizare prevăzute.

ISO/IEC 25010/2011 - Modelul “Calitatea in utilizare” - 4

5. **Acoperirea contextului**: gradul în care produsul poate fi utilizat cu eficacitate, eficiență, lipsă de risc și satisfacție atât în contextele de utilizare specificate cât și în contexte diferite de cele identificate inițial în mod explicit.

Subcaracteristici:

– **Completitudinea contextului**: gradul în care produsul poate fi utilizat cu eficacitate, eficiență, lipsă de risc și satisfacție în toate contextele de utilizare specificate.

– **Flexibilitatea**: gradul în care produsul poate fi utilizat cu eficacitate, eficiență, lipsă de risc și satisfacție în contexte diferite de cele specificate

Flexibilitatea permite adaptarea unui produs pentru grupuri de utilizatori, sarcini și preferințe individuale care nu au fost anticipate în prealabil.

Alte modele de calitate (1)

Modele produse de diferite companii, adaptate la mediul lor de afaceri si marketing:

- Corespund vederii externe asupra produselor software
- Fiecare caracteristica de calitate este evaluata din punct de vedere al satisfactiei utilizatorilor, pe o scara de la 1 la 5: foarte satisfacut, satisfacut, neutral, nesatisfacut, foarte nesatisfacut

IBM: CUPRIMDS (Capability, Usability, Performance, Reliability, Installation, Maintenance, Documentation, and Service)

- Service: capacitatea organizatiei de a mentine produsul in folosinţă

CUPRIMDSO = CUPRIMSD + Overall customer satisfaction

Hewlett-Packard: FURPS (functionality, usability, reliability, performance and service)

Pentru applicatiile Web a fost identificat un set de atribute compus din:

- Atribute primare: fiabilitate, usurinta de utilizare, si securitate
- Atribute secundare: disponibilitatea, scalabilitatea, usurinta de intretinere.

Pentru aplicatiile timp-real:

- Performanta / eficienta si fiabilitatea sunt mai importante decat usurinta de utilizare si de intretinere, **invers decat pentru aplicatiile comerciale.**

Alte modele de calitate(2)

Pentru sisteme ale caror caderi pot avea consecinte extrem de severe:

- **Gradul de incredere al sistemului in ansamblul sau** (hardware, software, oameni) este scopul principal, in plus fata de cel de realizare a functiilor de baza.

  Un grad inalt de incredere include atribute ca: toleranta la defecte, siguranta in functionare, securitatea, usurinta de utilizare.

Pentru sisteme inteligente si bazate pe cunostinte:

- Proprietatea **"oricand" (anytime)**: garanteaza raspunsul cel mai bun care poate fi obtinut intr-un interval de timp specificat
- **Capacitatea de explicare** (explică procesul de gandire la furnizarea unui raspuns).

Pentru sisteme de interfaţă om-calculator si de interactiune

- **Usurinta de adaptare la trasaturile si interesele utilizatorilor, Help inteligent, s.a.**

Caracteristici de calitate software care afecteaza procesul de inginerie software:

- Stilul codului
- Reutilizabilitatea codului
- Modularitatea codului si independenta modulelor

Bibliografie

1. https://en.wikipedia.org/wiki/ISO/IEC_9126

2. ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en

3. Stephen H. Kan. Metrics and Models in Software Quality Engineering, 2nd Edition. Published Sep 16, 2002 by Addison-Wesley Professional.

4. Software Quality Models (Feb 25, 2020): https://www.professionalqa.com/comparison-quality-models