



# Fundamente de Inginerie Software

## Cap. 1

### Introducere în Inginerie Software

Conf.Dr.Ing. Dan Pescaru

2022



# Organizare curs

---

## Curs inginerie software

- Conf. Dan Pescaru
- Marti 8:00 A109
- Email: [dan.pescaru@cs.upt.ro](mailto:dan.pescaru@cs.upt.ro)
- Materiale curs: <https://cv.upt.ro>

## Laborator inginerie software

- B623
- Miercuri 14,16,18; Joi 16, 18
- Materiale laborator, teme: <https://cv.upt.ro>

## Evaluare - notare

- Examen pe [cv.upt.ro](https://cv.upt.ro): 2/3
  - Nota activitati practice: 1/3
-

# Cuprins

---

- 1. Introducere.**
  - 2. Ciclul de Viață a unui Sistem Software. Modele Specifice Fazelor Ciclului de Viață.**
  - 3. Ingineria Cerințelor.**
  - 4. Limbaje de Modelare Software.**
  - 5. Proiectarea Sistemelor Software. Principii de Inginerie Software.**
  - 6. Dezvoltarea Sistemelor Software.**
  - 7. Testarea si Validarea unui Sistem Software.**
  - 8. Managementul Proiectelor Software.**
-

# Introducere

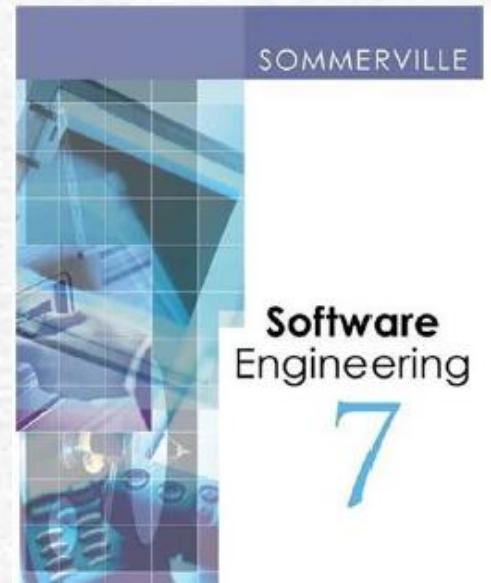
**“Software engineering is the field of computer science that deals with the building of software systems that are so large or so complex that they are built by a team or teams of engineers.”**

Ghezzi et al. '2003

- Ingineria Software se ocupă de dezvoltarea (și mențenanța) Sistemelor Software Complexe
- The Chaos Report 2003: doar **34%** din proiectele software au fost terminate la timp și în bugetul alocat

# Inginerie Software

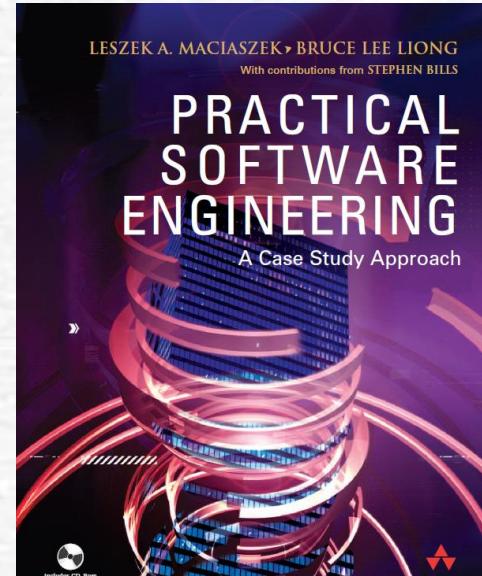
- Întrebări fundamentale despre ingineria software (Sommerville'07):
  - Ce este Ingineria Software (IS)?
    - "Ingineria Software este o disciplină inginerească care se ocupă cu toate aspectele producției software."
  - Care este deosebirea dintre IS și CS?
    - "Știința Claculatoarelor (CS) se ocupă cu teoria iar IS se ocupă cu practica dezvoltării sistemelor software industriale."
  - Care sunt problemele importante ale IS?
    - "Stăpânirea diversității, reducerea timpului de livrare și creșterea încrederii în sistemele dezvoltate."



# Inginerie Software

## ➤ Fațete ale Inginieriei Software (Maciaszek '05):

- Modele și faze de dezvoltare
- Limbaje de modelare (ex. UML)
- Unelte de inginerie software
- Planificare (ex. timp, buget etc.)
- Management  
(ex. personal, risc etc.)



# Dezvoltare Sisteme Software

---

## ➤ Fazele de dezvoltare ale unui sistem software:

- **Analiza** cerințelor
- **Proiectarea** sistemului
- **Implementarea**
- Instalarea și **integrarea**
- Operarea și **mențenanța**

## ➤ Faze independente, aplicate pe toate nivelele:

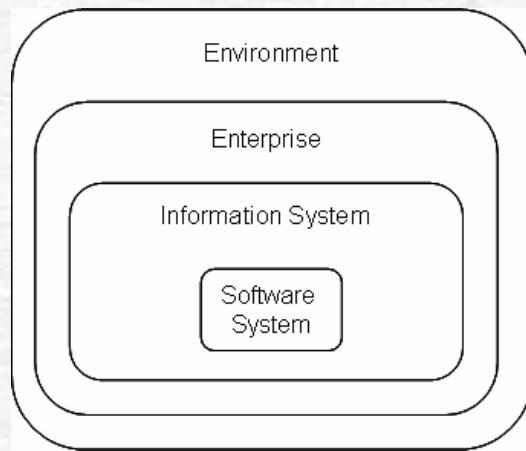
- Testarea și validarea
- Managementul

# Observații

## Observații esențiale referitoare la ingineria software:

- Sisteme software sunt incluse în sisteme informaticale ale companiilor (fără să se substituie acestora)
- Software-ul face parte și este subordonat afacerii
- Ingineria software este diferită de alte ramuri inginerești
- Ingineria software înseamnă mai mult decât programare
- Ingineria software se sprijină pe modelare
- Sistemele software sunt sisteme complexe

# **Sisteme software sunt incluse în sistemele informaticice ale companiilor (fără să se substituie acestora)**

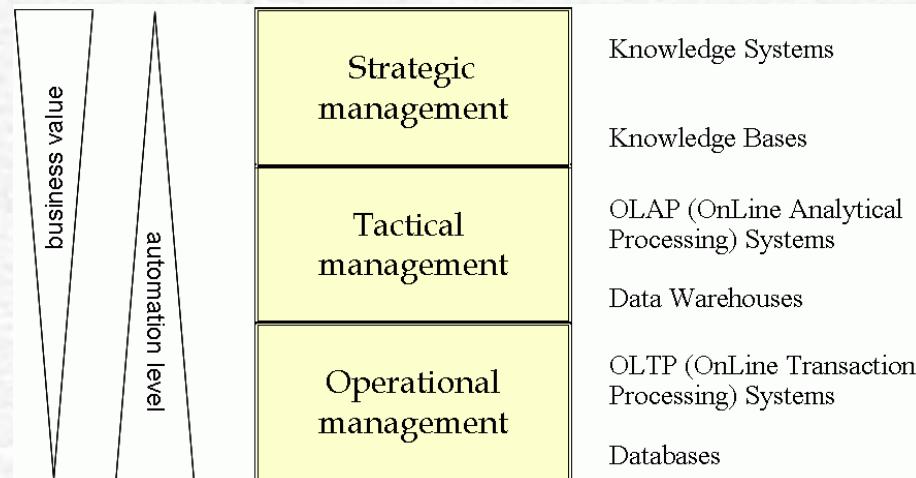


## Sistemul Informatic

- Generează și gestionează informații pentru personal
- Conține: personal, informații, proceduri, software, echipamente hardware și căi de comunicații

# Software-ul face parte și este subordonat afacerii

- Scopul software-ului este acela de a asigura eficiență afacerii
- Legătura dintre nivelele de management și sistemul software într-o companie:



# **Ingineria software este diferită de alte ramuri inginerești tradiționale (1)**

- Software-ul este imaterial
  - Modelele matematice clasice pot fi aplicate doar unor anumite aspecte ale lui
  - Software-ul este definit în termeni fuzzy – “bun”, “rău”, “acceptabil”, “satisfăcător conform cerințelor”, etc. (asemănător cu sectorul serviciilor)
- Ingineria software, în ciuda terminologiei fuzzy, nu trebuie să fie mai puțin riguroasă sau imposibil de demonstrat
- Software trebuie să fie ușor de înțeles, ușor de întreținut și scalabil

## **Ingineria software este diferită de alte ramuri înginerești tradiționale (2)**

- ➊ Diferența principală față de ramurile tradiționale: cerințele se pot modifica mult mai drastic (ex. nu se cere mutarea unui pod cu 10 m mai încolo după ce a fost deja construit)
- ➋ Aplicațiile software nu sunt fabricate ci implementate. Cele destinate afacerilor vor fi mulțate pe situația existentă în realitate
- ➌ Inginerii software trebuie să cunoască în general destul de bine domeniile pentru care implementează aplicații

# **Ingineria software înseamnă mai mult decât programare**

---

- ➊ Ingineria software se aplică problemelor complexe ce nu pot fi rezolvate doar prin programare
    - Sistemele complexe trebuie proiectate înainte de a se trece la programare
  - ➋ Înainte de proiectare trebuie înțelese clar cerințele
  - ➌ Inginerii software trebuie să integreze componentele pe care programatorii le implementează
  - ➍ Ingineria software este o disciplină de echipă
    - Echipele trebuie gestionate
-

# Ingineria software se sprijină pe modelare (1)

- Modelele sunt abstractii ale realitatii
- Prin abstractizare se reduce complexitate, permitand concentrarea asupra celor mai importante aspecte
- Abstractizarea se aplică și asupra produselor și a proceselor software
  - “Software process model” este o reprezentare abstractă a unui proces software
  - “Software product model” este o reprezentare abstractă a unui produs software

## Ingineria software se sprijină pe modelare (2)

- Există modelele pe diferite nivele de-a lungul ciclului de viață a unui produs software:
  - Model pentru **cerințe** – relativ informal
  - Model pentru **specificații** – de obicei formal (UML)
  - Modelul **arhitectural** – arhitectura sistemului
  - Model detaliat de **proiectare** – detaliază soluțiile software/hardware
  - Model de **programare** – model folosit la implementare
- Modelarea este influențată de paradigmile utilizate:
  - **Paradigma funcțională** se bazează pe decompoziție funcțională
  - **Paradigma orientată pe obiecte** împarte sistemul în pachete și componente legate prin diverse relații

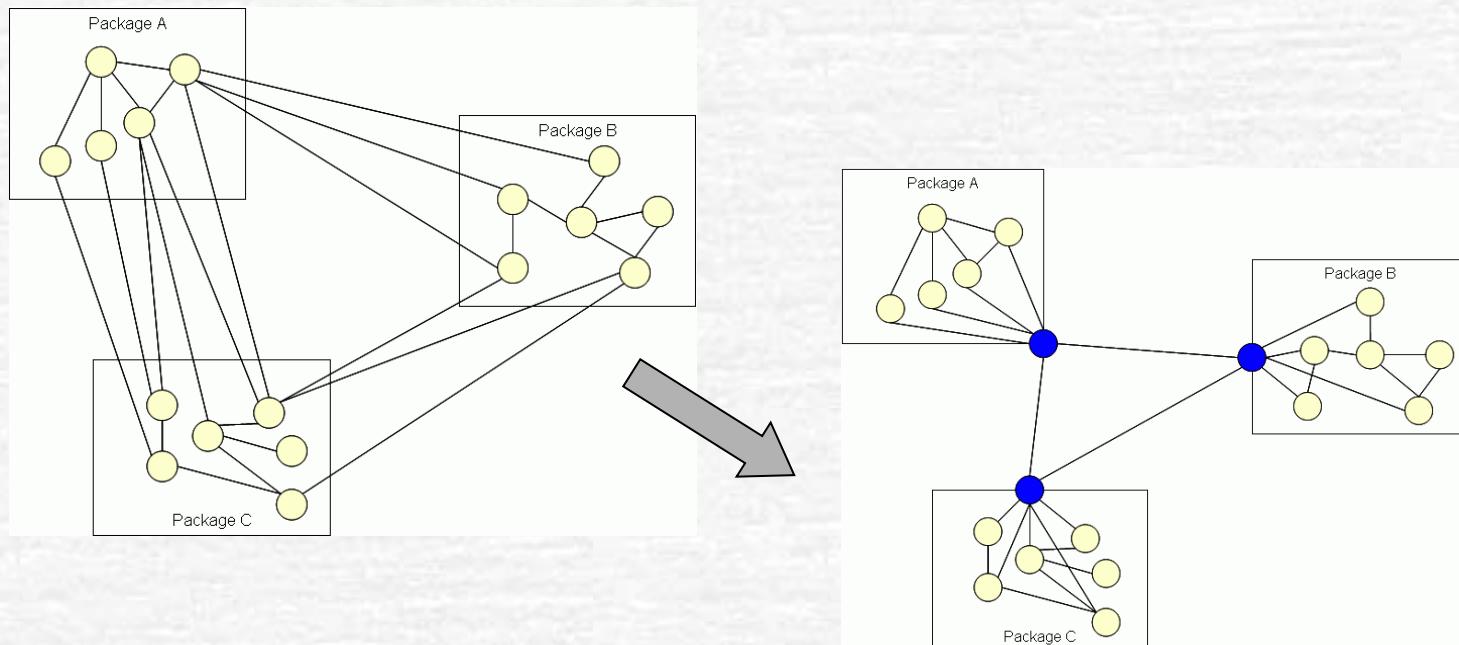
# Sistemele software sunt sisteme complexe (1)

- ◉ Pentru software monolitic: complexitate = dimensiune
- ◉ La sistemele din componente (distribuite) complexitatea stă în legăturile și comunicațiile dintre componente
  - “Cost of glue code is three times cost of application code” (Endres,Rombach, 2003)
- ◉ Măsurare: Cumulative Class Dependency (CCD):

$$CCD = \frac{n!}{2!(n-2)!}$$

## Sistemele software sunt sisteme complexe (2)

- Structurile ierarhice reduc complexitatea (interfețe între pachete – DP Facade)



# **Sistemele software sunt sisteme complexe.**

## **Observații.**

- O structură este stabilă dacă **coeziunea** este **puternică** și **cuplajul slab** (Larry Constantine)
  - Coeziunea – comunicarea în interiorul modulului
  - Cuplajul – interacțiunea între module
- Doar ce este ascuns poate fi schimbat fără riscuri (David Parnas)
- Separarea preocupărilor conduce la arhitecturi standard (Ernst Denert)
- Un sistem care evoluează își mărește complexitatea dacă nu se depune efort pentru menținerea ei sub control (Meir Lehman)

# Concluzii

---

- Domeniul Ingineriei Software este dezvoltarea sistemelor software de mari dimensiuni
  - Sistemul Software este parte a afacerii
  - Natura imaterială și schimbările frecvente sunt factorii care diferențiază Ingineria Software de ramurile inginerești tradiționale
  - Ingineria Software înseamnă mai mult decât programare.
  - Ingineria Software este în mare parte legată de modelare
-



# Fundamente de Inginerie Software

## Cap. 2

**Ciclul de Viață a Unui Sistem Software.**

**Modele Specifice Fazelor Ciclului de Viată.**

**Conf.Dr.Ing. Dan Pescaru**

Textbooks: Maciaszek "Practical Software Engineering", 2005, Cap. 1  
Sursă: <http://www.comp.mq.edu.au/books/pse/>



2022

# Fazele ciclului de viață

➤ ***Fazele ciclului de viață a unui sistem software (Maciaszek'05):***

- **Analiza cerințelor**
- **Proiectarea sistemului**
- **Implementarea**
- **Integrarea și instalarea la beneficiar**
- **Operarea și întreținerea**

# Analiza cerințelor

- Determinarea cerințelor – una din provocările cele mai dificile ale industriei software
- Specificarea cerințelor – Unified Modeling Language (UML)
- Cerințe de documentat:
  - Serviciile oferite de sistem (ce trebuie să facă sistemul)
  - Constrângerile sistemului
- Computer Assisted Software Engineering (CASE)
- Asigurarea calității software-ului
  - Parcurgeri și inspecții

# Proiectarea Sistemului (1)

- ➊ Proiectarea sistemului constă în (Sommerville'04):
  - O descriere a structurii sistemului de implementat
  - Datele care sunt prelucrate în sistem
  - Interfețele între componentele sistemului
  - Algoritmii utilizați (doar în anumite situații)
- ➋ În practică distincția între analiză și proiectare nu este foarte clară
  - Modelele de viață sunt iterative și incrementale
  - Același limbaj de modelare (UML) este utilizat și la analiză

# Proiectarea Sistemului (2)

- Proiectarea arhitecturală (proiectarea de nivel înalt)
- Proiectarea detaliată (adaugă detalii modelului rezultat din analiza cerințelor)
- Problema: gestionarea relaționării partilor aflate în diverse stadii de dezvoltare – cerințe, proiect sau segmente de cod – (Traceability Management)

# Implementarea

- ☞ Implementarea este în mare parte **programare**, dar
  - Proiectul este sub-specificat (în zona algoritmilor)
  - Extra-proiectare înainte de codificare
- ☞ Un programator este un “inginer de componente”
- ☞ Programarea este o “inginerie în circuit”
- ☞ Integrated Development Environments (IDEs)
  - Generare de cod (forward engineered) din model (proiect) +reverse
  - Testare și depanare
- ☞ Revizuirea codului (prin treceri și inspecții)
- ☞ Testare bazată pe execuție (observarea comportamentului)
  - Testarea conformă cu specificațiile (black-box testing)
  - Testarea conformă cu codul (white-box testing) – se urmăresc căi de execuție

# Integrarea și Instalarea

- In pasul de integrare se **asamblează aplicația** din setul de componente implementate și testate în prealabil
  - Dificil de distins față de:
    - Implementare (integrare continuă la “agile development”)
    - Testare (“integration testing”)
  - Condusă de proiectarea arhitecturală a sistemului
- Instalarea reprezintă înmânarea sistemului funcțional beneficiarilor pentru utilizarea în producție
  - Softul este instalat în diverse versiuni
  - Fiecare versiune este precedată de testarea de sistem (dezvoltator – alpha-testing) și testarea de acceptare (beneficiar – beta-testing)
  - Training pentru beneficiar
  - Documentație de utilizare

# Testarea

- **Stub** – o piesă de cod care simulează comportamentul unei componente neimplementate încă
  - Se utilizează în testarea “top-down”
  - Problemă: dependențele circulare. **Big-bang testing** nu este o soluție
- **Driver** – o piesă de cod care conduce integrarea a.î. versiunea increment (build) poate primi datele și contextul care ar fi furnizate de componentele neimplementate încă
  - Se utilizează în testarea “bottom-up”
- **Teste de suport (test harness)** – teste care utilizează “stubs” și “drivers” (utile doar în timpul integrării)

# Operarea și Întreținerea

- ➊ **Operarea** semnifică acea fază a ciclului de viață în care un produs software este utilizat în munca de zi-cu-zi, eventual înlocuind sistemul precedent
- ➋ Startul **Operării** coincide cu începerea procesului de **Mențenanță**
  - Corectivă (“de casă”)
  - Adaptivă
  - Perfectivă
- ⌋ Generează sisteme “moștenite” (**Legacy Systems**)

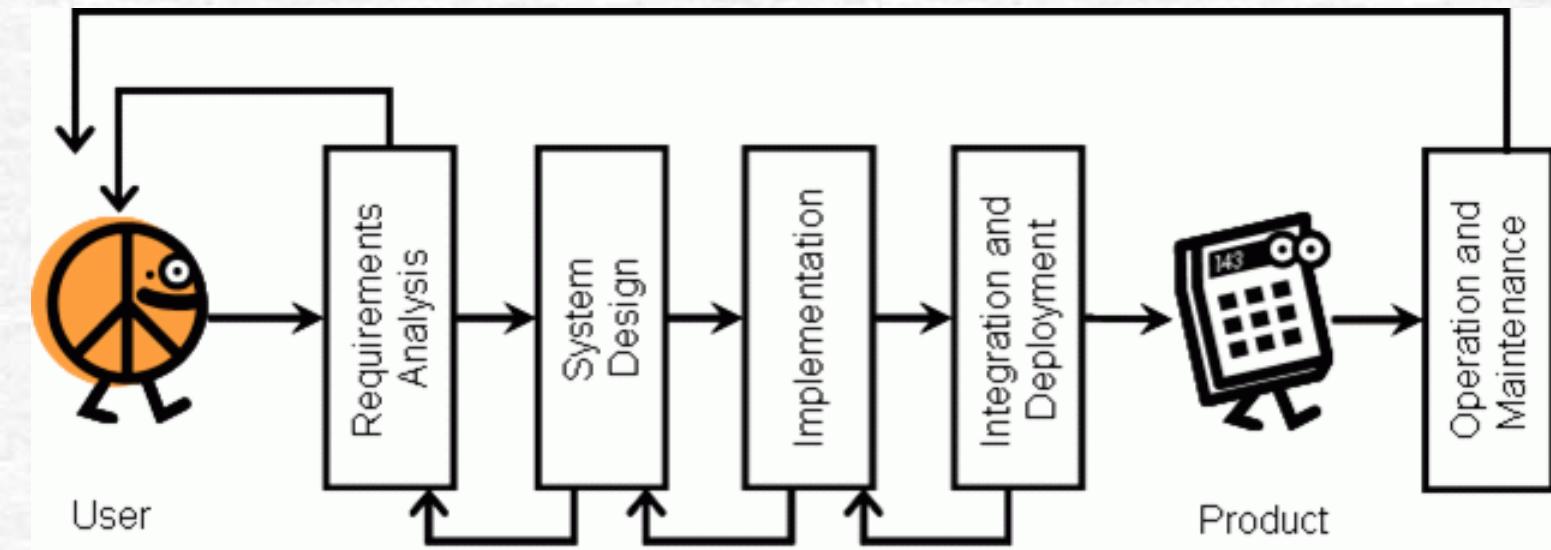
# Variațiile ciclului de viață

➤ Ciclul de viață poate dări funcție de:

- Experiența, abilitățile și cunoștințele membrilor echipei de dezvoltare
- Gradul de cunoaștere și experiența în afacerea vizată de sistem
- Tipul domeniului aplicației
- Schimbările din mediul afacerii
- Schimbările din interiorul afacerii
- Dimensiunea proiectului

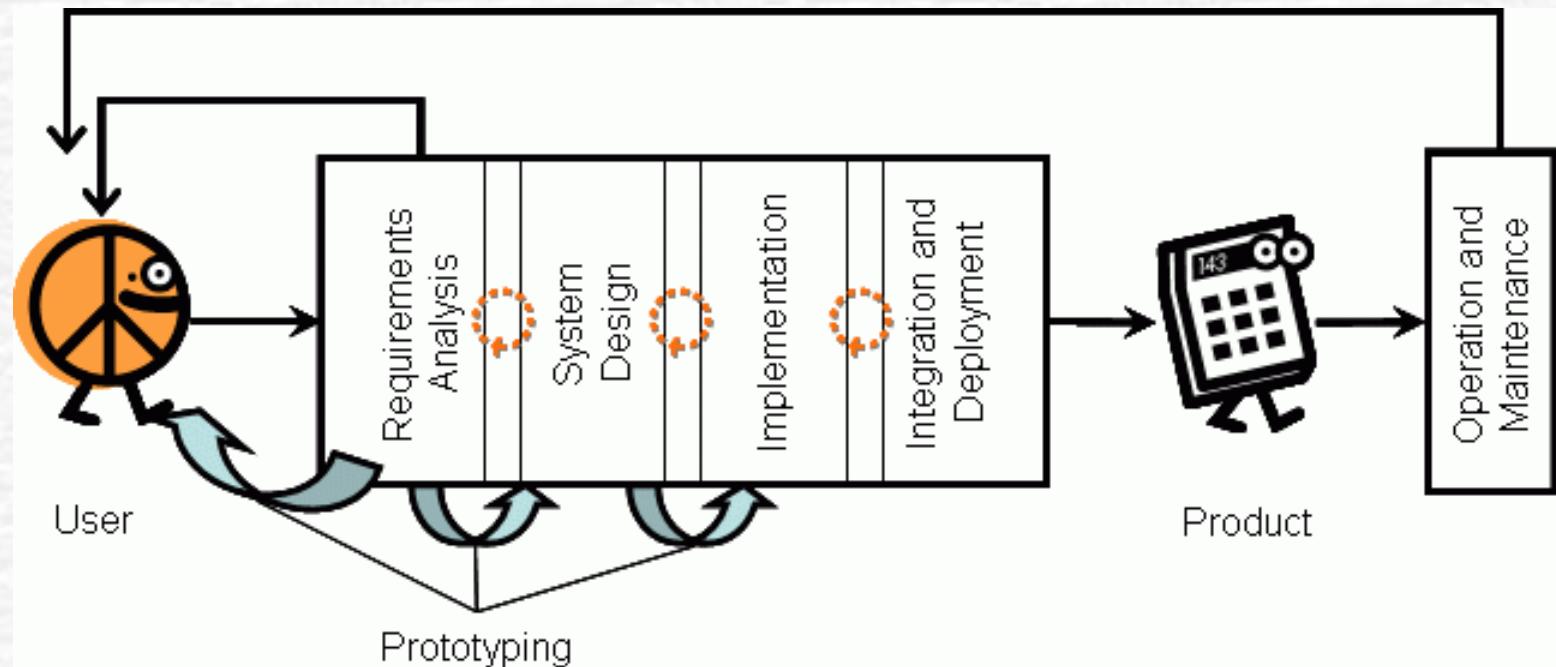
# Modelul în cascadă cu reacție

- Modelul în cascadă cu reacție (Waterfall lifecycle with feedback) (Maciaszek'05)



# Modelul în cascadă cu suprapuneri

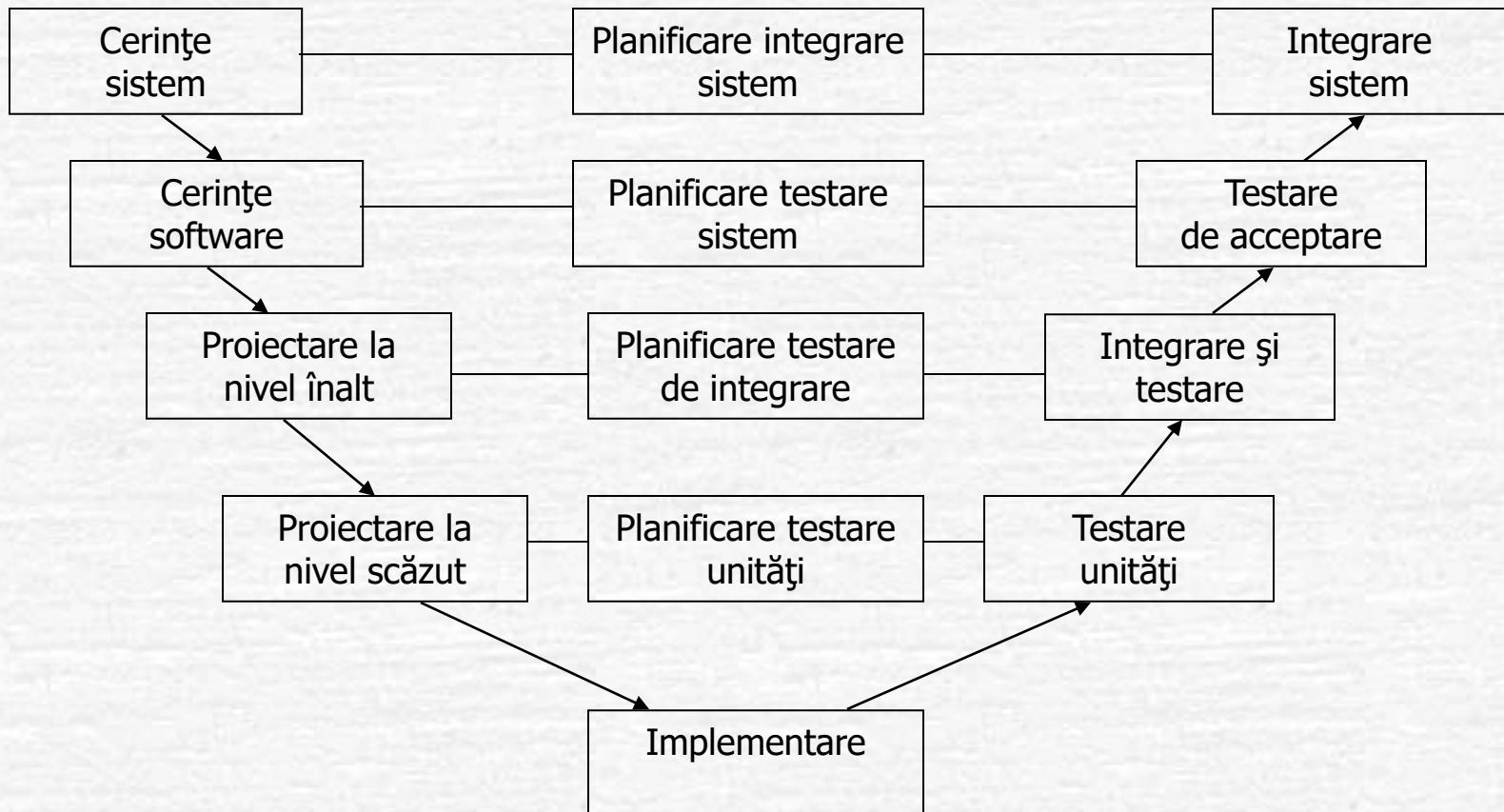
- Modelul în cascadă cu reacție, suprapuneri și prototipuri (Waterfall with feedback, overlaps, and prototypes)



# Modelele în cascadă

- Ambele modele au aproximativ aceleași caracteristici
- Avantaje:
  - Simplu și ușor de utilizat
  - Ușor de gestionat datorită rigidității
  - Fazele și procesele sunt terminate **pe rând** (ușor de urmărit)
  - Bun pentru proiectele mici unde cerințele sunt bine înțelese încă de la început
- Dezavantaje
  - Modificarea cerințelor este foarte greu de gestionat
  - Nu sprijină dezvoltarea orientată pe obiecte
  - Nu se produce prototipuri executabile decât foarte târziu

# Modelul în "V"



# Modelul în “V”

- Modelul în “V” presupune un ciclul de viață **secvențial**
- Planificările se fac o dată cu parcurgerea primei ramuri
- Avantaje:
  - Simplu și ușor de utilizat, mai ales pentru proiecte mici
  - Fiecare fază are lucruri specifice, controlabile, de livrat
  - Mai bun decât modelul în cascadă deoarece planul de teste este făcut încă de la început
- Dezavantaje
  - Rigid, greu de introdus modificări dacă apar
  - Nu produce prototipuri timpurii

# Ciclul de viață iterativ

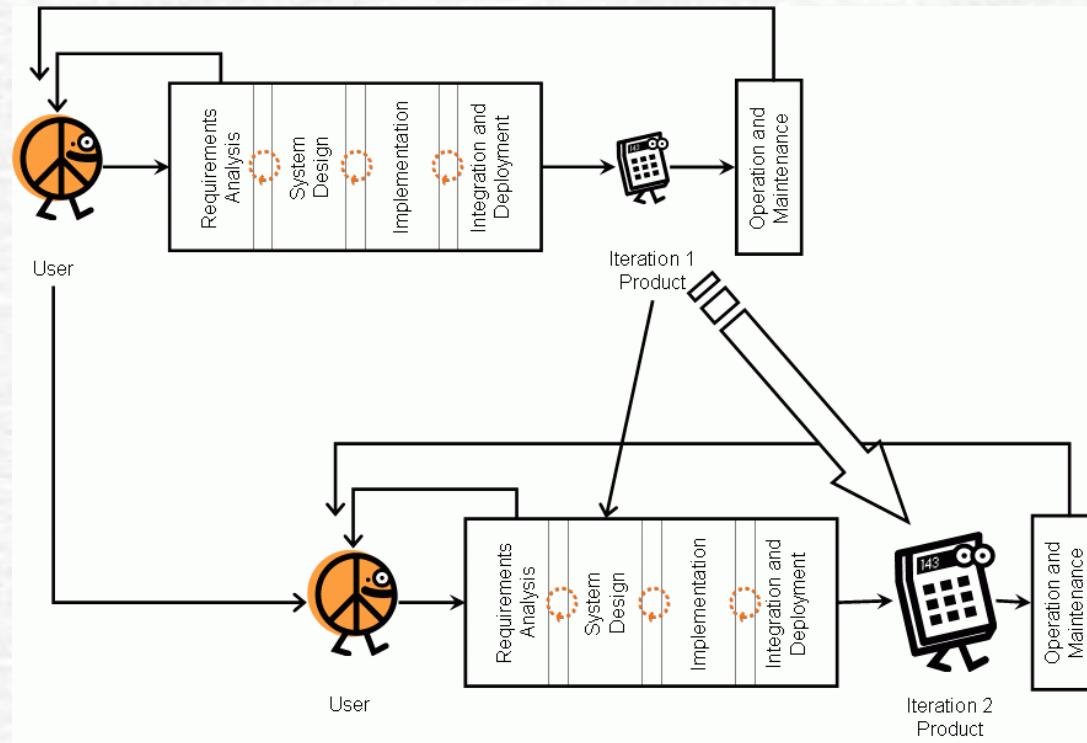
- **Iterația** în inginerie software este o repetiție a unui proces cu scopul de adăuga funcționalități unui produs software
- Ciclul de viață bazat pe iterații presupune creșteri succesive (**increments**) – versiuni îmbunătățite sau extinse ale produsului la sfârșitul fiecărei iterații
- Ciclul de viață iterativ presupune versiuni successive îmbunătățite (**builds**) – sub formă de cod executabil livrabil la încheierea fiecărei iterații
- Dezavantaj: mai greu de gestionat față de secvențial

# Ciclul de viață iterativ. Modele

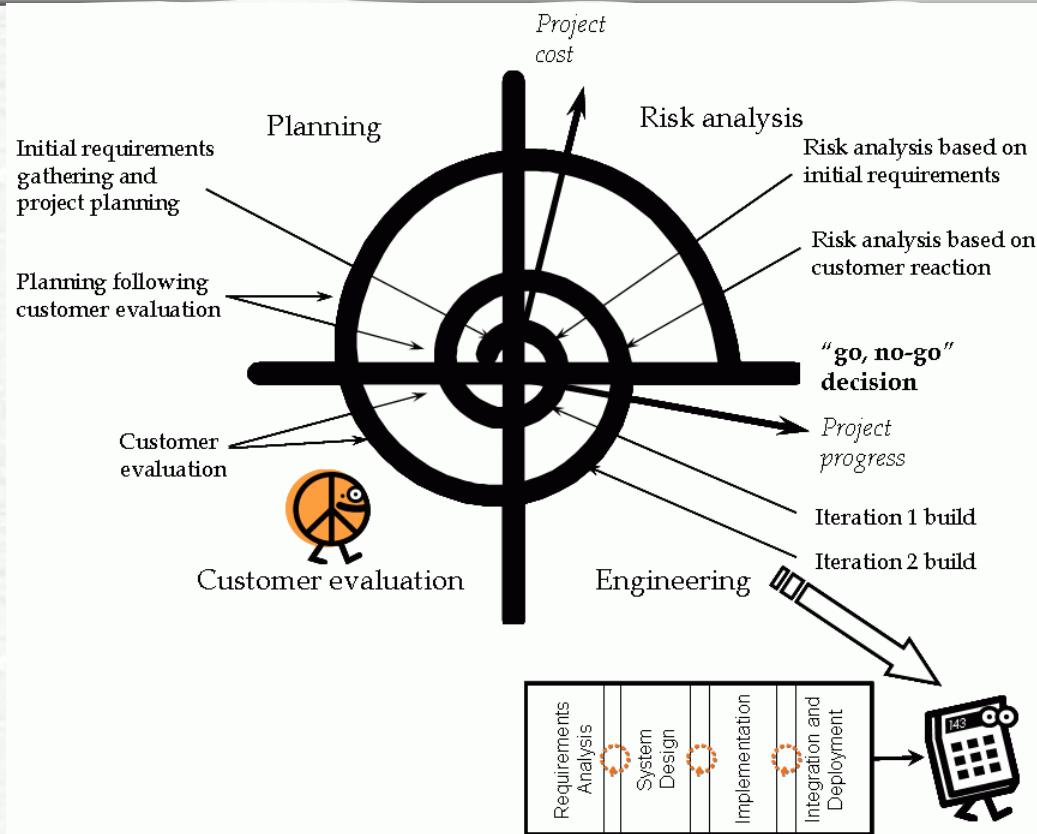
- ☞ Ciclul de viață iterativ presupune **iterații scurte** între variante succesive (zile sau săptămâni, nu luni)
- ☞ Modele principale:
  - Spirală (Boehm, 1988)
  - Agile lifecycle with short cycles (Agile Alliance, 2001)
  - IBM Rational Unified Process (RUP) (IBM, 2003)
  - Model Driven Architecture (MDA) (OMG, 2003)

# Ciclul de viață iterativ. Schemă

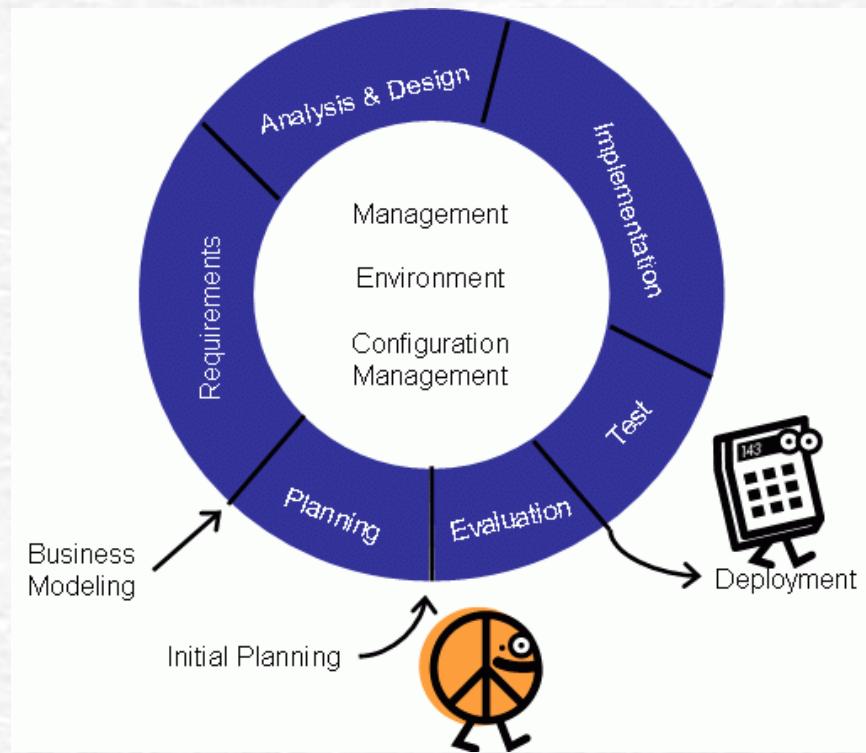
## Ciclul de viață iterativ cu variante successive



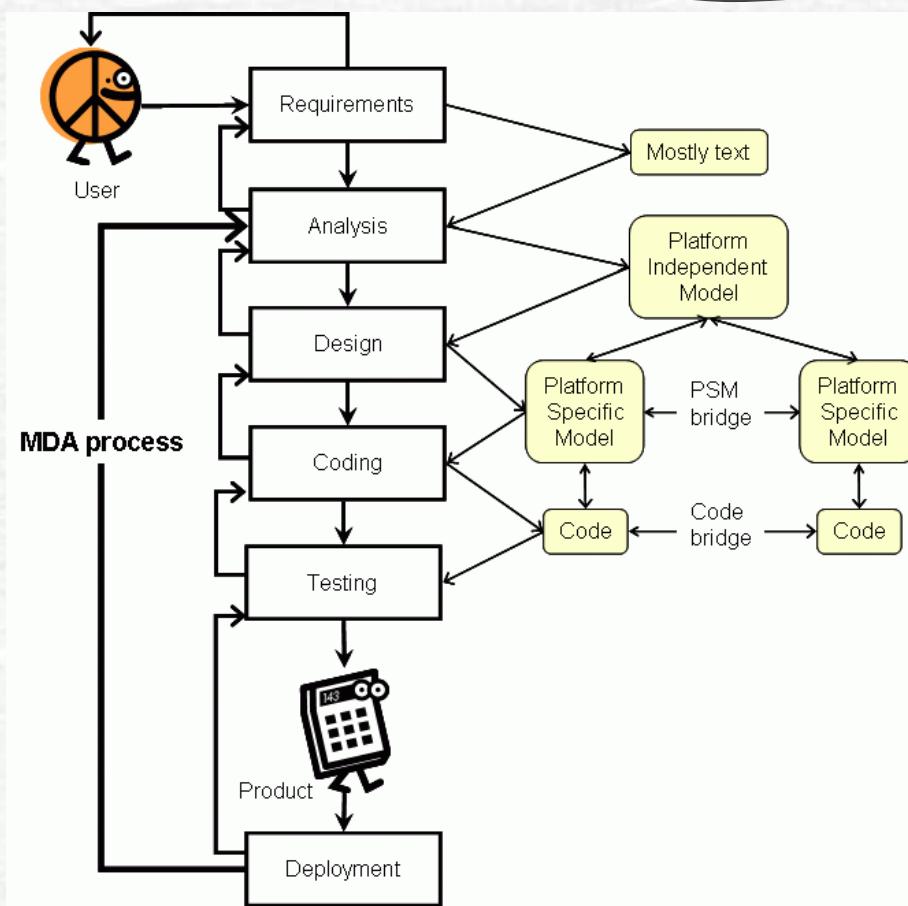
# Modelul Spirală



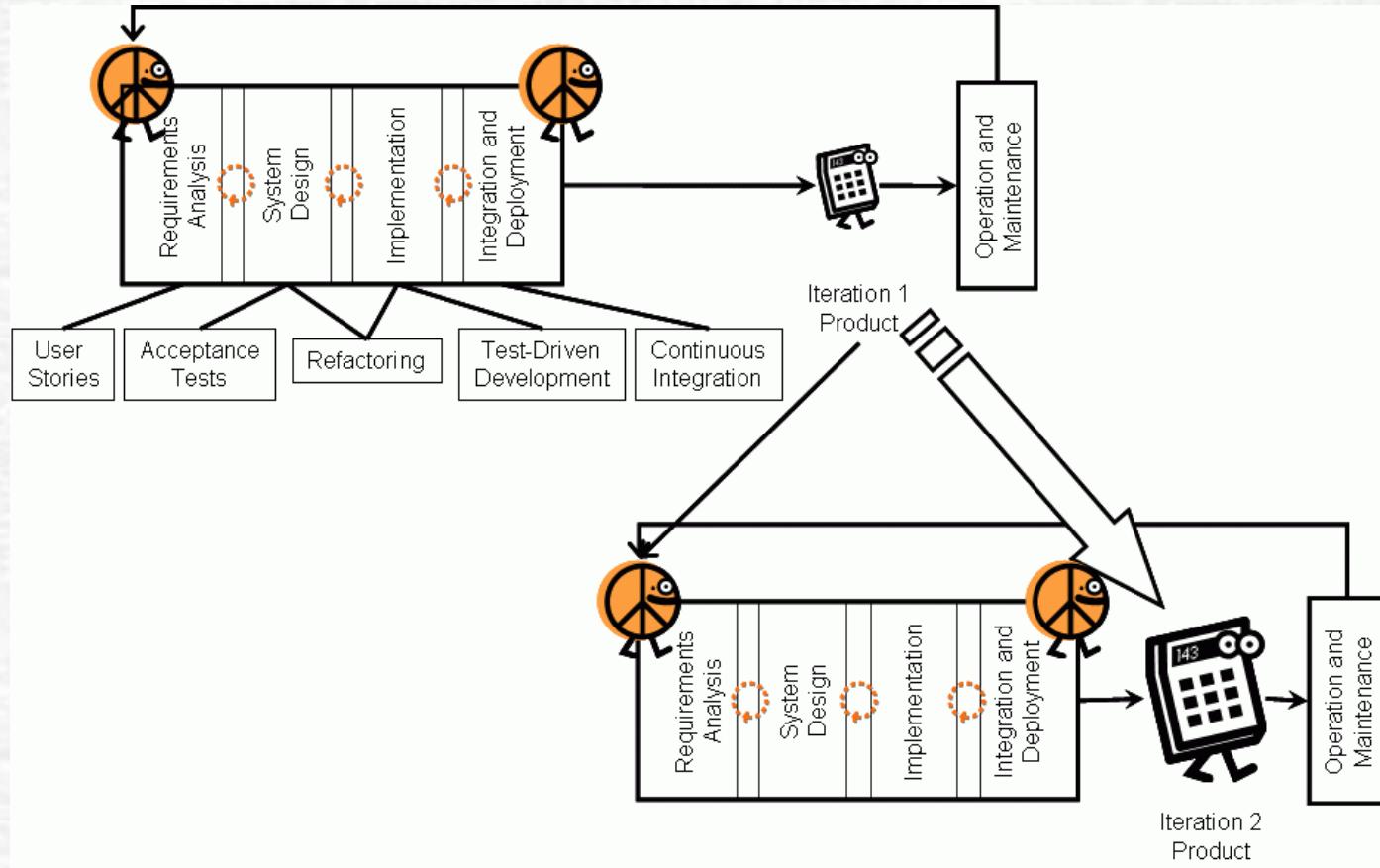
# Modelul RUP



# Modelul MDA



# Modelul Agile



# Practici Extreme Programming (XP)

## Reacție fină

- Programare în perechi
- Planificarea "jocului"
- Dezvoltare condusă de teste
- Echipă completă

## Proces Continuu

- Integrare continuă
- Refactorizare (îmbunătățirea proiectului)
- Variante în pași mici

## Cadru adecvat pentru programator

- Atmosferă relaxată

## Practici comune echipei

- Codare standard
- Drepturi comune asupra codului
- Proiectare simplă
- Denumiri explicite



\*XP Development. Courtesy of wikipedia

# Concluzii

---

- Stadiile procesului de dezvoltare a software-ului sunt cunoscute ca faze ale ciclului de viață software
- Fazele ciclului de viață sunt: analiza cerințelor, proiectarea sistemului, implementarea, integrarea și instalarea, operarea și mențenanța
- Modelele ciclului de viață pot fi împărțite în:
  - Modele cascadă cu reacție
  - Modele iterative cu variante succesive
- Modelele în cascadă nu sunt adecvate proceselor moderne de dezvoltare
- Cele mai reprezentative patru modele iterative sunt: Spirală, Rational Unified Process (RUP), Model Driven Architecture (MDA) și Agile.



# Fundamente de Inginerie Software

## Cap. 3

### Ingineria Cerințelor.

Conf.Dr.Ing. Dan Pescaru

Textbooks: Sommerville "Software Engineering 7", 2004, Cap.6, Cp. 7  
Sursă: <http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/>



2009

# Probleme Specifice

---

## ■ Ingineria cerințelor (**Sommerville**):

- Cerințe funcționale și non-funcționale
- Cerințele utilizatorilor
- Cerințele sistemului
- Specificarea interfeței
- Documentele de specificare a cerințelor

# Ingineria Cerințelor

- Def. (Sommerville'04): Ingineria cerințelor este procesul de stabilire a **serviciilor** cerute sistemului de către clienți precum și a **constrângerilor** sub care acesta va fi dezvoltat și va opera
- Cerințele** sunt descrieri ale serviciilor oferite de sistem și a constrângerilor care sunt generate de-a lungul desfășurării procesului de inginerie a cerințelor

# Cerințe

- ➊ Cerințele pornesc de la afirmații abstracte de nivel înalt până la specificații matematice funcționale detaliate
- ➋ Cerințele îndeplinesc mai multe funcții:
  - Pot constitui baza negocierilor pentru un contract – ca atare trebuie să fie deschise diverselor interpretări
  - Pot constitui baza contractului în sine – ca atare trebuie definite cât mai detaliat
  - Ambele afirmații trebuie acoperite de aceeași cerințe

# Tipuri de Cerințe

## Cerințe utilizator

- Afirmații în limbaj natural și diagrame a serviciilor oferite de sistem îaloală cu constrângerile operaționale.
- Scrise pentru clienți.

## Cerințele sistemului

- Un document structurat stabilind descrierea detaliată a funcțiilor sistemului, serviciile oferite și constrângerile operaționale.
- Poate fi parte a contractului cu clientul.

# Exemplu

## Cerință utilizator

- Sistemul trebuie să asigure accesarea fișierelor externe create de alte programe

## Cerințe sistem corespunzătoare

- Trebuie prevăzută de definire a tipurilor de fișiere externe
- Fiecare tip de fișier extern trebuie să aibă asociat o aplicație care să fie apelată pentru fișierul respectiv
- Fiecare tip de fișier extern trebuie să fie reprezentat în fereastra utilizatorului printr-o icoană specifică
- La selectarea icoanei se va apela aplicația asociată tipului respectiv de fișier extern

# Utilizarea Cerințelor

☞ **Cerințele utilizator** se adresează:

- Managerilor clientului
- Utilizatorilor finali
- Inginerilor clientului
- Managerilor de contracte
- Proiectanților de sistem

☞ **Cerințele de sistem** se adresează:

- Utilizatorilor finali
- Inginerilor clientului
- Proiectanților de sistem
- Programatorilor

# Cerințe Funcționale și Non-funcționale

---

## ■ Cerințe funcționale

- Afirmații despre servicii pe care sistemul trebuie să le conțină, cum trebuie el să răspundă la anumite intrări și cum reacționeze în anumite situații

## ■ Cerințe non-funcționale

- Constrângeri ale serviciilor și funcțiilor oferite de sistem cum ar fi constrângeri de timp, constrângeri ale procesului de dezvoltare, standarde, etc.

## ■ Cerințe ale domeniului

- Cerințe impuse de domeniu aplicației care reflectă caracteristicile acestuia

# Cerințe Funcționale

---

- Descrie funcționalitatea sistemului și serviciile oferite
- Depind de tipul softului, de utilizatorii avuți în vedere și de tipul sistemului pe care softul este utilizat
- Cerințele funcționale ale utilizatorilor pot fi descrieri de ansamblu dar cerințele funcționale ale sistemului trebuie să descrie în detaliu serviciile oferite

# Calitatea Cerințelor

---

- ☞ Trebuie evitate cerințele ambigue
- ☞ Cerințele trebuie să fie în același timp și complete și consistente
  - Complete
    - Trebuie să includă descrieri pentru toate facilitățile oferite
  - Consistente
    - Trebuie să nu existe conflicte și contradicții
- ☞ În practică este greu de obținut

# Cerințe Non-funcționale

- ➊ Definesc proprietăți și constrângeri ale sistemului.
  - Ex: fiabilitatea, timpul de răspuns, cerințele pentru spațiul de stocare, cerințe ale sistemului de intrări-ieșiri etc.
- ➋ La întocmirea lor se va ține cont de un anumit program CASE, limbaj de programare sau metodă de dezvoltare
- ➌ Cerințele non-funcționale pot fi mai critice decât cele funcționale. Dacă nu sunt îndeplinite, sistemul nu va fi util scopului în care a fost dezvoltat

# Tipuri de Cerințe Non-funcționale (I)

## ➤ Cerințe ale **produsului**

- Cerințe care specifică un anumit comportament al produsului (ex: viteză de execuție, fiabilitatea etc.)

## ➤ Cerințe legate de **organizare**

- Cerințe care sunt consecințe ale politicilor de organizare a producției software (ex: standarde utilizate, cerințe de implementare etc.)

## ➤ Cerințe **externe**

- Cerințe asociate unor factori externi (ex. cerințe de interoperabilitate, cerințe legislative etc.)

# Tipuri de Cerințe Non-funcționale (II)

## Cerințe ale produsului

- Cerințe legate de gradul de utilitate
- Cerințe de eficiență (performanță/spațiu)
- Cerințe de fiabilitate
- Cerințe de portabilitate

## Cerințe legate de organizare

- Cerințe de livrare
- Cerințe de implementare
- Cerințe legate de standarde

## Cerințe externe

- Cerințe de interoperabilitate
- Cerințe legate de etică
- Cerințe legislative

# Cuantificarea Cerințelor

Proprietate	Exprimare
Viteză	Tranzacții/sec. Timp de răspuns la utilizator Viteză de reîmprospătare a ecranului
Dimensiune	Fizică – MB Număr de cipuri ROM
Fiabilitate	Timpul mediu dintre două defecte Rata de apariție a defectărilor
Robustete	Probabilitatea de corupere a datelor la eroare Timpul de restart după apariția defectării
Portabilitatea	Procentul de linii de cod dependente de ținta implementării Numărul de sisteme țintă

# Cerințe de Domeniu

---

- Derivate din domeniul aplicației
- Descriu caracteristici și facilități legate de domeniu
- Pot fi cerințe funcționale noi, constrângeri sau cerințe deja existente
- Dacă nu sunt îndeplinite sistemul va fi nefuncțional

# Probleme cu Cerințele de Domeniu

---

## Gradul de înțelegere a cerințelor

- Cerințele sunt exprimate în limbajul specific domeniului aplicației
- Aceasta deseori nu este înțeles de inginerii software care dezvoltă sistemul

## Cerințe lapidare

- Specialiștii dintr-un domeniu sunt foarte familiari cu acesta și au tendința de a nu explica pe înțelesul programatorilor cerințele pe care le enunță

# Cerințe ale Utilizatorilor

- Cerințele utilizator trebuie să descrie cerințe funcționale și non-funcționale într-o manieră în care sunt pe înțelesul utilizatorilor sistemului care nu dețin cunoștințe tehnice detaliate
- Cerințele utilizator sunt definite utilizând limbajul natural, tabele și diagrame care pot fi înțelese de toți utilizatorii

# Probleme la Utilizarea Limbajului Natural

---

## Lipsa clarității

- Obținerea unei bune precizii este dificilă dacă se dorește ca documentul să fie ușor de citit

## Confuzii privind cerințele

- Cerințele funcționale și non-funcționale tind să fie amestecate

## Amagarea cerințelor

- Diferite cerințe pot fi exprimate împreună

# Sfaturi Utile la Scrierea Cerințelor

- Creați un format standard și utilizați-l la toate cerințele
- Utilizați limbajul într-un format consistent. Folosiți trebuie pentru cerințele obligatorii și ar trebui pentru cele de dorit să apară
- Scoateți în evidență textul care identifică părțile importante ale cerințelor
- Evitați folosirea jargonului la specificarea cerințelor

# Cerințele Specifice Sistemului

- ➊ Cerințele sistemului sunt specificate mai detaliat decât cerințele utilizator
- ➋ Scopul principal al lor este acela de a fi baza proiectării sistemului
- ➌ Ele pot fi incorporate în contact

# Utilizarea Cerințelor În Proiectare

---

- ➊ În principiu, cerințele trebuie să exprime ce poate face sistemul, iar proiectul trebuie să exprime cum poate face aceste lucruri
  
- ➋ În practică cerințele și proiectul sunt inseparabile
  - Arhitectura sistemului se proiectează pe baza cerințelor
  - Sistemul poate inter-opera cu alte sisteme care generează la rândul lor noi cerințe
  - Utilizarea unui anumit tip de proiectare poate fi o cerință de domeniu

# Alternative de Reprezentare a Cerințelor

---

- ➊ Utilizarea unui format standard sau a machetelor în conjuncție cu limbajul natural
- ➋ Utilizarea unui limbaj de proiectare – asemănător unui limbaj de programare dar mai abstract
- ➌ Utilizarea unui limbaj grafic suplimentat cu adnotări textuale (mai ales pentru cerințe sistem)
- ➍ Utilizarea unor specificații matematice (ex. mașini cu stări finite). Elimină ambiguitățile dar este dificil de înțeles.

# Documentarea Cerințelor

- Documentul de specificare a cerințelor este actul oficial care specifică ce se așteaptă de la dezvoltatori
- Trebuie să includă atât definirea cerințelor utilizator cât și specificarea cerințelor sistem
- Nu este un document de proiectare. Pe cât posibil el trebuie să specifice CE trebuie să facă sistemul și nu CUM să facă aceste lucruri

# Utilizatorii Documentației Cerințelor

## Clienții

- Impun cerințele și verifică apoi dacă acestea sunt conforme cu nevoile lor. Pot cere și modificări ale cerințelor

## Managerii

- Utilizează documentul pentru a stabili termenii contractului și a planifica procesul de producție

## Inginerii de sistem

- Utilizează documentul pentru a înțelege ce trebuie dezvoltat

## Inginerii de la testare

- Utilizează documentul pentru a proiecta testele de validare

# Standardul IEEE pentru Specificarea Cerințelor

- Definește o structură generică pentru documentarea cerințelor:
  - Introducere
  - Descrierea generală
  - Cerințe Specifice
  - Anexe
  - Index

# Structura Documentului de Specificare a Cerințelor

- Prefață
- Introducere
- Glosar de termeni
- Definirea cerințelor utilizatorilor
- Arhitectura sistemului
- Specificarea cerințelor de sistem
- Modelarea sistemului
- Evoluția sistemului
- Anexe
- Index

# Concluzii

---

- Cerințele specifică ce trebuie să știe să facă sistemul și definesc constrângeri pentru operare și implementare
- Cerințele funcționale stabilesc serviciile pe care sistemul trebuie să le asigure
- Cerințele non-funcționale stabilesc constrângeri ale sistemului și ale procesului de dezvoltare
- Cerințele utilizator sunt specificații de nivel înalt referitoare la ce trebuie să facă sistemul. Ele sunt scrise în limbaj natural plus table și diagrame

# Concluzii

---

- ➊ Cerințele sistem specifică funcțiile pe care sistemul trebuie să le ofere
- ➋ Documentul de specificare a cerințelor software este rezultatul acordului între client și dezvoltator asupra cerințelor sistemului
- ➌ Standardul IEEE este un punct de start util pentru definirea unui standard mai detaliat pentru specificarea cerințelor



# Fundamente de Inginerie Software

## Cap. 4

### Limbaje de Modelare Software.

Conf.Dr.Ing. Dan Pescaru

Textbooks: Maciaszek "Practical Software Engineering", 2005, Cap. 2  
Sursă: <http://www.comp.mq.edu.au/books/pse/>



2021

# Limbaje de modelare

---

- Ingineria software se sprijină pe modelare:
    - Produsul final – programul – este un model executabil
  - Modelarea necesită un limbaj adecvat pentru:
    - Comunicarea între parteneri
    - Exprimarea proceselor de dezvoltare și a artefactelor pe multiple nivele de abstractizare
  - Limbajul UML (Unified Modeling Language)
  - Profile UML pentru diverse domenii și tehnologii
  - Un model constă în una sau mai multe diagrame precum și în informațiile adiționale stocate în cadrul proiectului
-

# Limbaje de modelare structurate

---

## Programarea structurată

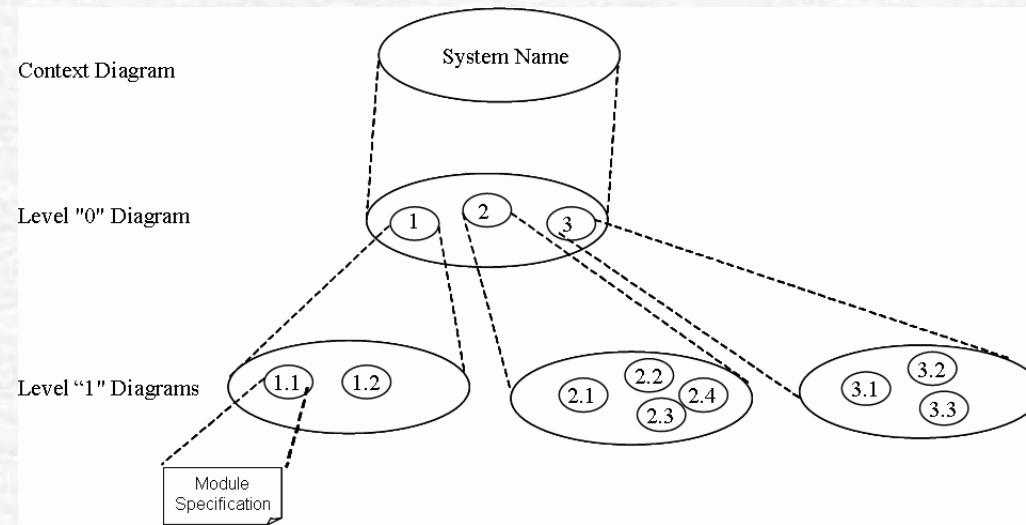
- Fără instrucțiuni *go to*
- Structuri de control condiționate (*if, switch*) și ciclice (*for, while*)
- Stil de proiectare top-down a programelor

## Programare structurată → modelare structurată (analiză și proiectare structurată)

- Exprimă caracterul monolitic și procedural al sistemelor stil Cobol
- Decompoziția funcțională – dezvoltare top-down orientată funcțional
- Tehnici de vizualizare
  - Diagrame de flux de date (Data Flow Diagrams DFD)
  - Diagrame Entitate-Relație (Entity-Relationship Diagrams ERD)
  - Diagrame structurale (Structure Charts SC)

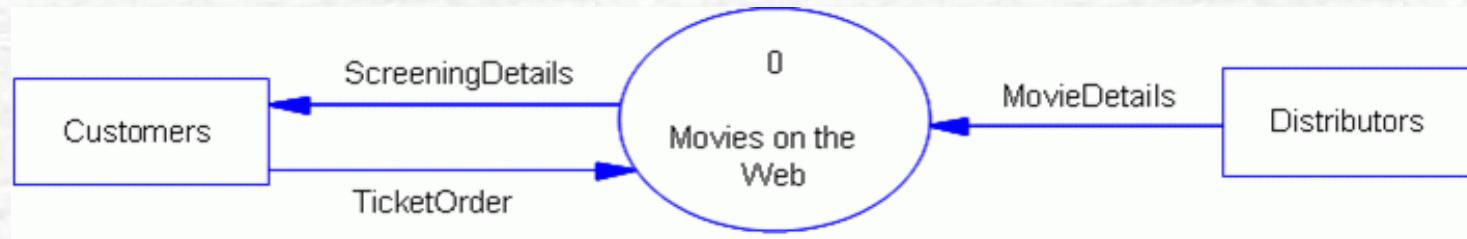
# Diagrame de flux de date

- Una dintre tehniciile cele mai răspândite de modelare din istoria ingineriei software
- Astăzi este pusă în umbră de modelele orientate pe obiecte
- Se bazează pe **decompoziția funcțională**



# Diagrama de context

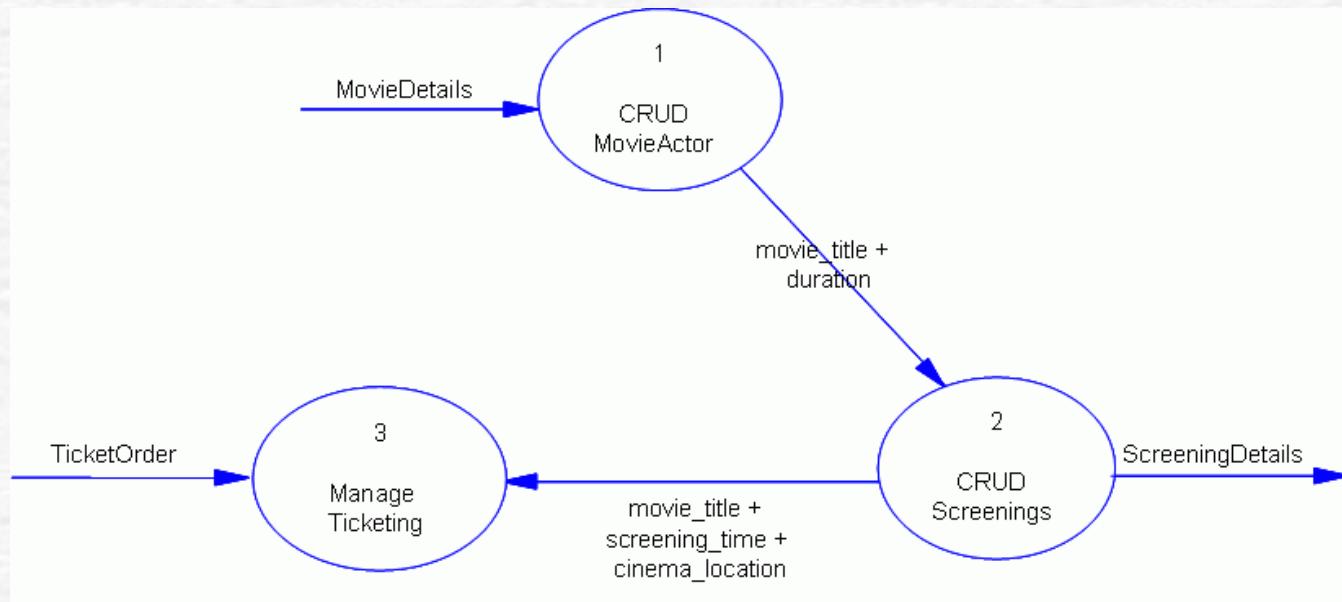
- Constă în:
  - Un singur proces
  - Un număr de entități externe
  - Fluxuri de intrare-ieșire între proces și entitățile externe
- Denotă locul sistemului în mediul în care va evoluă, stabilind granițele cu acesta



- Ex: un sistem de reclamă pe Web a unui lanț de cinematografe (Maciaszek, 2005)

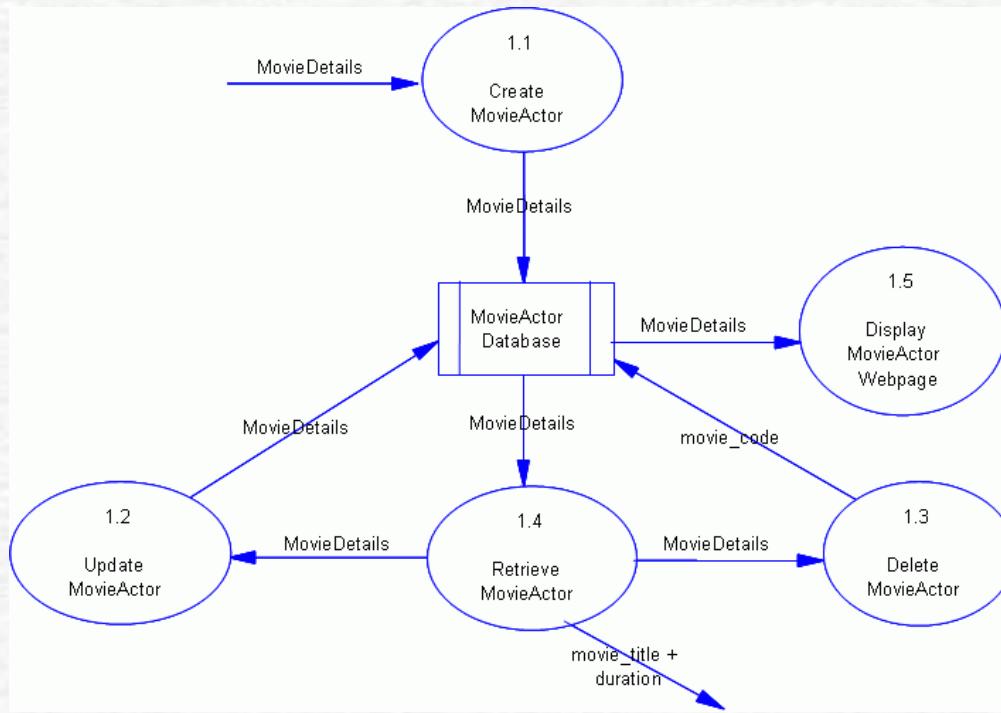
# Diagrama de nivel “0”

- Denumită și diagramă de ansamblu
- (CRUD = Create, Read, Update and Delete)



# Diagrama de nivel “1”

- Balansarea fluxurilor (între procese), module de stocare a datelor
- Decompoziție a proceselor de pe nivelul “0” (ex. CRUD MovieActor)



# Modele Entitate-Relație (ER)

- O tehnică de modelare a datelor
- Diagramele ER pot să conțină doar trei elemente de modelare: entități, relații și attribute
- O **Entitate** este o structură de date conceptuală reprezentând un fapt sau regulă distinct identificabilă în problemă
- O **Relație** reprezintă o asociere între instanțele entităților
- Un **Atribut** este o pereche data-valoare de tip:
  - Valoare singulară
  - Valoare multiplă (atribut compus) – în general de evitat

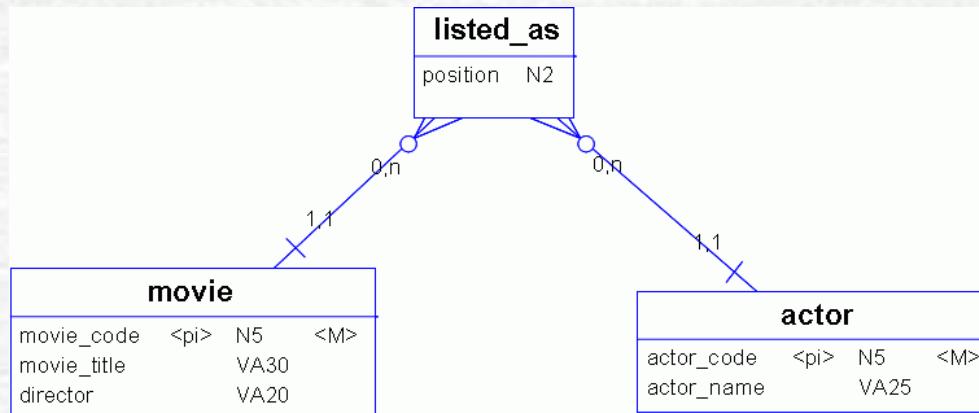
# ER “Crow’s foot notation”

## Attributele apar în cadrul entităților

- Nume, tip, dacă este cheie (pi – primary identifier), dacă trebuie să aibă o valoare (m – mandatory)

## Multiplicitatea relațiilor

- Participare în relație optională sau obligatorie



# Limbaje de Modelare Orientate pe Obiecte

---

- ☞ **Unified Modeling Language (UML)** “*...is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.*” (*UML, 2003b, p.1-1*)
- ☞ În UML modelarea vizuală constă în principal în **aranjarea** aşa numiților **clasificatori**
  - Un **Clasificator** este un element de modelare care descrie comportamentul sau structura sistemului printr-o reprezentare vizuală
    - Exemple de clasificatori: **class, actor, use case, relationship**
- ☞ La rularea sistemului, clasificatorilor le corespund obiectele
  - Un **Obiect** este o bucată de cod care are:
    - **Stare** – definită de valorile atributelor
    - **Comportament** – definit de operațiile care le poate executa
    - **Identitate** – care îl diferențiază de alte obiecte (chiar cu aceeași stare și comportament)

# Tipuri de Diagrame UML

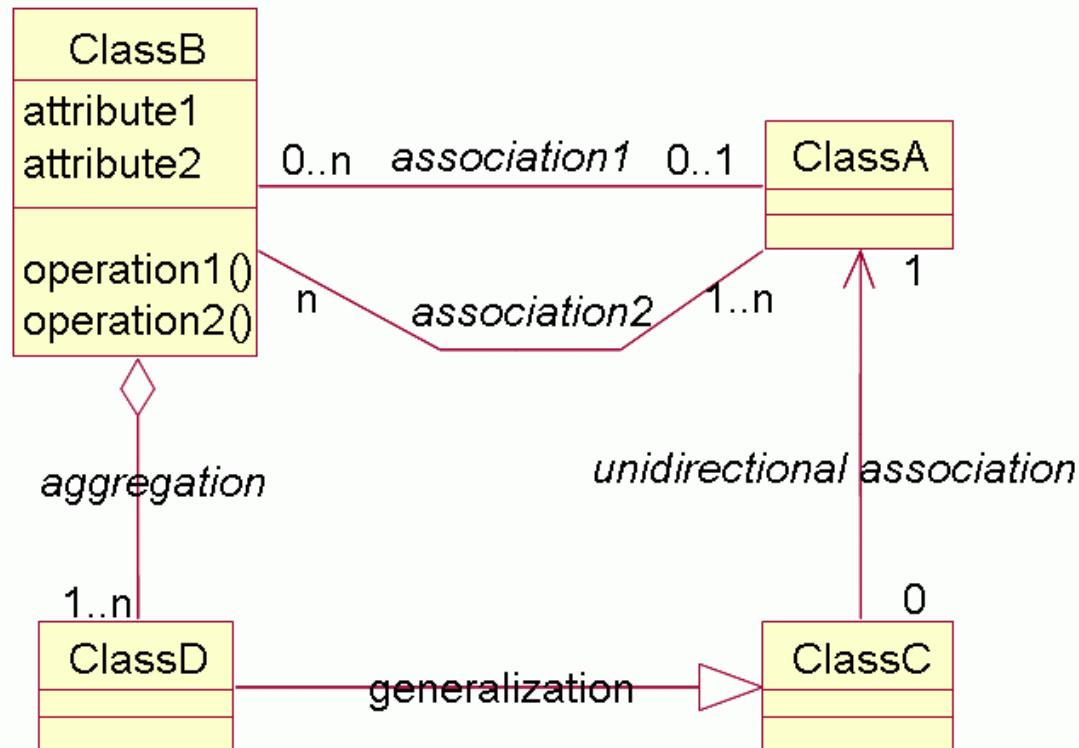
---

- ➊ Class Diagram (state structure)
- ➋ Use Case Diagram
- ➌ Interaction Diagram
- ➍ Statechart Diagram
- ➎ Activity Diagram
- ➏ Implementation Diagram

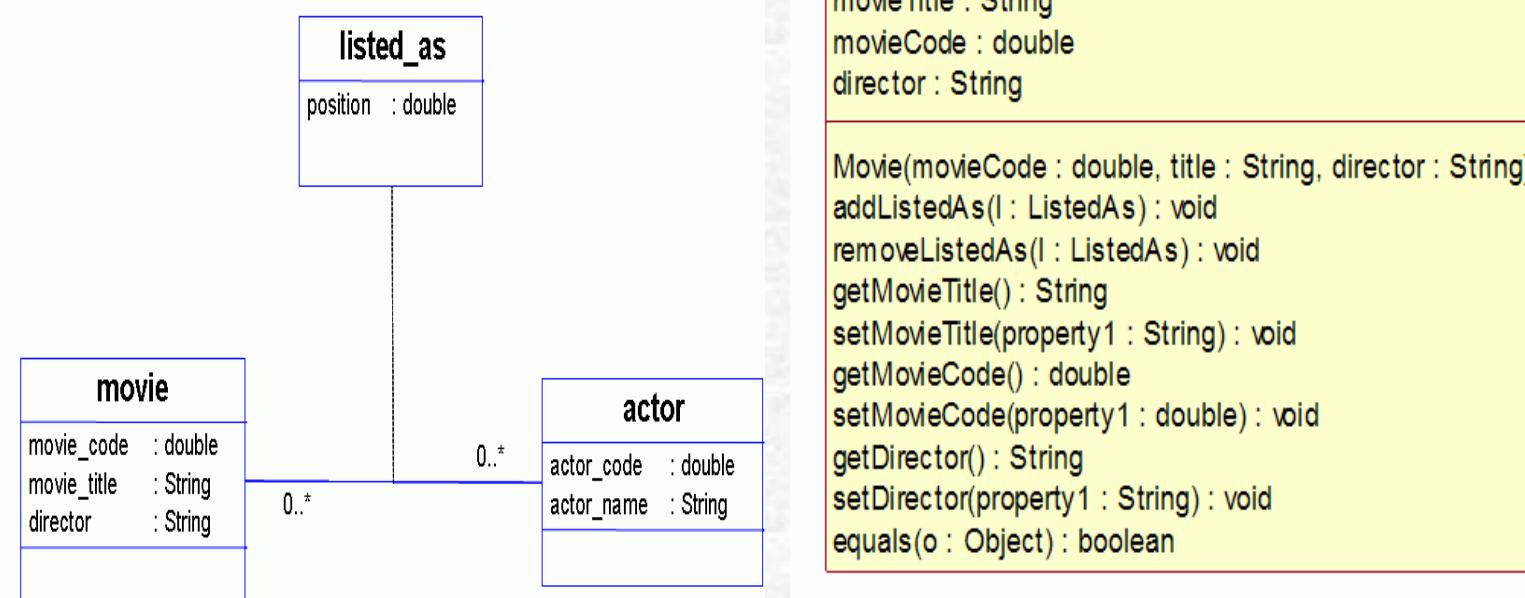
# UML Class Diagram

- ☞ Class diagram (diagramă de clase)
  - Exprimă în mod static **structura modelelor** (numită și **model de stare**)
  - Vizualizează **clase (și interfețe)**, structura lor internă și relațiile lor cu alte clase (interfețe)
- ☞ “**a Class is the descriptor for a set of objects with similar structure, behavior, and relationships**” (UML, 2003, p.3-35)
- ☞ **Atributul** este o caracteristică (tipizată) structurală a unei clase
  - Poate fi: data membru, variabilă membru, variabilă de instanță sau câmp
- ☞ **Operația** este o caracteristică comportamentală a unei clase
  - Poate fi: funcție membră sau metodă

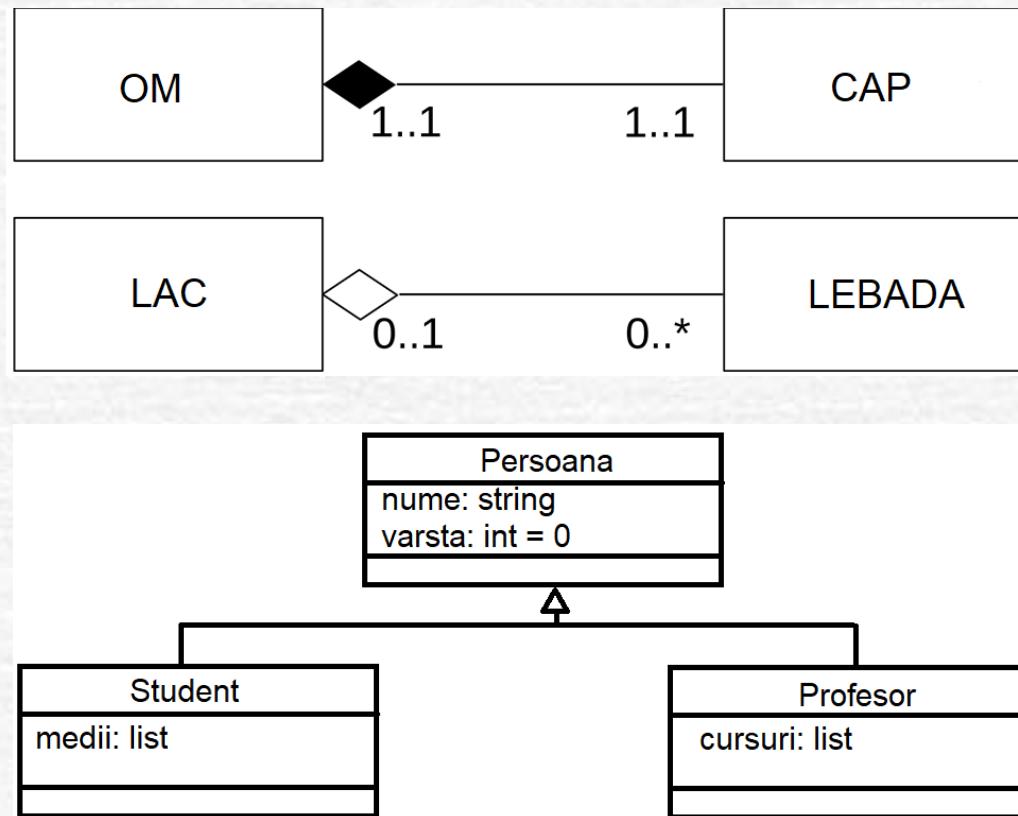
# Elemente de Modelare la o Diagramă de Clase



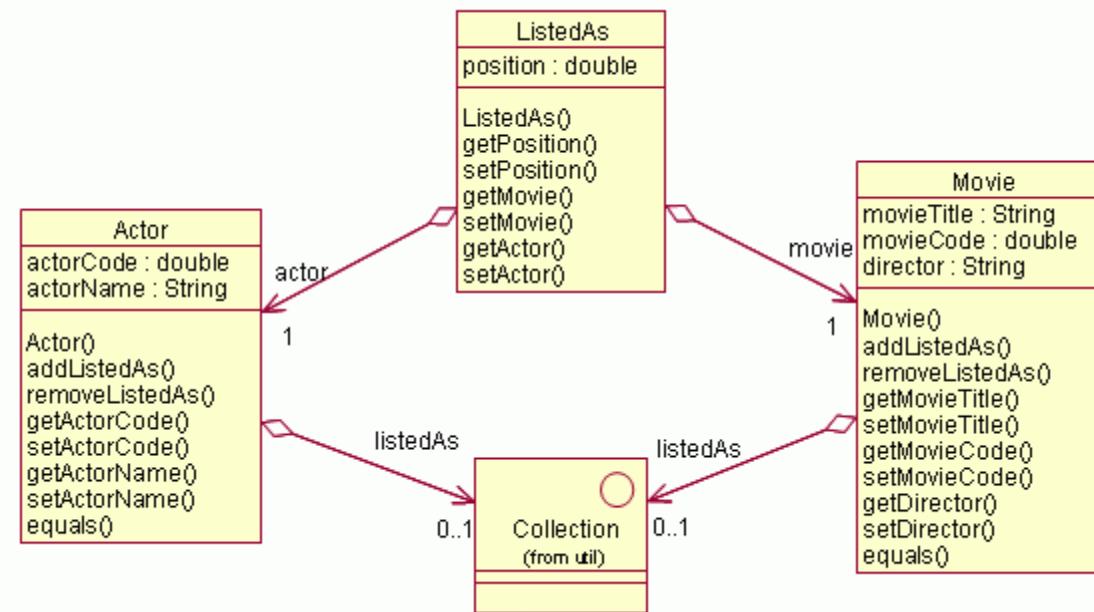
# Diagrama de Clase ca o Diagramă de Structură Statică



# Caracteristici de Stare și Comportament



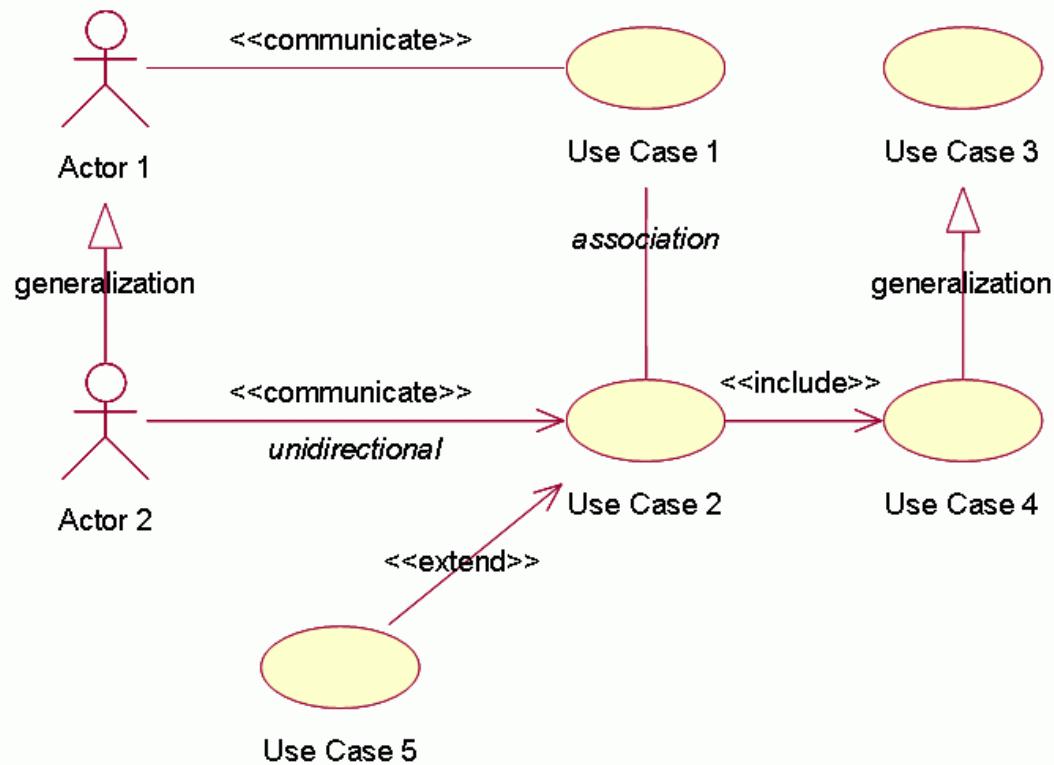
# Proiectarea Diagramelor de Clase



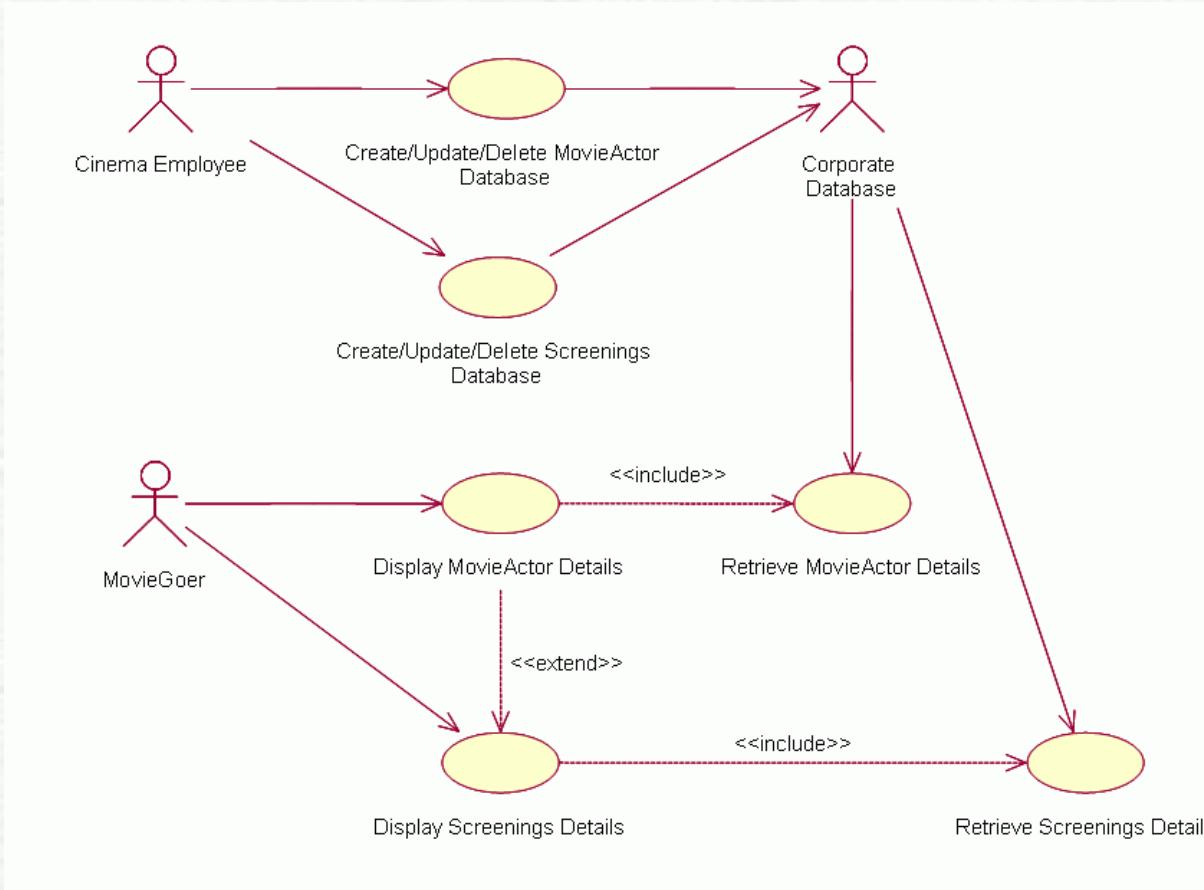
# Diagrame Use Case

- Principala tehnică UML de modelare la nivel de analiză a comportamentului
- Expresivitatea acestor diagrame nu rezidă neapărat în reprezentarea grafică. Puterea reală a acestor diagrame stă mai ales în specificațiile textuale asociate
- Diagramele **Use Case** (cazuri de utilizare) reprezintă o piesă importantă în definirea funcționalității sistemului
- Un Actor este un rol pe care cineva sau ceva îl joacă relativ la un Use Case.
  - Actorul comunică cu un caz de utilizare (prin relația de «**communicate**») și așteptă o **reacție** de la acesta – o **valoare sau un alt rezultat observabil**

# Elemente de Modelare Use Case



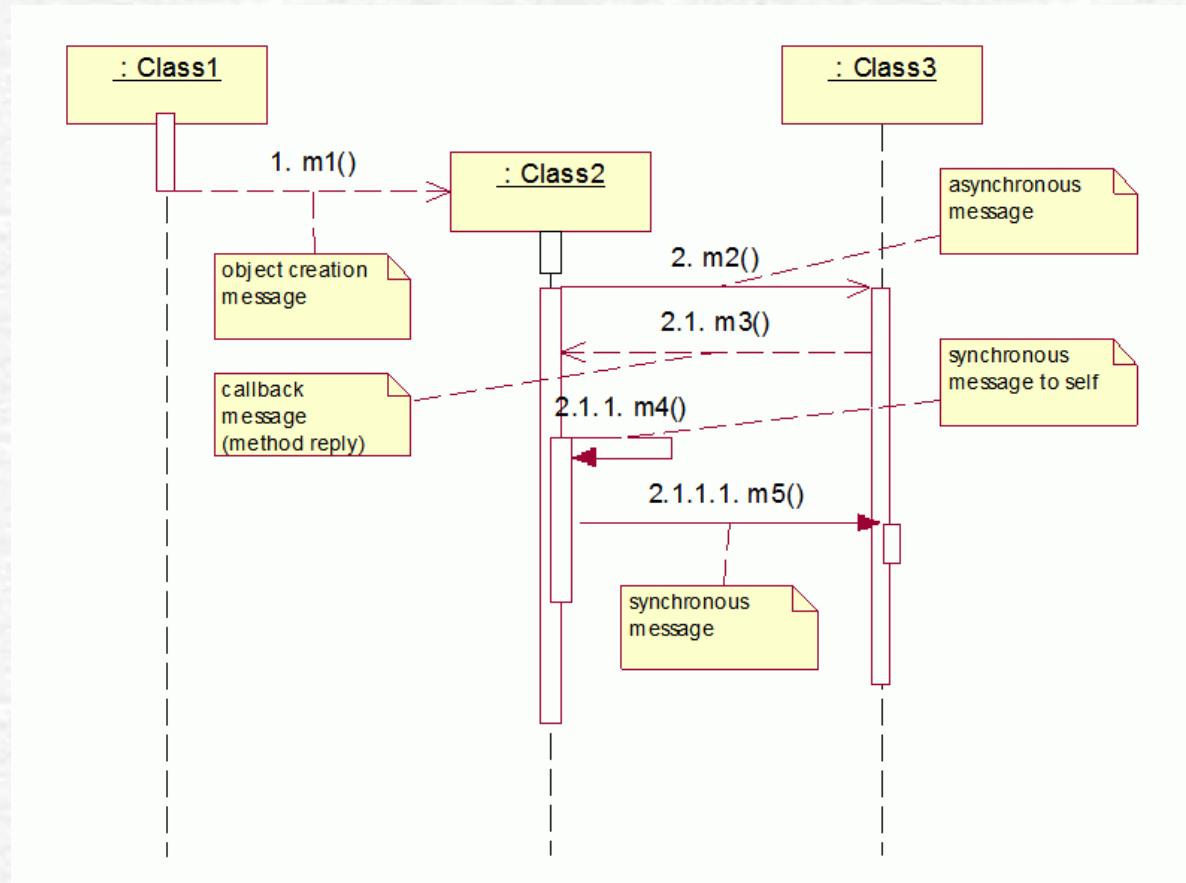
# Diagramme Use Case. Exemplu



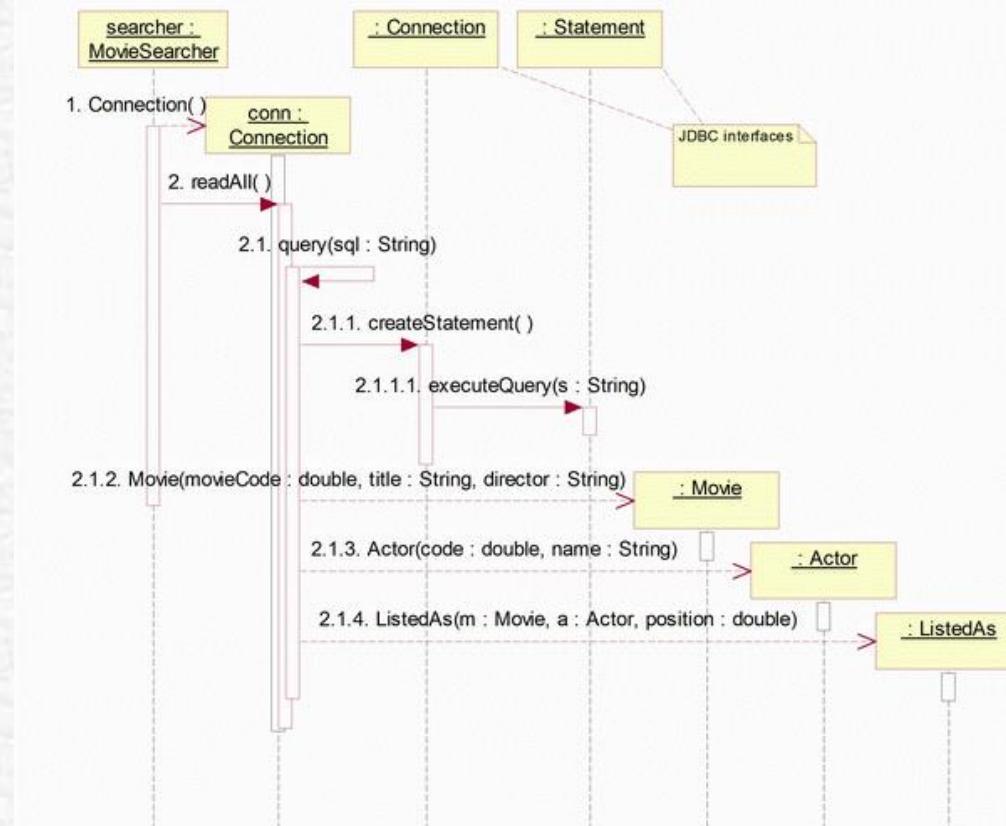
# Diagrame Sequence

- ➊ Diagramele **Interaction** reprezintă principala tehnică de modelare la nivel de **proiectare a comportamentului**
- ➋ Există două tipuri de diagrame **Interaction: Sequence și Collaboration**
- ➌ Diagramele **Sequence** (diagramele de secvență) sunt reprezentări grafice a secvenței **mesajelor dintre obiecte**
  - Secvența poate fi reprezentată prin plasarea mesajelor unul sub celălalt
  - Optional se poate recurge și la o numerotare a mesajelor care să denote ordinea lor
- ➍ Obiectul care recepționează un mesaj va activa metoda corespunzătoare acestuia
- ➎ Timpul în care fluxul de control ajunge la un obiect se numește **activare**
  - Se reprezintă printr-un dreptunghi ascuțit pe linia de viață a obiectului

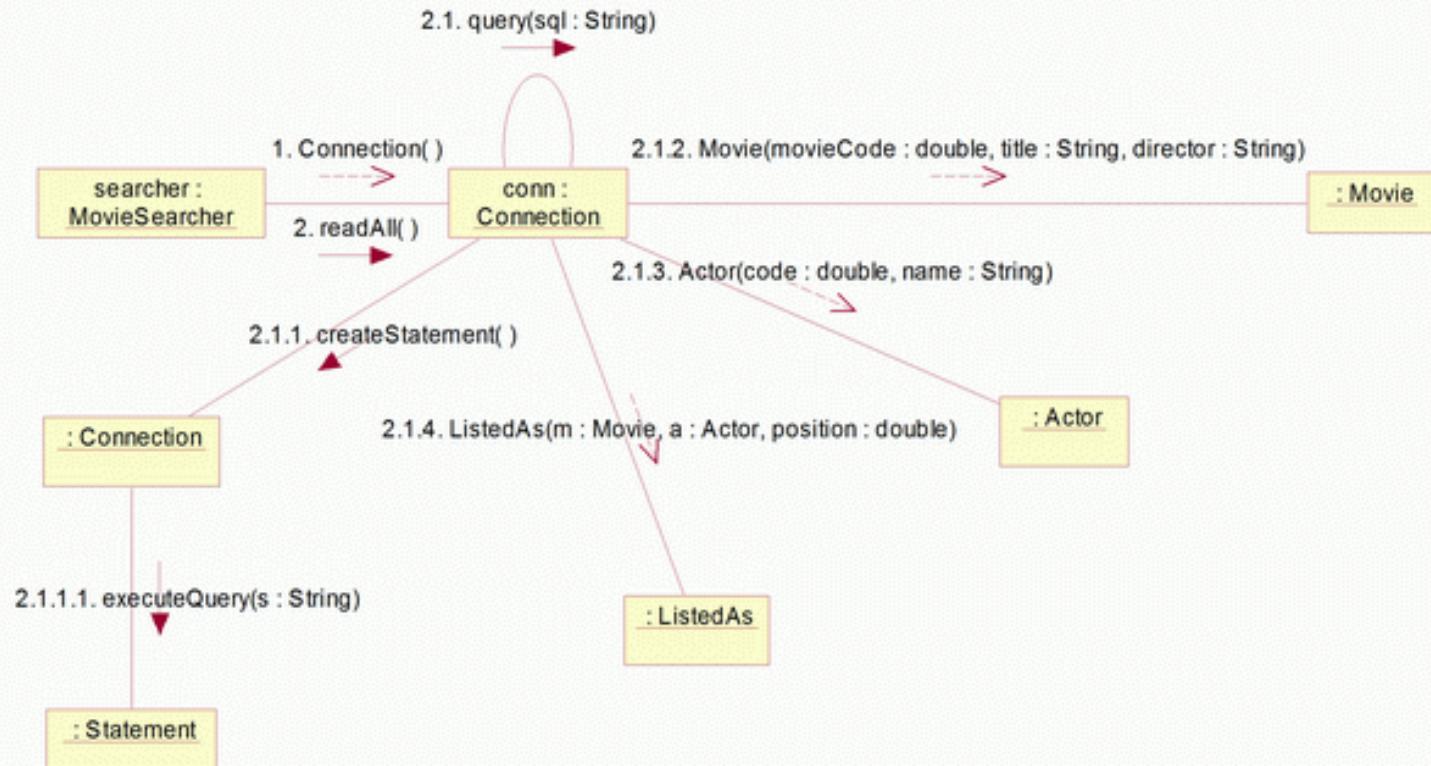
# Diagramme Sequence. Reprezentarea Mesajelor



# Diagramme Sequence. Exemplu



# Diagramme Collaboration. Exemplu

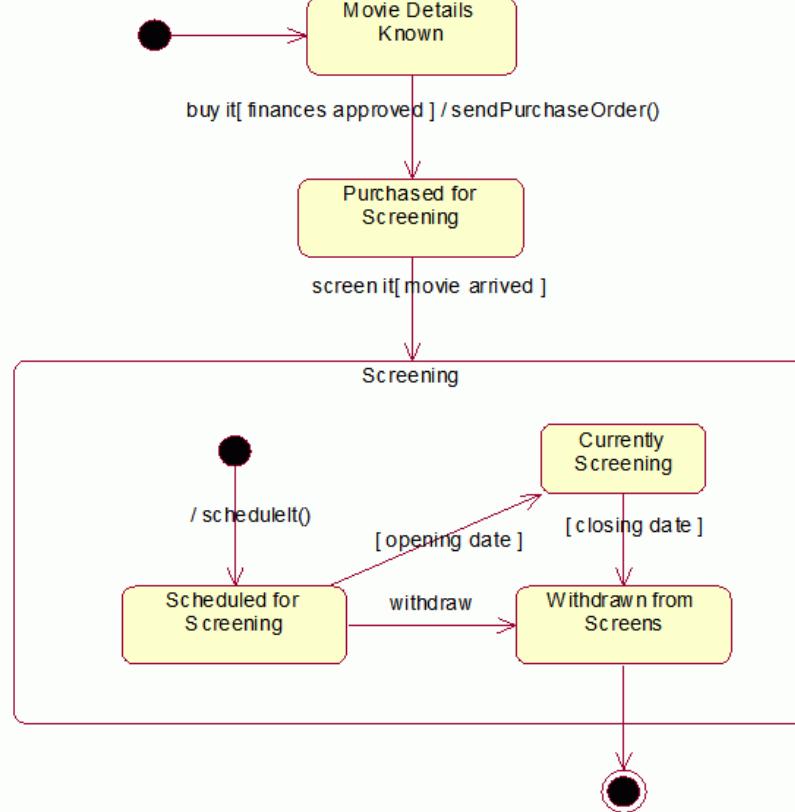


# Diagrame Statechart

- Diagramele **Statechart** nu sunt specifice modelării orientate pe obiecte
- Surprind **starea** unui obiect și **acțiunile** care conduc la schimbarea stării acelui obiect
- Reprezentate pentru fiecare clasă în parte care are o schimbare de stare demnă de luat în seamă pentru model
- Starea unui obiect (instanță de clasă) se schimbă când se modifică valoarea unor attribute ale sale
- Starea are o anumită durată care corespunde cu intervalul de timp dintre două tranziții

event-signature [guard-condition] /action-expression

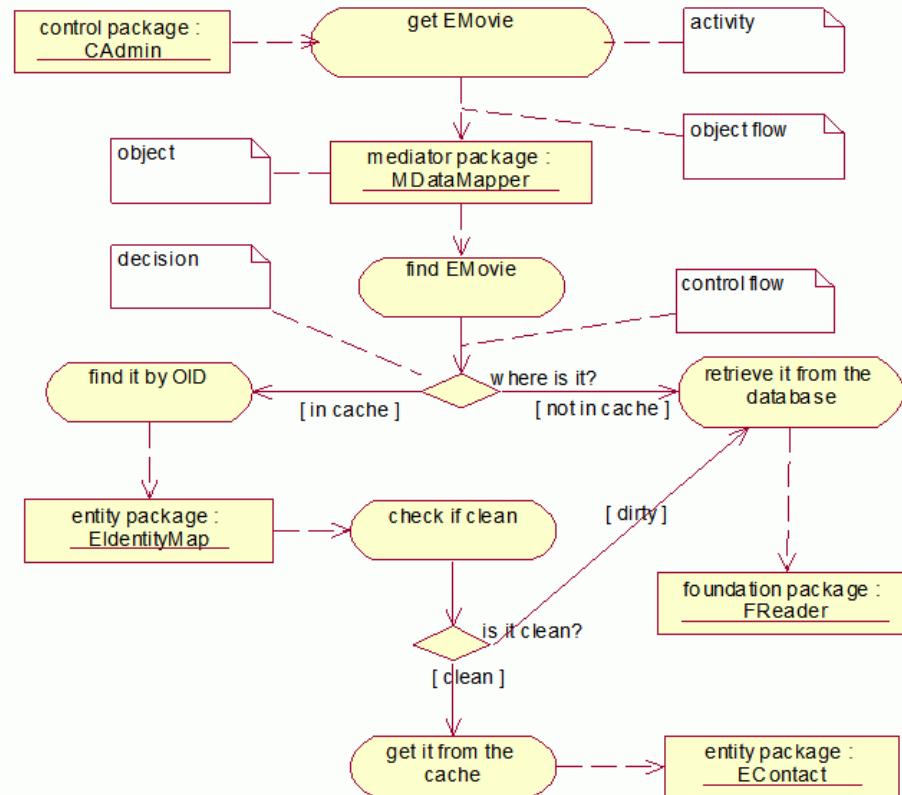
# Diagram Statechart. Exemplu



# Diagrame Activity

- Diagramele **Activity** sunt automate cu stări care reprezintă un calcul, acțiunile executate și tranzițiile rezultate din efectuarea acțiunilor
- În mod normal o diagramă **Activity** este atașată unei implementări de operații sau unui caz de utilizare
- Action states** (acțiunile de stare) sunt acțiuni care nu trebuie întrerupte de evenimente externe. Ele nu trebuie să aibă nici o tranziție de ieșire bazată pe evenimente explicite
- Tranzițiile de ieșire dintr-o acțiune de stare sunt rezultatul încheierii activităților din acea stare

# Diagramme Activity. Exemplu



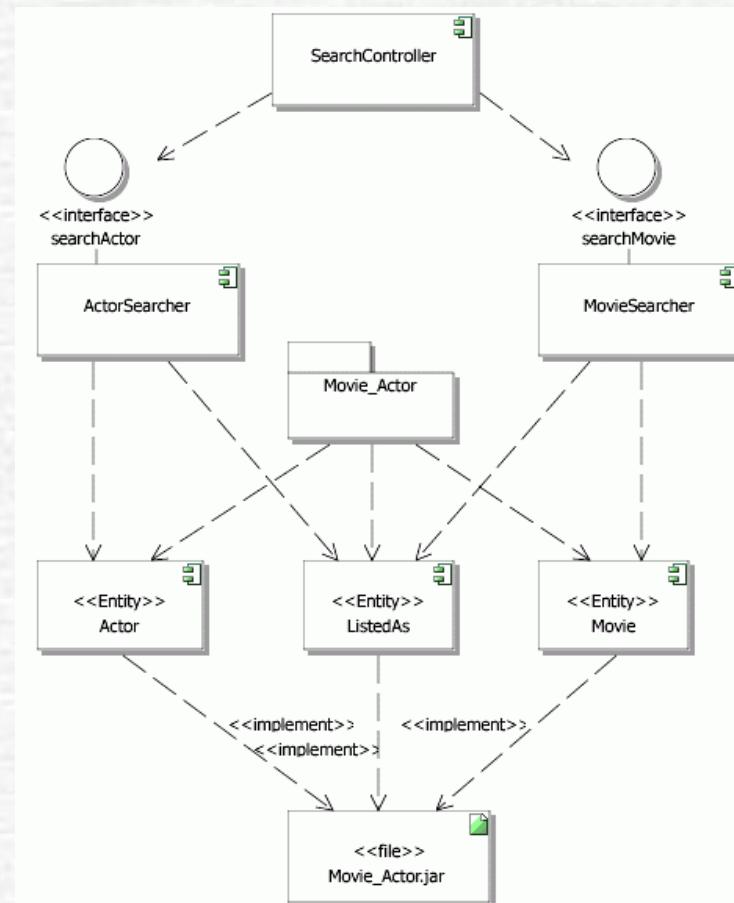
# Diagrame Implementation

- Sunt modele pentru **implementarea fizică a sistemului**
- Evidențiază **componentele sistemului, structura lor și dependențele**. De asemenea arată cum sunt ele plasate pe diverse calculatoare din rețea
- Există două tipuri de diagrame:
  - **Component**
  - **Deployment**
- Diagramele **Component** arată structura componentelor, inclusiv interfețele lor și implementarea dependențelor
- Diagramele **Deployment** arată instalarea pentru rulare pe calculatoarele din rețea

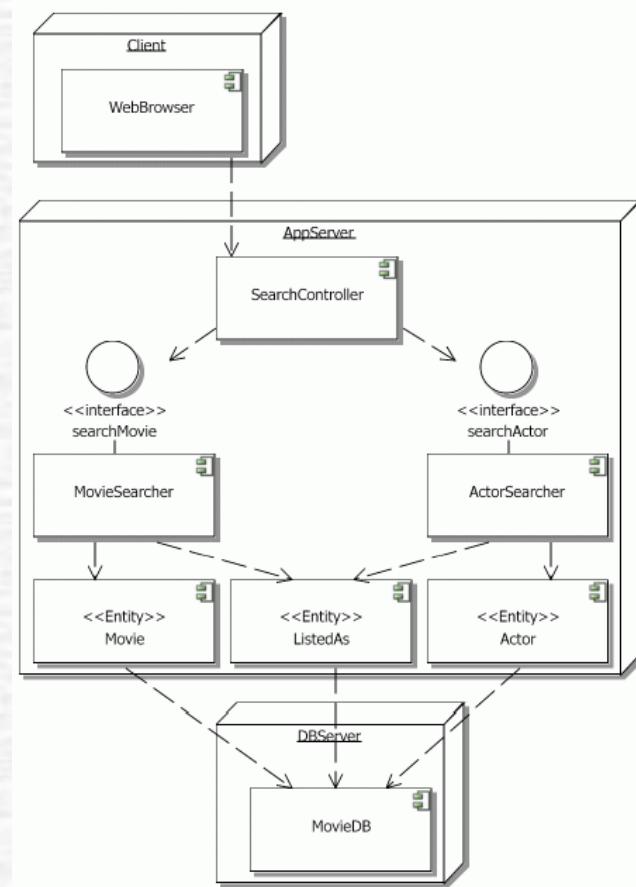
# Diagramme Component

- „A **component diagram** shows the dependencies among software components, including the classifiers that specify them (for example, implementation classes) and the artifacts that implement them; such as, source code files, binary code files, executable files, scripts.” (UML, 2003b, p.3-169)
- „A **component** represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.” (UML, 2003b, p.3-174)
- O componentă expune un set de interfețe care reprezintă serviciile oferite de elementele care compun componenta
- Componentele pot fi conectate la alte componente prin înglobarea lor fizică în acestea (relații de tip «**reside**» sau «**implement**»)
- Pentru ilustrarea grupării componentelor se pot utiliza **Pachete**

# Diagram Component. Exemplu



# Diagramme Deployment. Exemplu



# Concluzii

---

- Limbajul **UML** este un limbaj de modelare standard pentru sistemele software orientate pe obiecte
- Limbajele potrivite pentru modelare structurată sunt: **Data Flow Diagrams (DFDs)**, **Entity-Relationship Diagrams (ERDs)** și **Structure Charts**
- **UML** include diferite tipuri de diagrame precum: **Class**, **Use Case**, **Interaction**, **Statechart**, **Activity** și **Implementation**.
- Modelarea orientată pe obiecte în **UML** este centrată pe clase dar este condusă de diagramele **Use Case**

# Concluzii

- Diagramele **Interaction** reprezintă principala tehnică de modelare la nivel de proiectare a comportamentului
- Diagramele **Statechart** surprind **starea** unui obiect și **acțiunile** care conduc la schimbarea stării acelui obiect
- Diagramele **Activity** sunt automate cu stări și reprezintă un calcul
- Diagramele **Implementation** sunt modele pentru implementarea fizică a sistemului



# Fundamente de Inginerie Software

## Cap. 5

### Proiectarea Sistemelor Software.

**Conf.Dr.Ing. Dan Pescaru**

Textbooks: Sommerville "Software Engineering 7", 2004, Cap. 11

Maciaszek "Practical Software Engineering", 2005, Cap. 9

Sursă: <http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/>

<http://www.comp.mq.edu.au/books/pse/>

**2009**



# Arhitectura unui Sistem Software

---

- **Arhitectura unui Sistem Software** este organizarea tuturor elementelor software ținând cont de:
  - **Suportabilitate** (claritate + mentenabilitate +scalabilitate)
  - **Managementul interdependențelor**
  - **Organizarea în module** specifice (clase, pachete, componente)
  - **Comportamentul modulelor**
  - **Tipare și principii arhitecturale**
- **Proiectarea arhitecturală** este setul de decizii menite să asigure o arhitectură eficientă și logica din spatele acestor decizii

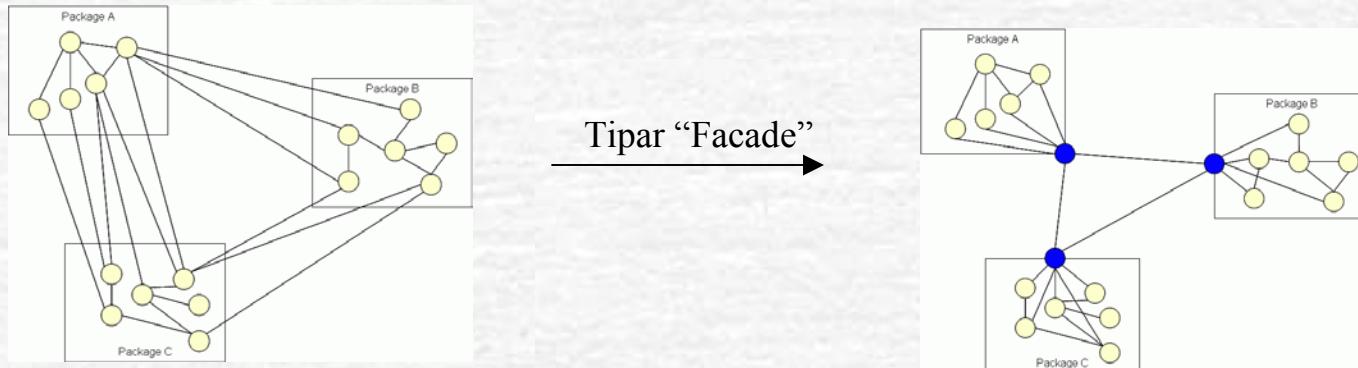
# Dimensiune și Complexitate

## Sisteme “moștenite” (Legacy Systems):

- Monolitice, procesare secvențială și predictibilă
- Complexitate = Dimensiune

## Sisteme obiectuale:

- Distribuite, procesare aleatoare și nepredictibilă
- Complexitatea = Legături, dependențe



# Obiectivele Proiectării Arhitecturale

---

- ➊ Identificarea **componentelor majore** ale sistemului și a **comunicațiilor** dintre acestea
- ➋ Proiectarea unor module software organizate pe **nivele** pentru a reduce complexitatea și a mari gradul de înțelegere a dependențelor dintre module
  - Se interzic intercomunicațiile directe între obiectele de pe nivale vecine

# Caracteristicile unui Sistem

## Performanța

- Localizarea operațiilor critice și minimizarea comunicațiilor. Se recomandă utilizarea componentelor cu granularitate mare

## Securitatea

- Utilizarea unui arhitecturi stratificate cu plasarea secvențelor critice pe nivelele interne

## Siguranța

- Localizarea detaliilor critice din punct de vedere al siguranței într-un număr mic de sub-sisteme

## Disponibilitatea

- Include componente redundante și mecanisme tolerante la erori

## Mantenabilitatea (Ușurința la întreținere)

- Utilizarea unor componente fine, ușor de înlocuit

# Conflicturi Arhitecturale

---

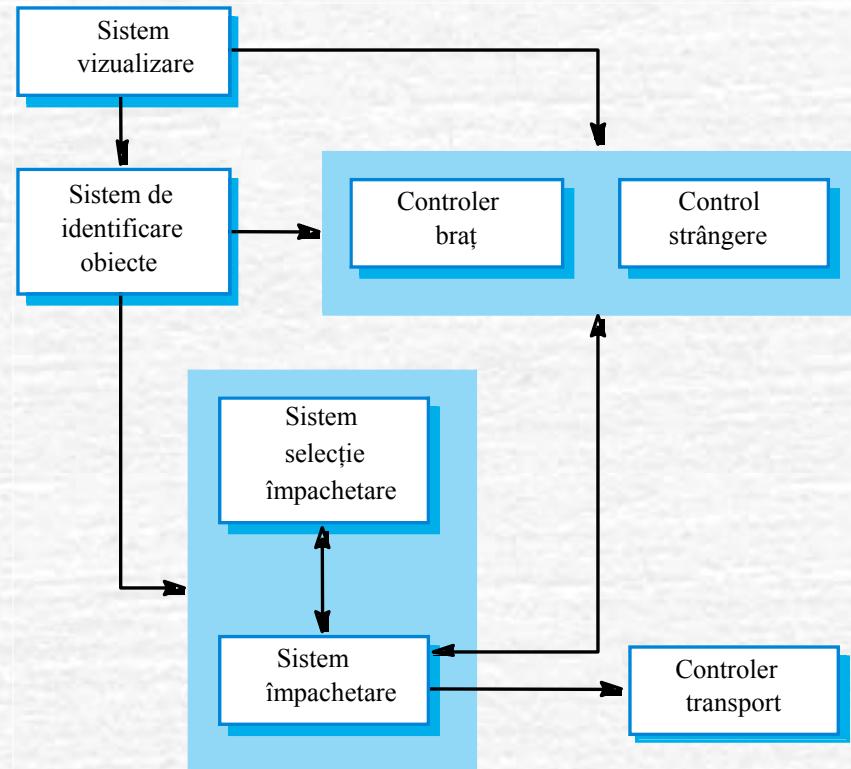
- Utilizarea unor componente de granularitate mare îmbunătățește performanța dar reduce menținabilitatea
- Introducerea unor date redundante crește disponibilitatea dar face mult mai dificilă asigurarea securității
- Localizarea detaliilor critice din punct de vedere al siguranței de obicei duce la creșterea comunicațiilor și deci scăderea performanței

# Structurarea Sistemului

- Prin structurare se urmărește descompunerea sistemului în subsisteme care interacționează între ele
- În mod normal proiectarea arhitecturală este exprimată ca o diagramă bloc
- Diagramele care descriu arhitectura oferă o privire de ansamblu asupra sistemului

## Exemplu

- ▶ Sistem de control pentru un robot de împachetat (Sommerville '04):



# Decizii la Proiectarea Arhitecturală

---

- ➊ Proiectarea este un proces creativ. Totuși este bine de considerat următoarele întrebări la luarea deciziilor:
  - Există o arhitectură generică ce poate fi utilizată?
  - Cum va fi distribuit sistemul?
  - Ce stil arhitectural se potrivește?
  - Ce concept poate fi aplicat pentru structurarea sistemului?
  - Cum va fi sistemul descompus pe module?
  - Ca strategie de control se poate utiliza?
  - Cum va fi evaluată proiectarea arhitecturală?
  - Cum va fi documentată arhitectura propusă?

# Reutilizarea Arhitecturii

- Sistemele dintr-un anumit domeniu au foarte des arhitecturi similare. Aceste arhitecturi reflectă conceptele din respectivele domenii.
- Liniile de produs aplicații sunt create în jurul unei arhitecturi centrale cu variante care să satisfacă cerințe particulare a diversilor clienți

# Stiluri Arhitecturale

---

- Stilul arhitecturii unui sistem se subscrive de regulă unui stil arhitectural generic
- Considerarea unui stil general poate simplifica definirea arhitecturii sistemului
- Totuși, cele mai multe sisteme mari sunt eterogene și nu se conformează unui singur stil

# Modele de Arhitecturi

- Modele **structurale (static)** – scot în evidență componentele majore ale sistemului
- Modele **de proces (dinamice)** – arată structura proceselor sistemului
- Modele **de interfețe** – definesc interfețele diverselor sub-sisteme
- Modele **de relații** (ex. modele de fluxuri de date) – indică relațiile între sub-sisteme
- **Modelul de distribuție** – arată cum sunt distribuite sub-sistemele pe diversele sisteme de calcul

# Organizarea unui Sistem

---

- ➊ Reflectă strategia de bază care este utilizată pentru structurarea unui sistem
- ➋ Cele mai răspândite trei stiluri de organizare sunt:
  - Cu **zonă de date partajată**
  - Stil **server cu servicii partajate**
  - Model organizat pe **nivele** (mașină abstractă)

# Modelul cu zonă de date partajată

- ➊ Sub-sistemele schimbă date în două moduri posibile:
  - Datele partajate sunt reținute într-o bază de date centrală și pot fi accesate de toate sub-sistemele
  - Fiecare sub-sistem menține propria bază de date și transferă datele explicit la celelalte sub-sisteme
- ➋ Modelul cu zonă partajată de date se adresează în principal sistemelor care partajează cantități mari de date

# Caracteristici

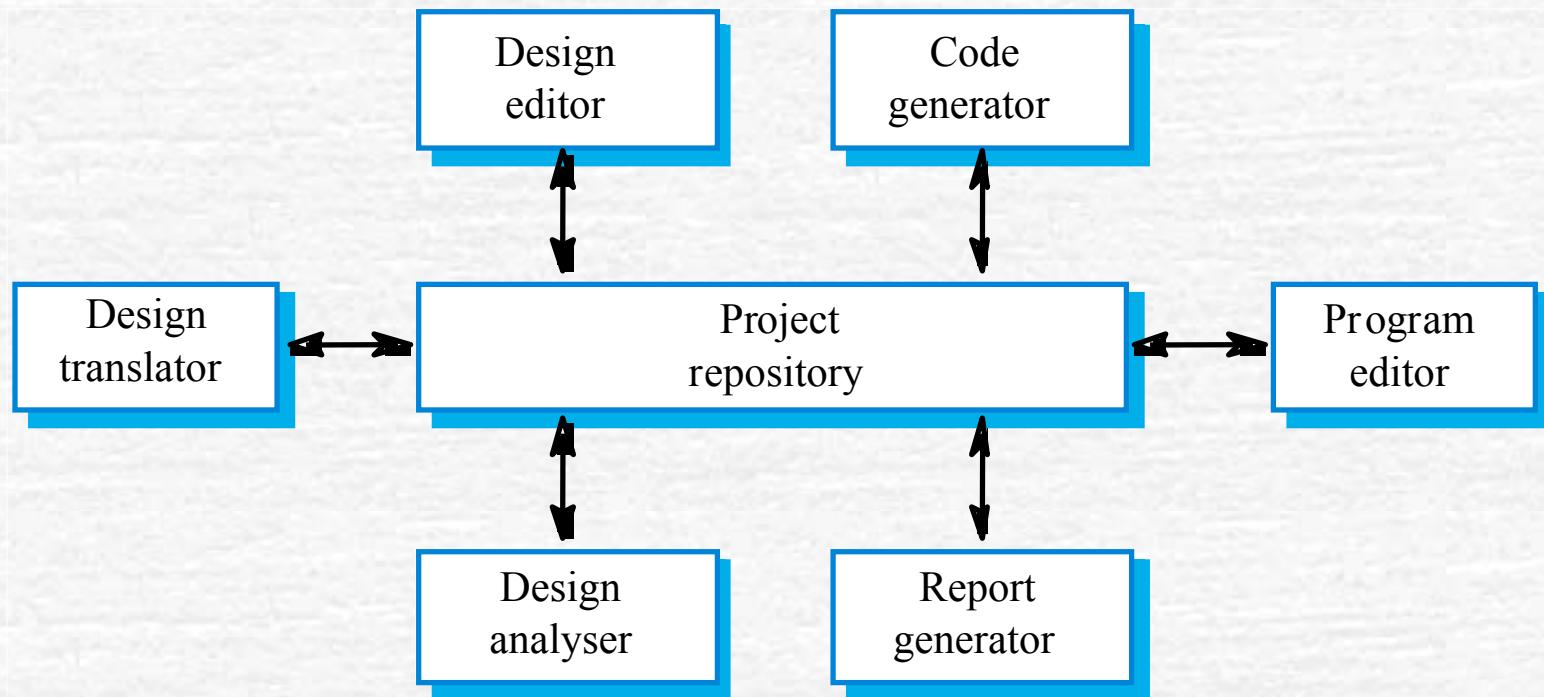
## Avantaje

- Mod eficient de partajare a unor cantități mari de date
- Gestionare centralizată: securitate, backup etc. Sub-sistemele nu sunt influențate de maniera de producere a datelor
- Modelul partajat este prezentat printr-o schemă de depozitare

## Dezavantaje

- Sub-sistemele trebuie să adere la același model de stocare a datelor. Ca atare se vor impune compromisuri
- Evoluția datelor este dificilă și costisitoare
- Nu permite politici de gestiune specifice
- Dificil de distribuit în mod eficient

# Arhitectură CASE tool pentru model



# Modelul Client-Server

- ➊ Un model de sistem distribuit axat pe modul în care datele și procesele sunt distribuite pe mai multe sisteme de calcul
- ➋ Componete:
  - Un set de servere care asigură servicii specifice precum: tipărire, gestionarea datelor etc.
  - Un set de clienți care apelează aceste servicii
  - O rețea care asigură accesul clientilor la servere

# Caracteristici

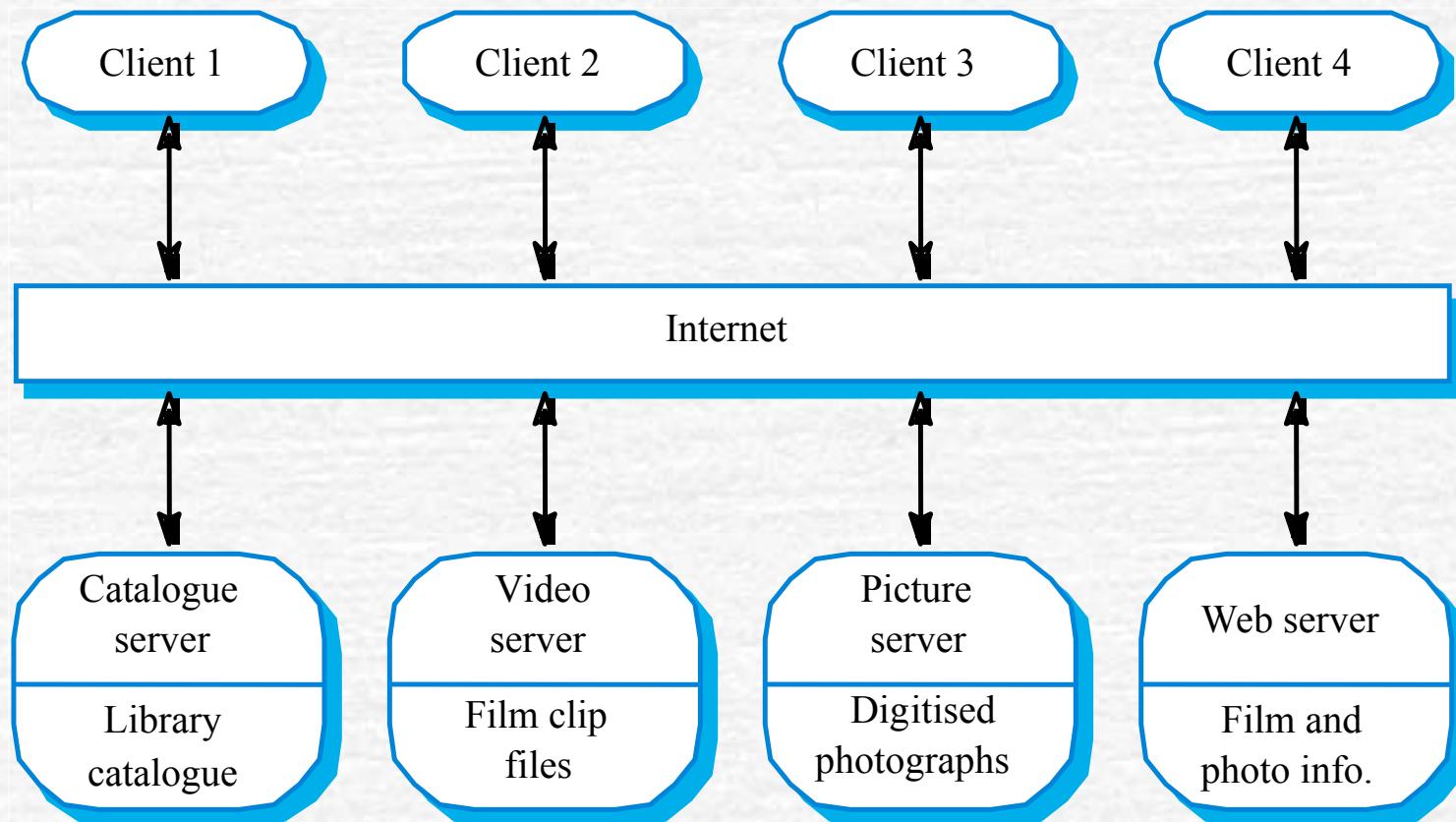
## Avantaje

- Distribuția datelor este directă;
- Utilizează eficient sistemele de rețea. Costuri mici cu hardware-ul
- Ușor de adăugat noi servere sau de îmbunătățit cele existente

## Dezavantaje

- Sub-sistemele pot organiza datele în mod diferit, ceea ce poate duce la un schimb de date ineficient
- Gestiune redundantă la nivelul fiecărui server
- Nu există gestiune centralizată pentru nume și servicii. Poate devenii dificil de găsit ce servere și servicii sunt disponibile

# Exemplu de arhitectură în domeniul media



# Model pe Nivele (mașină abstractă)

---

- Utilizat pentru modelarea interfețelor între subsisteme
- Organizează sistemul într-un set de nivale (sau mașini abstracte), fiecare asigurând un set de servicii
- Setul de servicii definește un “limbaj mașină” pentru nivelul respectiv
- Ex: modelul de rețea OSI

# Model pe Nivele (mașină abstractă)

- Sprijină dezvoltarea incrementală a sub-sistemelor pe diferite nivele. Schimbarea interfeței unui nivel afectează doar nivelele adiacente acestuia
- Este portabil (necessită doar schimbarea nivelelor de bază)
- De multe ori însă este artificial de structurat un sistem în acest mod
- Nivelele superioare pot depinde direct de servicii oferite de nivelele cele mai de jos. Trimiterea cererilor din aproape în aproape pot degrada performanța

# Model pentru gestionarea versiunilor obiectelor

Configuration management system layer

Object management system layer

Database system layer

Operating system layer

(Buxton '80 – Ada Programming Support Environment APSE)

# Descompunerea Sub-sistemelor În Module

---

- Nu se face o distincție rigidă între descompunerea în module și organizarea generală a sistemului
- În general granularitatea modulele este mai fină decât cea a sub-sistemelor
- Sub-sistemele sunt la rândul lor sisteme compuse din module
- Modulele nu sunt sisteme independente

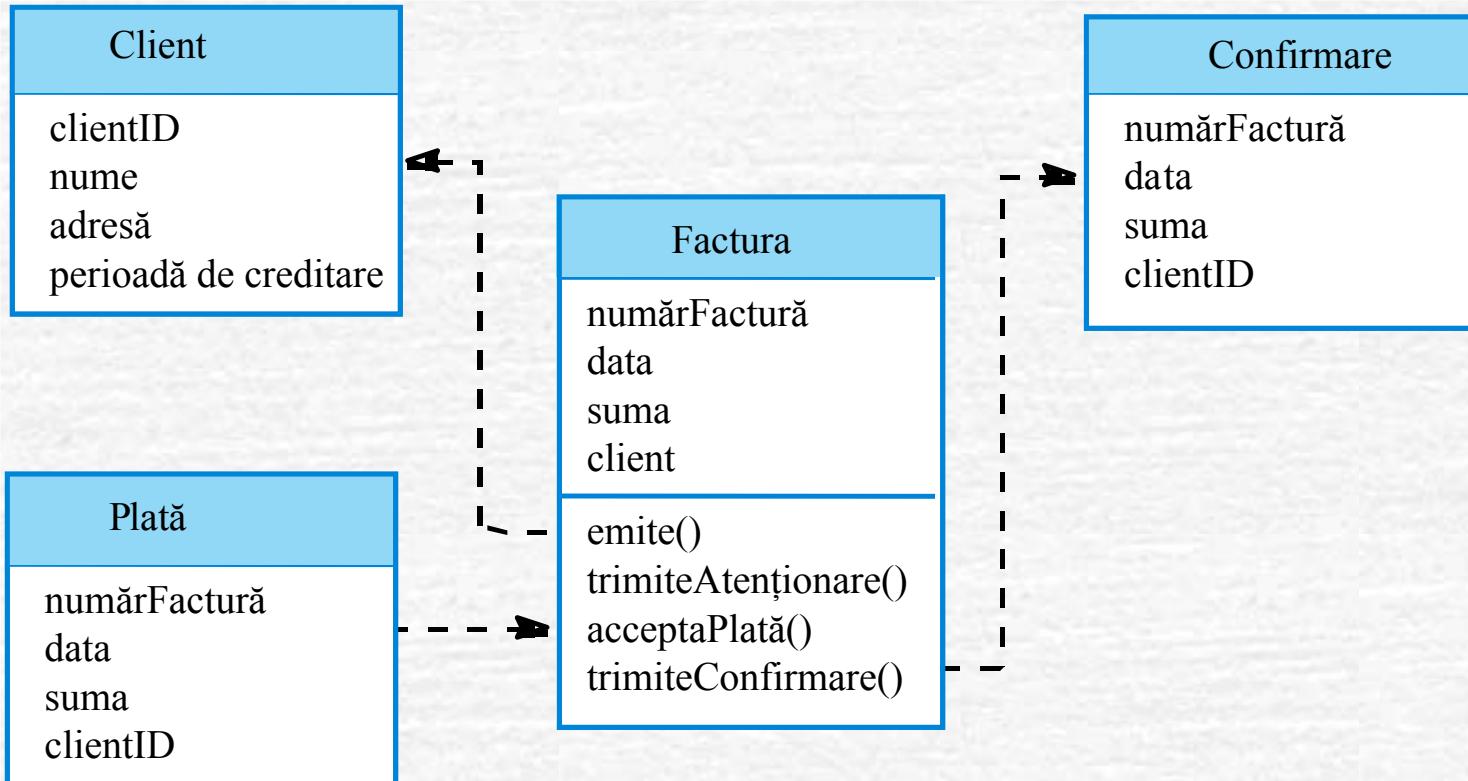
# Descompunerea În Module

- ➊ Două modele principale de descompunere:
  - Un model **obiectual** – un sistem este descompus în obiecte care interacționează între ele
  - Un model **pipeline** (sau flux de date) – un sistem este descompus în module funcționale care transformă intrările în ieșiri
- ➋ Dacă este posibil, deciziile care se referă la accesul concurrent se vor lua abia la implementarea modulelor, programele secvențiale fiind mai ușor de proiectat și implementat

# Modelul Obiectual

- Structurează sistemul într-un set de obiecte slab cuplate dar cu interfețe bine definite
- Pașii principali constau în identificarea claselor obiectelor a atributelor și a operațiilor lor
- Obiectele mențin o stare proprie (privată) și definesc operațiile prin care aceasta poate fi modificată

# Model obiectual pentru procesat facturi



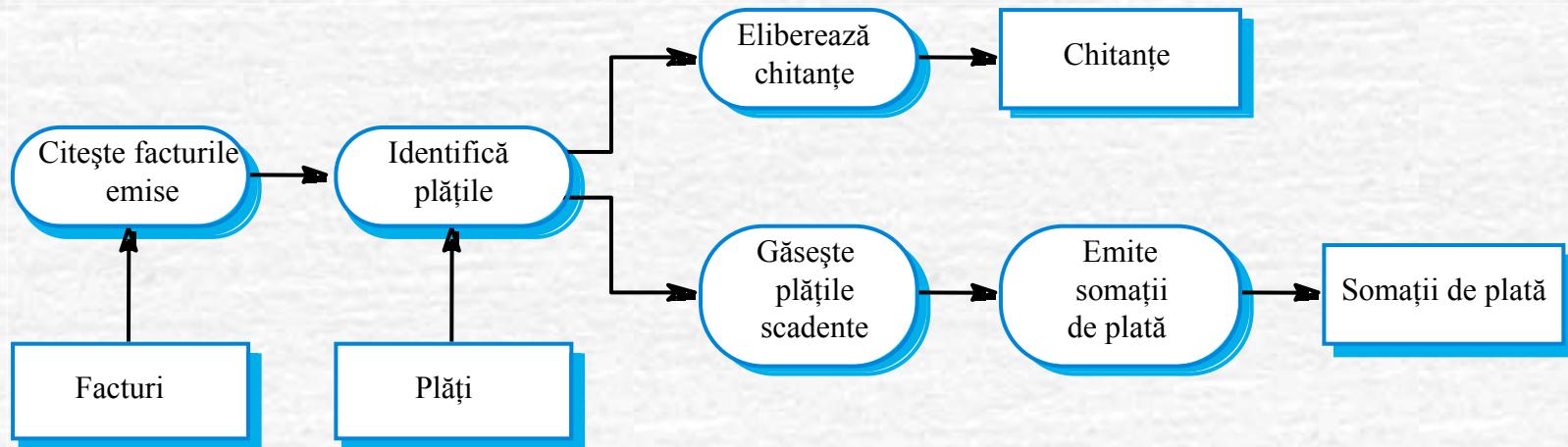
# Avantajele Modelul Obiectual

- Cuplarea între obiecte este slabă ca atare implementarea uneia poate fi modificată fără a afecta celelalte obiecte
- Obiectele pot reflecta în mod natural entitățile din lumea reală
- Limbajele orientate pe obiecte sunt larg utilizate
- Dezavantaje:
  - o schimbare în interfețele obiectelor poate crea probleme
  - entitățile complexe pot fi greu de reprezentat ca obiecte

# Modelul Funcțional Pipeline

- Procesează intrările în ieșiri conform unor transformări funcționale
- Asemănător cu modelul pipe din shell-ul UNIX
- Variante ale acestui concept sunt des întâlnite. Când transformările sunt secvențiale, acesta este un model de prelucrare a loturilor de date foarte comun în sistemele de procesare a datelor
- Nu este recomandabil în cazul sistemelor interactive cu intrări complex (tastatură, GUI, mouse etc.)

# Model Pipeline pentru procesare facturi



# Avantajele Modelului Pipeline

---

- Sprijină reutilizarea transformărilor
- Organizare intuitivă
- Ușor de adăugat noi transformări
- Relativ ușor de implementat atât concurrent cât și secvențial
- Dezavantaje:
  - necesită un format comun pentru transferul datelor de-a lungul pipeline-ului
  - interacțiunea bazată pe evenimente este dificil de implementat

# Stiluri de Control

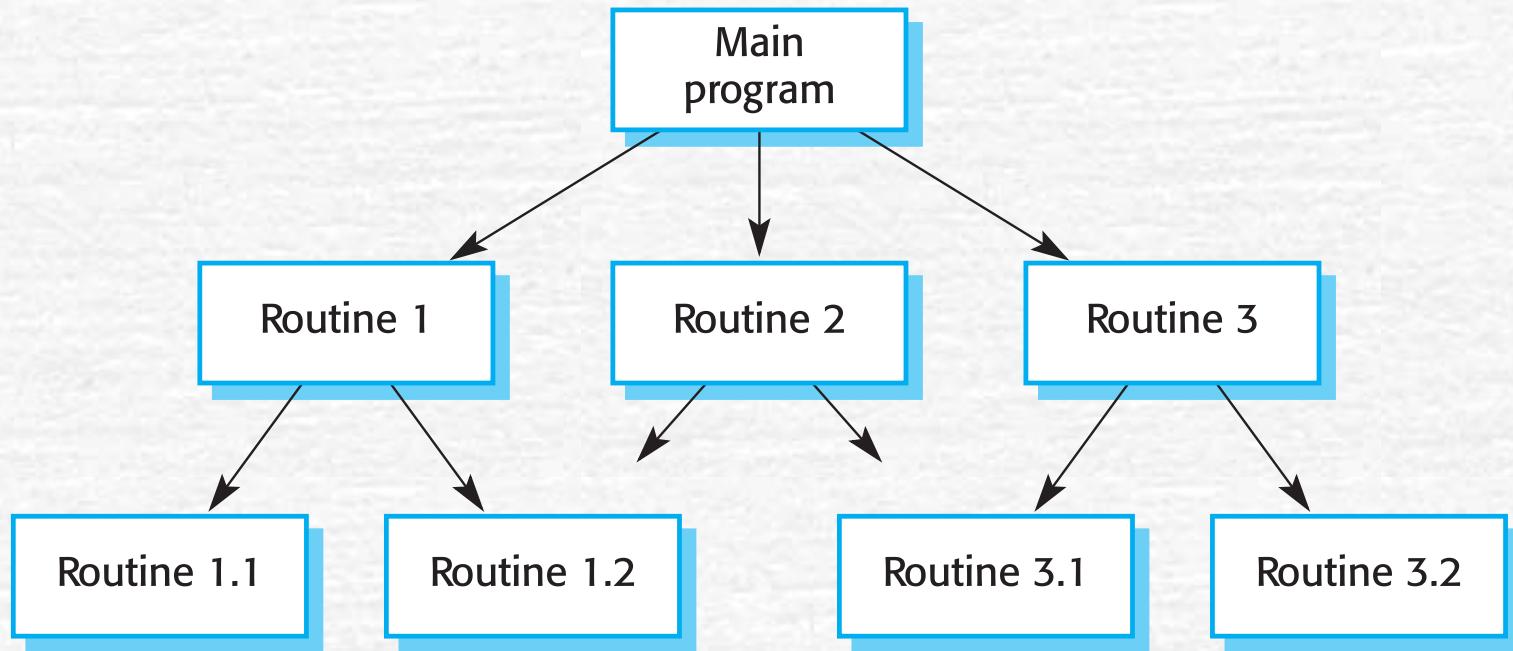
---

- Stabilește controlul fluxului între sub-sisteme (distinct față de modelul de decompoziție)
- Control centralizat
  - Un sub-sistem are răspunderea globală de a controla, porni și opri alte sub-sisteme
- Control bazat pe evenimente
  - Fiecare sub-sistem poate răspunde la evenimente venite din exterior de la alte sub-sisteme sau de la mediul în care sistemul funcționează

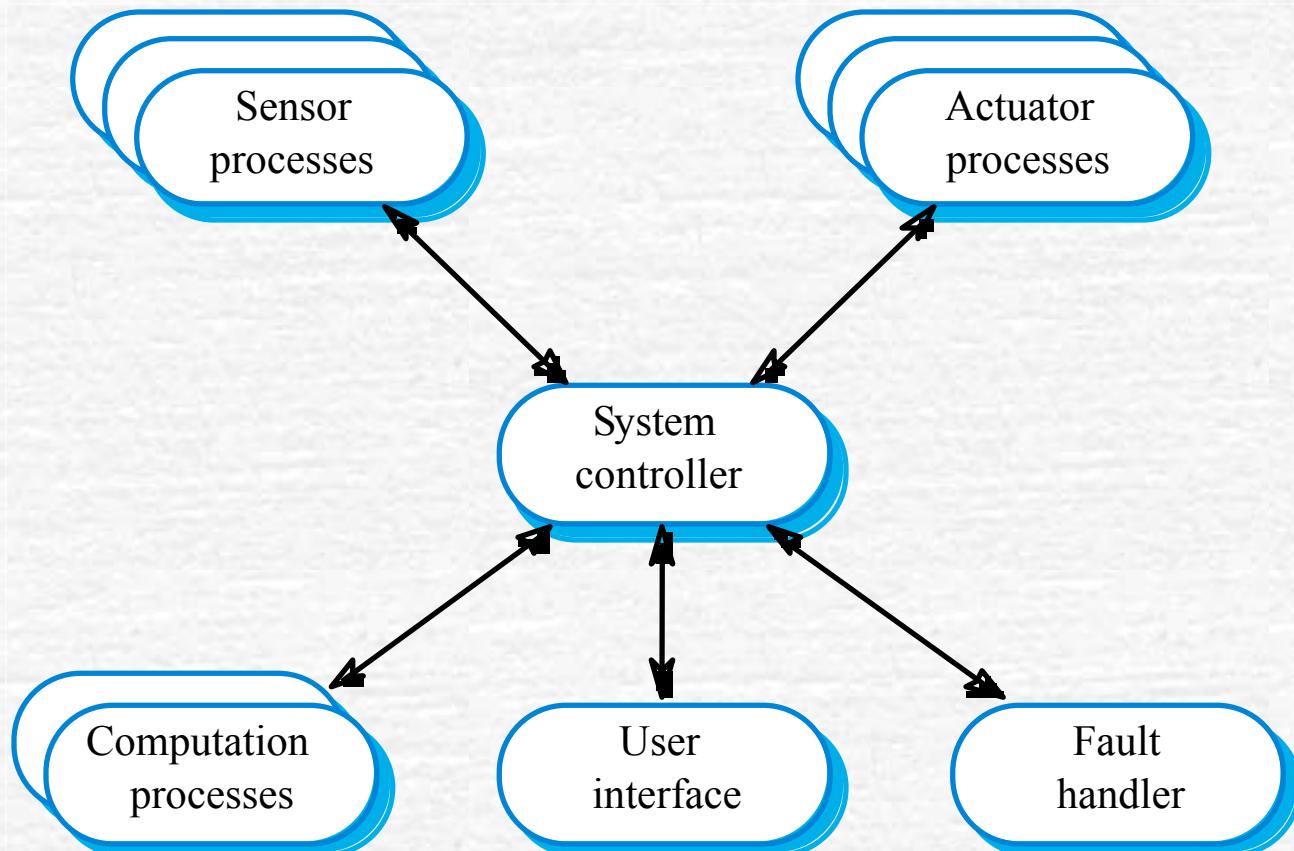
# Control Centralizat

- Un sub-sistem de control are responsabilitatea gestionării execuției altor sub-sisteme (decizii luate pe baza valorilor unor variabile de stare)
- Modelul apel-răspuns
  - Model top-down la care controlul pornește de la subrutele cele mai de sus din ierarhie și migreză în jos. Este aplicabil sistemelor secvențiale
- Modelul “gestionar”
  - Aplicabil în cazul sistemelor concurente. O componentă de sistem controlează oprirea, pornirea și coordonarea altor procese din sistem. Poate fi implementat în sisteme secvențiale ca și o secvență condițională cu valori multiple

# Modelul Apel-Răspuns



# Controlul unui sistem de timp-real



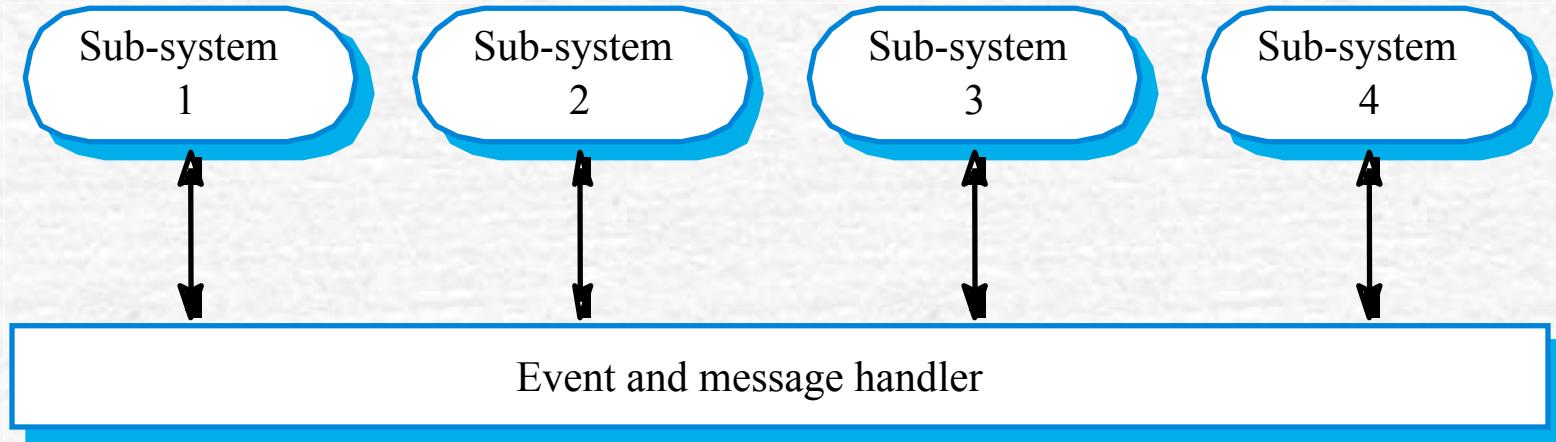
# Control Bazat pe Evenimente

- ➊ Condus de evenimente generate extern, independente de controlul sub-sistemelor care procesează evenimentele
- ➋ Două modele principale:
  - Modelul **broadcast**. Un eveniment este trimis tuturor sub-sistemelor. Orice sub-sistem poate procesa evenimentul
  - Modelul bazat pe **întreruperi**. Utilizat în sistemele de timp real unde îintreruperile sunt detectate de un handler de îintrerupere și trimise altor componente pentru procesare

# Modelul Broadcast

- Eficient la integrarea sub-sistemelor de pe diverse calculatoare într-o rețea
- Sub-sistemele se înregistrează pentru evenimentele de interes. Când un astfel de eveniment apare, controlul este transferat sub-sistemului care poate trata acel eveniment
- Politica de control nu este inclusă în eveniment sau în gestionarul de mesaje. Sub-sistemele decid ce evenimente le interesează
- Totuși, sub-sistemele nu știu dacă sau când un eveniment va fi tratat de alte sub-sisteme

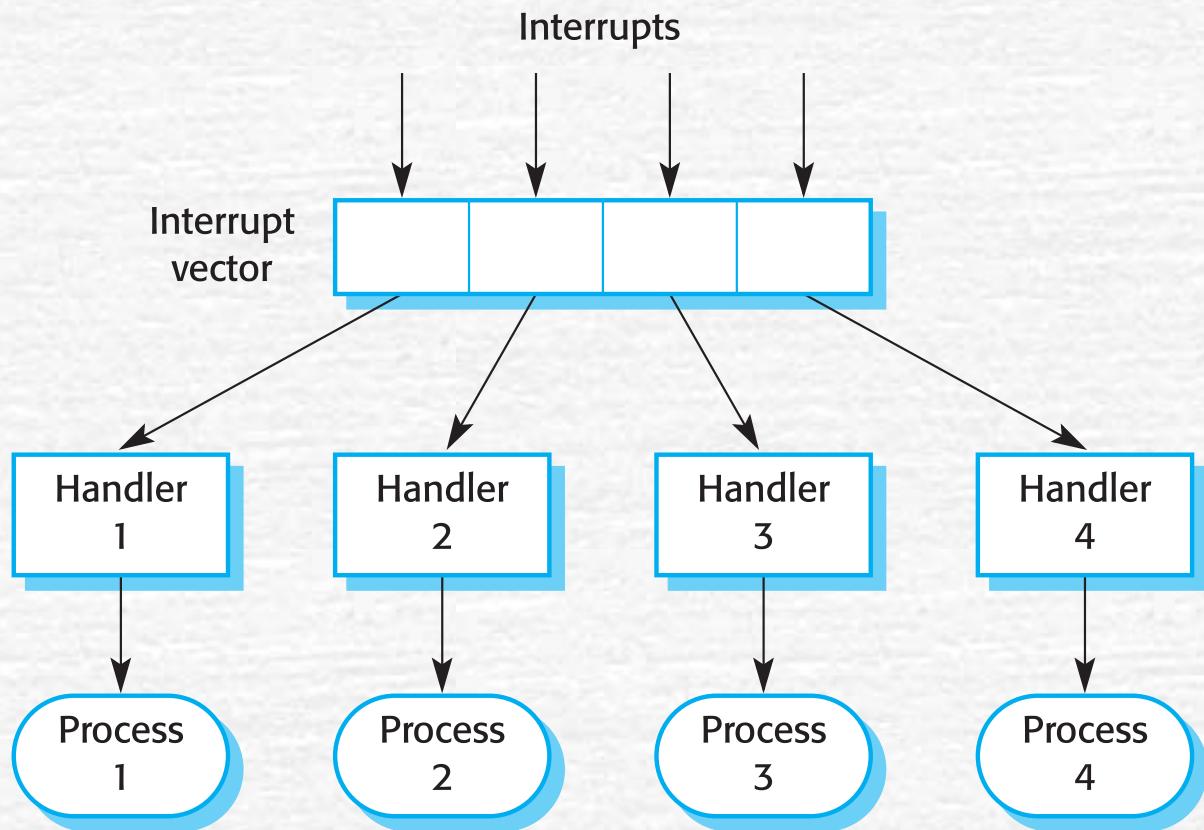
# Broadcast Selectiv



# Modelul Bazat pe Întreruperi

- Utilizat în sistemele de timp-real unde răspunsul rapid la evenimente este esențial
- Se bazează pe un set de tipuri cunoscute de întreruperi fiecare având definit câte un handler (rutină de tratare)
- Fiecare tip are asociată o locație de memorie. Un comutator hardware va face transferul către handlerul corespunzător
- Permite un răspuns rapid dar este complex de programat și dificil de validat

# Control Bazat pe Întreruperi



# Arhitecturi specifice

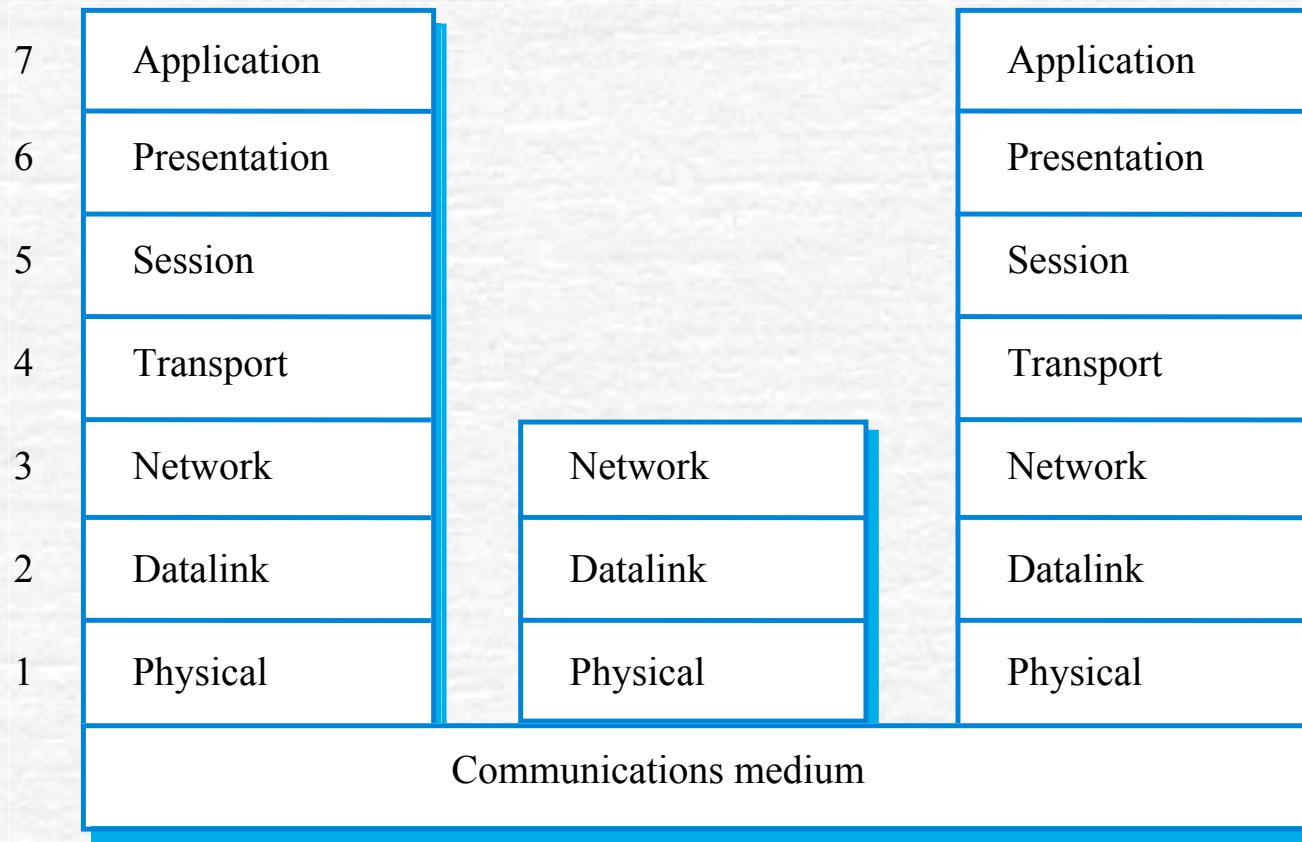
---

- ➊ Modelele arhitecturale pot fi specifice unui anumit domeniu
- ➋ Există două tipuri de modele specifice unui domeniu
  - Modele **generice** – abstracții ale unui număr mare de sisteme reale și care încapsulează caracteristicile principale ale acestora
  - Modele de **referință** – mai abstracte (modele ideale). Asigură o privire largă asupra clasei respective de sisteme și ajută la compararea diverselor arhitecturi
- ➌ Modelele generice sunt în general de tip bottom-up pe când cele de referință sunt de tip top-down

# Modele de referință

- Modelele de referință sunt derivate din studierea domeniului aplicației mai degrabă decât din studierea sistemelor existente
- Pot fi utilizate ca o bază pentru implementare sau pentru compararea sistemelor
- Pot juca rolul unui standard față de care sistemele pot fi evaluate
- Ex: Modelul OSI – un model pe nivele pentru sistemele de comunicații

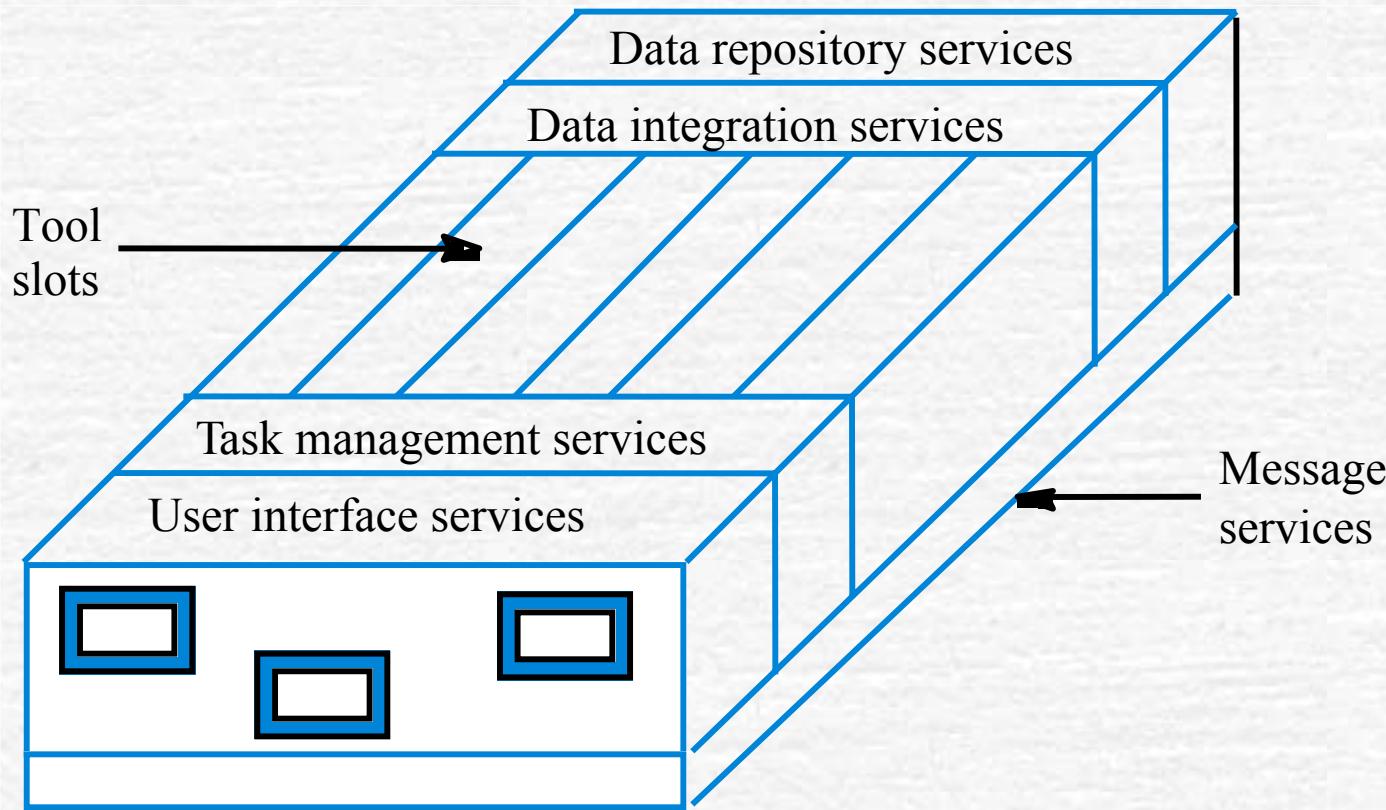
# Modelul de referință OSI



# Modelul de referință pentru CASE

- Modelul asigură următoarele servicii pentru un mediu CASE:
  - Data repository services
    - Stocarea și gestionarea datelor
  - Data integration services
    - Gestionarea grupurilor de entități
  - Task management services
    - Definirea modelelor de proces. Sprijin pentru integrarea proceselor
  - Messaging services
    - Comunicații între uneltele CASE și între unelte și mediu
  - User interface services
    - Dezvoltarea interfețelor utilizator

# Modelul ECMA (1991)



# Concluzii

---

- Arhitectura software este cadrul fundamental pentru structurarea unui sistem
- Proiectarea arhitecturală include deciziile referitoare la arhitectura aplicației, la distributivitate și la stilul arhitectural care va fi utilizat
- Pot fi dezvoltate diferite modele arhitecturale precum modele structurale, modele de control și modele de decompoziție
- Modelele de organizare a sistemelor includ modele de stocare, modele client-server și modele pe nivele (mașini abstracte)

# Concluzii

- Modelele de decompoziție a modulelor includ modele obiectuale respectiv pipeline
- Modelele de control includ controlul centralizat și modelele bazate pe evenimente
- Arhitecturile de referință pot fi utilizate pentru explicitarea arhitecturilor specifice unor domenii. De asemenea ele pot fi utilizate pentru evaluarea și compararea proiectelor arhitecturale

# Atribute ale Arhitecturii

## Performanță

- Localizarea operațiilor pentru minimizarea comunicațiilor la nivelul sub-sistemelor

## Securitatea

- Utilizarea unui arhitecturi pe nivele care să păstreze aspectele sensibile pe nivelele interioare

## Siguranță

- Izolarea componentelor critice

## Disponibilitatea

- Includerea unor componente redundante în arhitectură

## Mantenabilitatea

- Utilizarea unor componente de granularitate fină

## The Open-Closed Principle

*Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*

**1. They are “Open For Extension”.** This means that the behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.

**2. They are “Closed for Modification”.** The source code of such a module is inviolate. No one is allowed to make source code changes to it.

If we wish for a Client object to use a different server object, then the Client class must be changed to name the new server class.

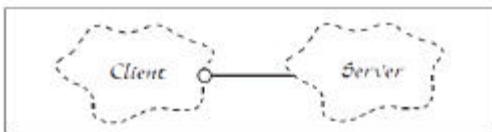


Figure 1  
Closed Client

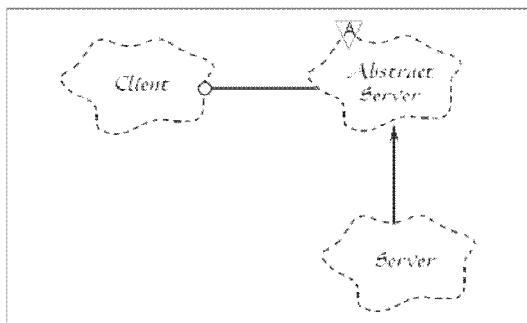


Figure 2  
Open Client

Member Variables Private.

No Global Variables.

## The Dependency Inversion Principle

**A. High level modules should not depend upon low-level modules. Both should depend upon abstractions.**

**B. Abstractions should not depend upon details. Details should depend upon abstractions.**

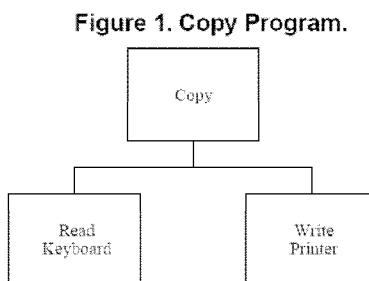


Figure 1. Copy Program.

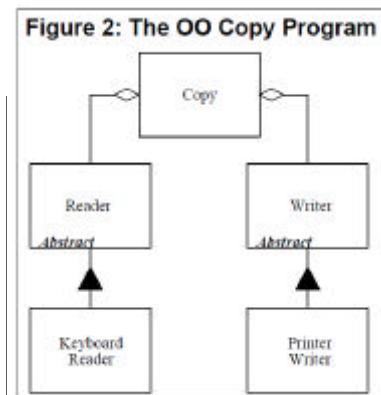
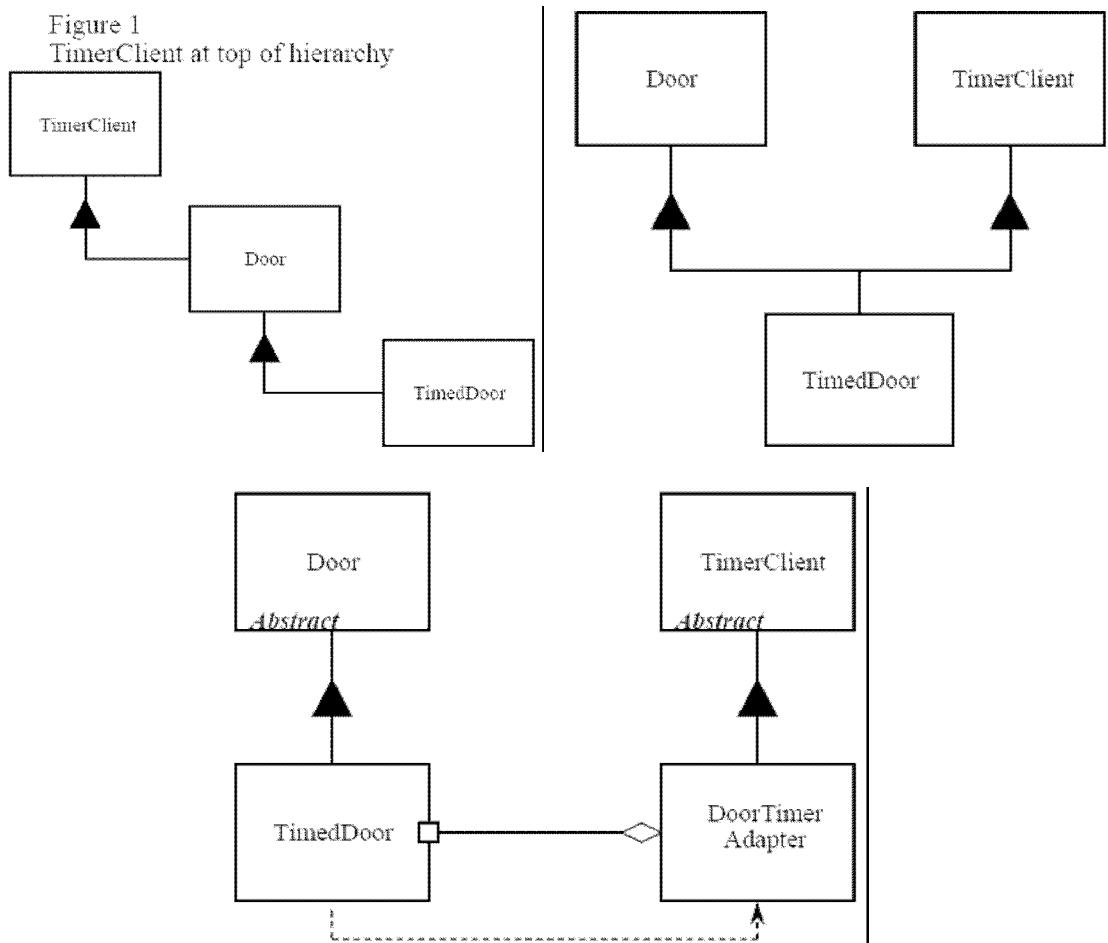


Figure 2: The OO Copy Program

## The Interface Segregation Principle

*Clients should not be forced to depend upon interfaces that they do not use.*



Even if the change to Timer were to be made, none of the users of Door would be affected. Moreover, TimedDoor does not have to have the exact same interface as TimerClient.

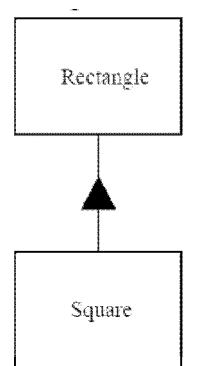
## Liskov Substitution Principle

**If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.**

LSP is a particular definition of a subtyping relation, called (strong) behavioral subtyping.

The dimensions of a Square cannot (or rather should not) be modified independently.

> More information on Wikipedia.





# Fundamente de Inginerie Software

## Cap. 6

### Dezvoltarea Sistemelor Software.

Conf.Dr.Ing. Dan Pescaru

Textbooks: Sommerville "Software Engineering 7", 2004, Cap. 17, 18  
Sursă: <http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/>



2009

# Dezvoltarea Rapidă a Aplicațiilor (RAD)

---

- ➊ Datorită schimbărilor rapide în mediul de afaceri, companiile trebuie să răspundă noilor oportunități și competiției
- ➋ Aceasta necesită dezvoltarea și livrarea rapidă a sistemelor software adecvate, poate cea mai critică cerință pentru aceste sisteme
- ➌ Afacerile pot accepta software de calitate chiar mai scăzută dacă este posibilă astfel livrarea rapidă a funcționalității esențiale

# Cerințe

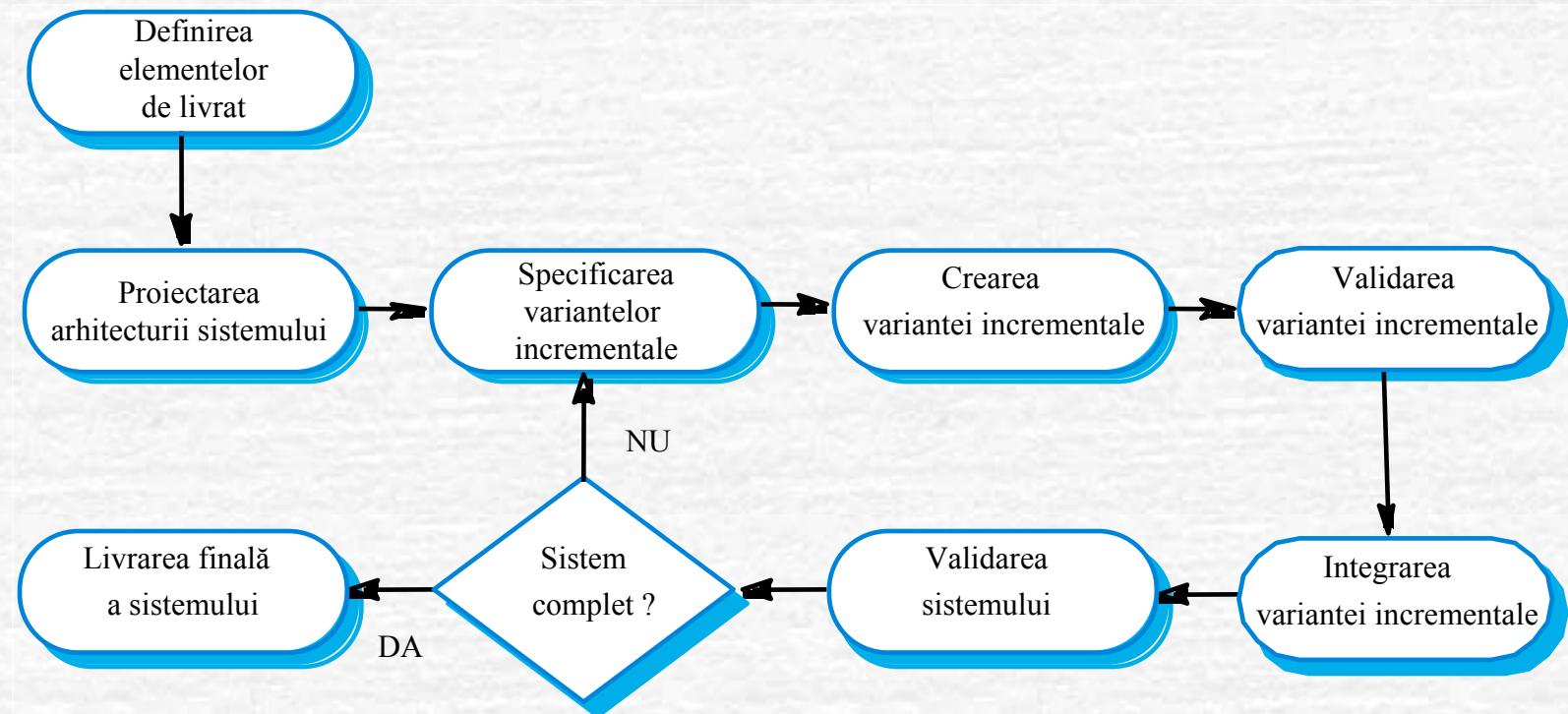
- Ajungerea la un set de cerințe de sistem consistente și stabile este deseori imposibilă din cauza mediului în schimbare
- Ca atare un model de dezvoltare în cascadă este nepractic în această situație
- Cel mai potrivit este un model de dezvoltare rapidă bazat pe specificații și livrare iterative

# Caracteristici ale unui proces RAD

---

- Procesul de specificare, proiectare și implementare este un proces concurrent. Nu există o specificație detaliată și documentarea proiectării este minimizată
- Sistemul este dezvoltat într-o serie incrementală. Utilizatorii finali evaluatează fiecare variantă incrementală și fac propuneri pentru următoarea variantă
- Interfețele utilizator ale sistemului sunt dezvoltate de obicei într-un proces interactiv

# Proces de dezvoltare iterativ



# Avantajele dezvoltării incrementale

---

- Livrarea accelerată de servicii clientilor. Fiecare variantă incrementală livrează cea mai prioritată funcționalitate către client
- Implicarea utilizatorului vizavi de sistem. Utilizatorul trebuie să se implice în dezvoltare ceea ce duce la un sistem care se mulează mai bine pe cerințele sale

# Problemele dezvoltării incrementale

---

## Probleme de management.

- Greu de judecat progresul dezvoltării. Problemele sunt greu de descoperit deoarece nu există documentație care să evidențieze exact ce s-a realizat

## Probleme contractuale

- Un contract obișnuit include specificații. În cazul acesta forma contractului nu va putea să includă specificații (greu de estimat un buget)

## Probleme la validare

- În lipsa specificațiilor, cum se testează sistemul?

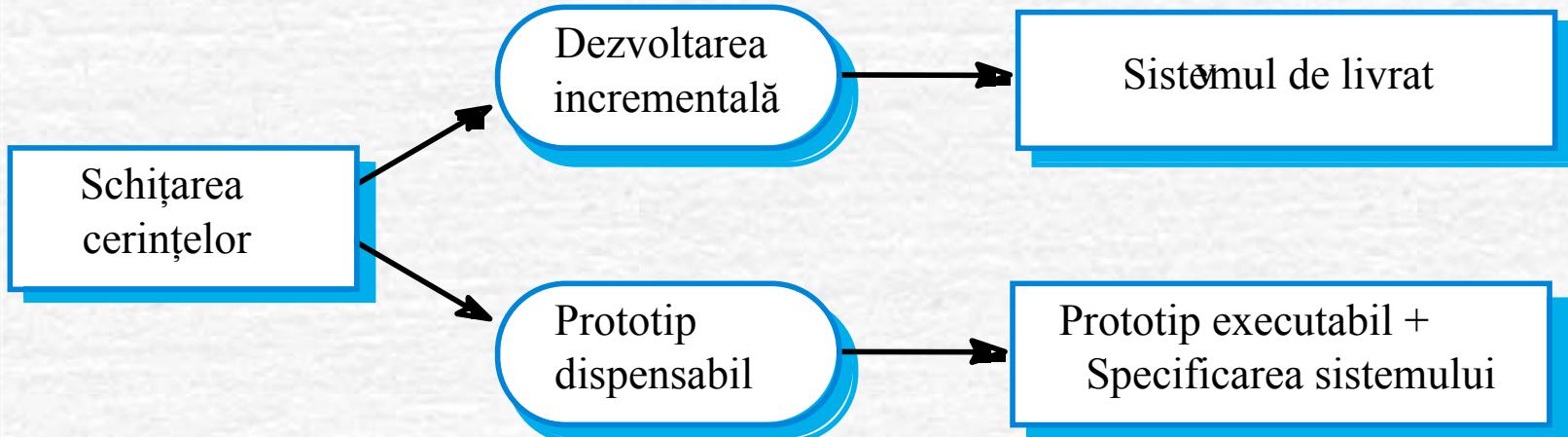
## Probleme de menenanță

- Schimbările continue tind să corupă structura sistemului ceea ce duce la costuri crescute la schimbări și evoluții necesare pentru satisfacerea noilor cerințe

# Prototipizarea

- În cazul unor sisteme mari, dezvoltarea iterativă incrementală poate fi nepractică, în special când mai multe echipe lucrează în locații diferite
- Prototipizarea se referă la dezvoltarea unui sistem experimental ce poate fi utilizat ca bază pentru formularea cerințelor. După ce se ajunge la o înțelegere, se poate renunța la acest sistem

# Dezvoltarea Incrementală și Prototipizarea



## Obiective conflictuale

- Obiectivul dezvoltării incrementale este livrarea unui sistem funcțional utilizatorilor finali. Dezvoltarea începe cu acele cerințe ce sunt cele mai bine înțelese
- Obiectivul prototipului dispensabil este acela de a valida sau furniza cerințele sistemului. Procesul de prototipizare începe cu acele cerințe care sunt mai neclare

# Metode “Agile”

- ➊ Metodele “Agile” au apărut datorită modului greoi characteristic metodelor clasice de proiectare. Acestea:
  - Se concentrează pe cod nu pe proiectare
  - Se bazează pe o dezvoltare iterativă
  - Au ca principal scop acela de a livra rapid software funcțional adaptat cerințelor în schimbare
- ➋ Metodele “Agile” sunt probabil cel mai potrivite sistemelor de dimensiuni mici/medii

# Principiile Metodelor “Agile”

## Implicitarea clientului

- Clientul trebuie implicat activ în procesul de dezvoltare. Rolul său este acela de a furniza și prioritiza cerințele, respectiv de a evalua iteratiile sistemului

## Livrare incrementală

- Software-ul este livrat în variante incrementale

## Accent pe oameni nu pe proces

- Abilitățile echipei de dezvoltare trebuie recunoscute și exploataate. Echipa trebuie lăsată să-și stabilească propria cale de a lucra la proiect

## Includerea schimbărilor

- Sistemul trebuie proiectat astfel încât să permită schimbarea cerințelor

## Menținerea simplității

- Simplitatea trebuie urmărită atât în cazul produsului cât și al procesului de dezvoltare

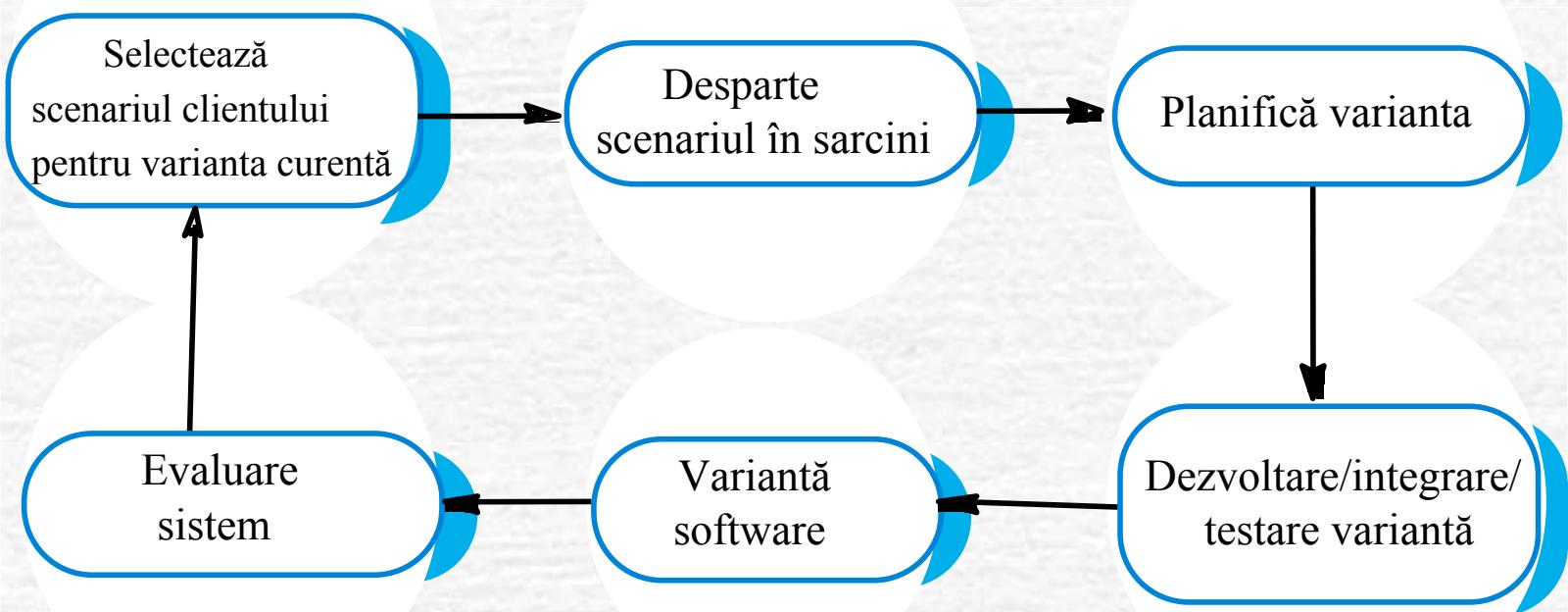
# Problemele Metodelor “Agile”

- ➊ Interesul clienților implicați în proces poate fi dificil de păstrat
- ➋ Membrii echipei pot să aibă dificultăți de adaptare la implicarea intensă care caracterizează metodele “Agile”
- ➌ Prioritizarea schimbărilor poate fi dificilă când există parteneri multiplii
- ➍ Menținerea simplității poate implica muncă în plus
- ➎ Contractele pot fi o problemă la fel ca și în cazul celorlalte abordări de dezvoltare iterativă

# Extrem Programming

- ➊ Probabil cea mai cunoscută și larg utilizată metodă “Agile”
- ➋ Extrem Programming (XP) aduce un concept ‘extrem’ dezvoltării iterative
  - Noi versiuni sunt dezvoltate chiar de câteva ori pe zi
  - Variante incrementale sunt livrate clienților la fiecare 2 săptămâni
  - Toate testările trebuie făcute pentru fiecare variantă, aceasta fiind acceptată doar dacă testele rulează cu succes

# Ciclurile de Dezvoltare XP



# Practici Extrem Programming (1)

## Planificarea incrementală

- Cerințele sunt înregistrate pe "Story Cards" și determinarea scenariului care să fie inclus într-o variantă funcție de timpul avut la dispoziție și de priorități. Dezvoltatorii despart aceste scenarii în sarcini

## Variante mici

- Setul minimal de funcționalități care esențiale pentru afacere este dezvoltat primul. Variantele sistemului sunt frecvente și adaugă funcționalități în mod incremental

## Proiectare simplă

- Proiectarea se limitează la suportul cerințelor curente

## Testare de la început

- O unitate cadru automată de testare va fi utilizată pentru scrierea testelor pentru fiecare funcționalitate nouă, înainte de implementarea acesteia

# Practici Extrem Programming (2)

## Refactorizare

- Codul va fi refactorizat continuu când se întrevăd posibile îmbunătățiri. În acest fel codul va fi menținut simplu și ușor de întreținut

## Programare în perechi

- Își vor verifica reciproc munca și se ajută pentru a obține rezultate cât mai bune

## Proprietate comună

- Proprietatea asupra codului este comună celor doi. Alt cineva nu poate interveni pentru a schimba codul

## Integrarea continuă

- Imediat ce o sarcină este completată rezultatul este integrat în sistem. După integrare trebuie trecute toate unitățile de test ale sistemului

# Principii XP și Agile

---

- ➊ Dezvoltarea incrementală este sprijinită prin variante ale sistemului mici și frecvente
- ➋ Implicarea clientului înseamnă munca full-time a acestuia în cadrul echipei
- ➌ Programarea va fi axată pe oameni nu pe procesul de producție, codul este deținut în comun și vor fi evitate orele suplimentare
- ➍ Schimbările vor fi înglobate “din mers” prin dezvoltarea de variante regulate ale sistemului
- ➎ Simplitatea va fi menținută prin refactorizarea continuă a codului

# Scenariile Cerințelor

- În XP, cerințele utilizatorilor sunt exprimate ca și scenarii sau relatări utilizator
- Acestea sunt scrise pe cartele și echipa de dezvoltare le desparte apoi în sarcini de implementat. Aceste sarcini sunt baza planificării și a estimării costurilor
- Clientii aleg scenariile care vor fi incluse în următoarea variantă pe baza priorităților de terminate de ei și a planificării

# Exemplu de “Story Card”

- Ex: aducerea din rețea a documentelor

## Aducerea și tipărirea unui articol

La început vei selecta articolul dorit din lista afișată. Apoi trebuie selectat cum vei plăti pentru el – printr-un abonament, printr-un cont al companiei sau printr-o carte de credit.

După aceasta vei primi un formular de copyright pentru a-l completa și, după ce îl trimiți documentul va fi salvat pe calculatorul tău.

Apoi vei alege o imprimantă și documentul va fi tipărit. Vei confirma apoi sistemului că documentul a fost corect tipărit.

Dacă articolul este print-only nu poți păstra o versiune PDF și ca atare documentul va fi șters automat din calculator.

# XP și Schimbările Cerințelor

- Sfatul convențional în ingineria software este să proiectezi pentru schimbare. Este util de implicat timp și efort pentru anticiparea schimbărilor pentru că aşa se reduc costurile ulterioare pe durata ciclului de viață
- Cu toate acestea, XP susține că nu este un lucru prea util cât timp schimbările nu pot fi anticipate exact
- Propune în schimb îmbunătățirea constantă a codului (refactorizare) pentru a face schimbările mai ușoare când vor trebui implementate

# Testarea în XP

- ➊ Dezvoltarea “test-first”
- ➋ Teste dezvoltate incremental din scenarii
- ➌ Implicarea utilizatorului în testare și validare
- ➍ Teste automate utilizate pentru a gestiona testarea tuturor componentelor de fiecare dată când o nouă variantă este creată

# Exemplu de “Task Cards”

**Task 1: Implementarea fluxului principal**

**Task 2: Implementarea catalogului de articole și a selecției**

**Task 3: Implementarea plășii**

Plata poate și făcută în trei moduri diferite. Utilizatorul selectează în ce mod dorește să plătească. Dacă utilizatorul are un abonament la bibliotecă poate introduce numărul de abonament care va fi verificat de sistem. Ca și alternativă el poate introduce un număr de cont al companie. Dacă acesta este valid se va înregistra plata în respectivul cont. A treia alternativă este introducerea unui număr de carte de credit și a datei de expirare a acesteia, urmată de verificare și înregistrarea plășii în contul asociat.

# Descrierea unui caz de test

## **Test 4: Testarea validității cărții de credit**

### **Intrare:**

Un sir reprezentând numărul cărții de credit și doi întregi reprezentând luna și anul expirării ei.

### **Teste:**

Verifică dacă toate caracterele din sir sunt cifre.

Verifică dacă luna este în intervalul 1-12 și anul este mai mare sau egal cu anul curent.

Utilizând primele patru cifre ale cărții de credit se verifică dacă entitatea emitentă este validă prin verificarea tabelei cu entitățile valide.

Verifică validitatea cărții de credit prin trimiterea informațiilor despre ea unității emitente.

### **Ieșire:**

OK sau mesaj de eroare indicând invaliditatea cărții de credit.

## Dezvoltare "Test-first"

---

- Teste sunt scrise înainte de implementarea în cod a cerințelor
- Testele sunt scrise ca și programe (nu ca seturi de date de test) astfel încât să poată fi executate automat. Acestea vor furniza automat date de intrare către program și vor testa rezultatele obținute
- Toate testele noi și existente sunt rulate automat când sunt adăugate noi funcționalități. În acest fel se testează că noile funcționalități nu au introdus erori

# Programarea în perechi

- În XP programatorii lucrează în perechi, dezvoltând codul împreună
- Acest fapt ajută răspândirea cunoștințelor în cadrul echipei și responsabilitatea comună asupra codului
- Servește pe post de proces de recenzie implicit deoarece fiecare linie de cod este privită de mai mult de o persoană
- Încurajează refactorizarea cât timp întreaga echipă beneficiază de aceasta
- Evaluările sugerează că productivitatea programării în perechi este similară cu cea a programării individuale pentru cei doi programatori

# RAD

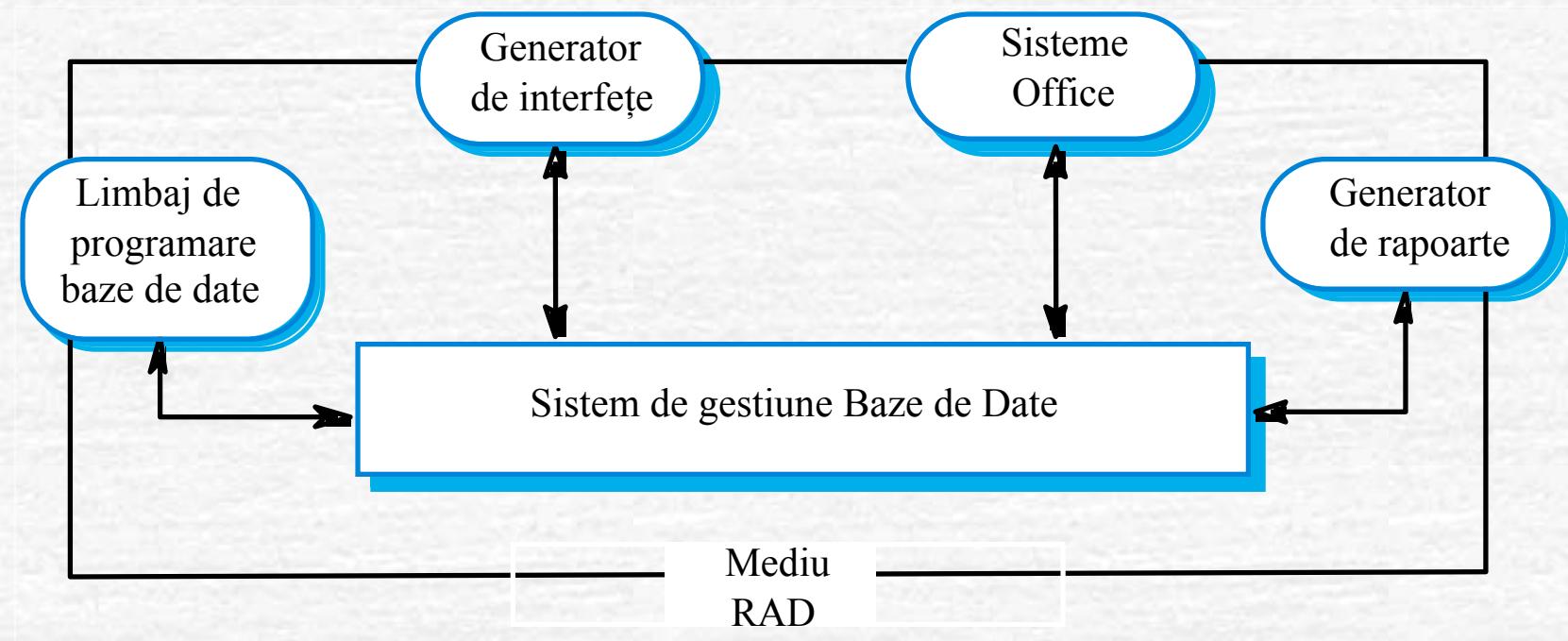
- RAD = Rapid Application Development
- Metodele Agile sunt în centrul atenției dar și alte tehnici RAD au fost aplicate cu succes de mai mulți ani
- Acestea sunt proiectate pentru aplicații de afaceri care prelucrează multe date și constau în procesarea și prezentarea informațiilor dintr-o bază de date

# Componentele unui Mediu RAD

---

- ➊ Limbaj de programare pentru baze de date
- ➋ Generator de interfețe
- ➌ Legături cu aplicații de birou (editoare de text, foi de calcul, editoare de grafice etc.)
- ➍ Generator de rapoarte

# Componentele unui Mediu RAD



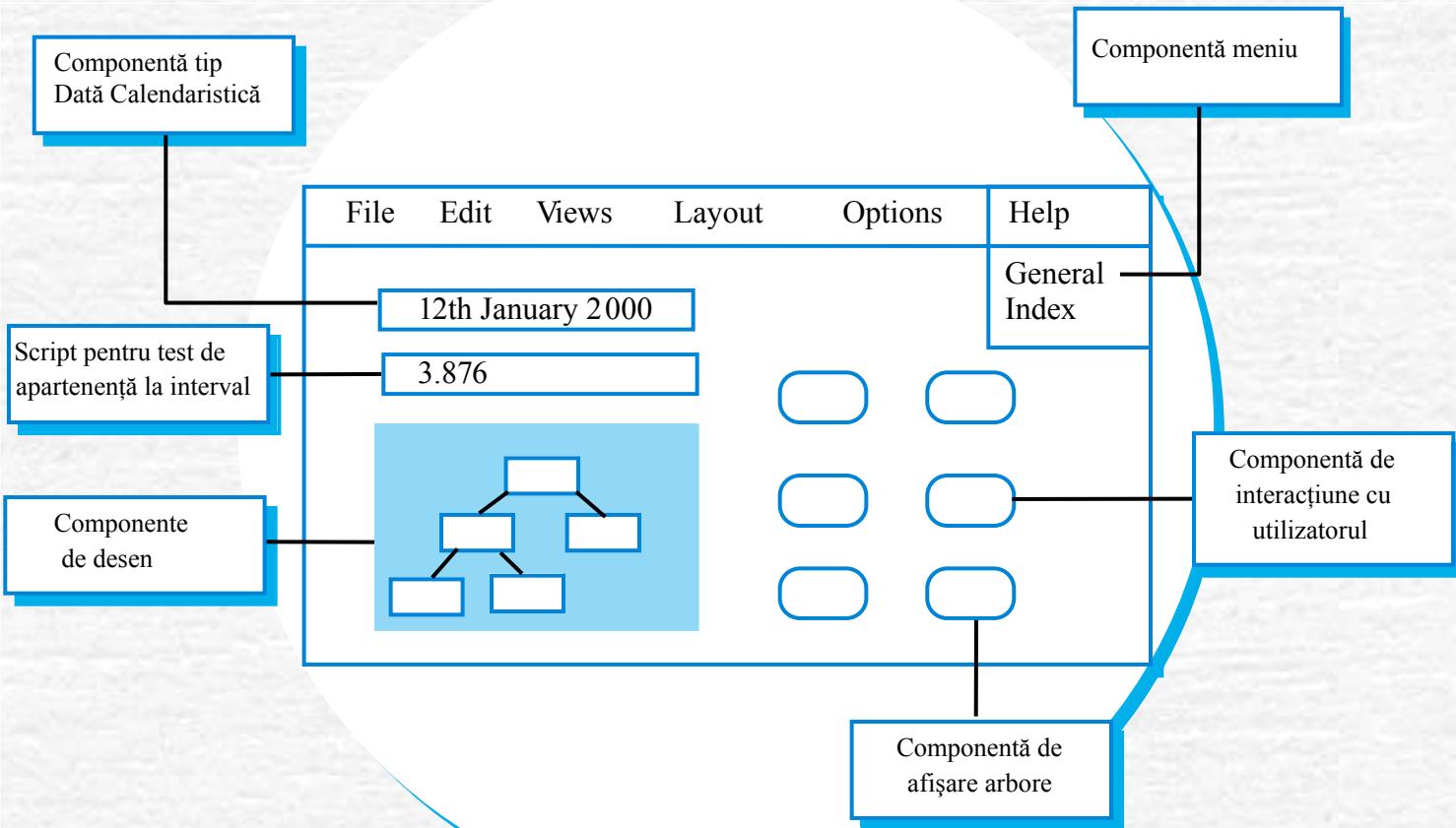
# Generarea Interfețelor

- Multe aplicații se bazează pe formulare de intrare complexe. Dezvoltarea unor astfel de formulare manual este o activitate mare consumatoare de timp
- Mediile RAD includ suport pentru generarea de ecrane de intrare, cuprinzând:
  - Definirea interactivă (cu mouse-ul) utilizând tehnici de drag-and-drop
  - Legarea formularelор pentru o secvență de formulare specificată
  - Verificarea formularelор pentru încadrarea câmpurilor în limite predefinite

# Programarea Vizuală

- Limbaje de programare tip script, de exemplu Visual Basic, suportă programarea vizuală unde prototipul este dezvoltat prin crearea interfeței utilizator pornind de la elemente standard și asociind componente cu aceste elemente
- Există o bibliotecă bogată de componente care să sprijine acest tip de dezvoltare
- Acestea pot fi adaptate să potrivească unor cerințe specifice ale aplicației

# Programarea Vizuală cu componente reutilizabile



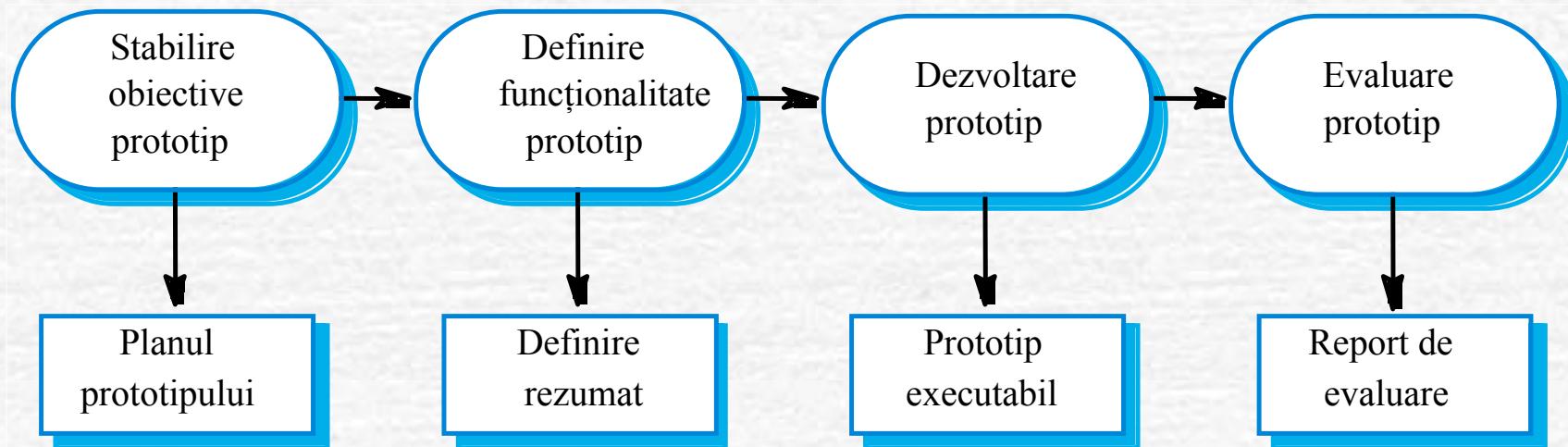
# Programarea Vizuală. Probleme

- Coordonare dificilă a echipei de dezvoltare
- Nu există o arhitectură explicită a sistemului
- Dependențele complexe între părțile de program pot cauza probleme de menenanță

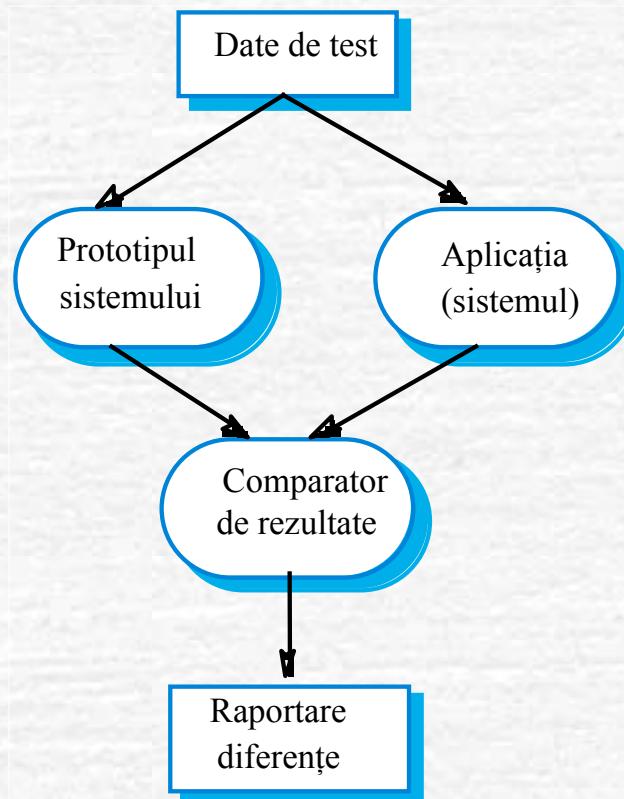
# Prototipizarea Software

- ➊ Un prototip este o versiune inițială a unui sistem utilizată pentru a demonstra concepte și a încerca opțiuni de proiectare
- ➋ Un prototip poate fi utilizat în:
  - Procesul de inginerie a cerințelor pentru a obține și valida cerințele
  - În procesul de proiectare pentru a explora opțiunile și a dezvolta interfețe utilizator
  - În procesul de testare pentru a rula testele *back-to-back* (comparație între rezultatele date de sistem și cele date de prototip pe același set de date de intrare)

# Procesul de Prototipizare



# Testarea back-to-back



# Beneficiile Prototipizării

- ➊ Îmbunătățește utilizabilitatea sistemului
- ➋ O apropiere mai mare de necesitățile reale ale utilizatorului
- ➌ Creșterea calității proiectării
- ➍ Creșterea mentenabilității
- ➎ Reducerea efortului de dezvoltare

# Utilitatea Prototipurilor. Probleme

---

- ➊ După dezvoltare în general se renunță la prototipuri, nefiind utile ca bază pentru sistemul de producție deoarece:
  - Poate fi foarte greu de ajustat sistemul pentru a satisface cerințele non-funcționale
  - În general prototipurile nu sunt documentate
  - Structura prototipului este în general degradată de schimbări rapide
  - Prototipul probabil nu va satisface cerințele standardelor de calitate ale organizației

# Reutilizare

- În majoritatea disciplinelor ingineresci, sistemele sunt proiectate prin compunerea componentelor existente, care au fost deja utilizate în alte sisteme
- Ingineria software s-a concentrat mai mult spre dezvoltarea de cod original. La momentul actual este recunoscut faptul că pentru a obține un software de bună calitate, rapid și cu costuri mici, este nevoie adoptarea unui proces de proiectare bazat pe reutilizare sistematică a software-ului

# Inginerie Software Bazată pe Reutilizare

---

## Reutilizare sisteme de aplicații

- Un întreg sistem poate fi reutilizat fie prin incorporarea fără schimbări în alte sisteme (reutilizare COTS - Commercial off-the-shelf) sau prin dezvoltarea de familii de aplicații

## Reutilizare componente

- Se pot reutiliza componentele unei aplicații, de la subsisteme până la obiecte singulare.

## Reutilizare obiecte și funcții

- Componentele software care implementează un singur obiect sau funcție bine definite pot fi de asemenea reutilizate

# Reutilizare. Beneficii

- Crește dependabilitatea (gradul de încredere)
  - Software-ul reutilizat, care a fost deja testat în sisteme funcționale, este mai de încredere decât un software nou. Utilizarea inițială scoate de obicei la lumină greșeli de proiectare sau implementare. Acestea sunt mai apoi reparate reducând astfel numărul de căderi când softul este reutilizat
- Risc redus al procesului de producție
  - Incertitudinea legată de costuri la reutilizare este mai mică decât la dezvoltare. Acest lucru determină reducerea marjei de erori la estimarea costului proiectului, mai ales în situația reutilizării de componente mari (ex. sub-sisteme)
- Utilizarea eficientă a specialiștilor
  - În loc de a face același lucru la mai multe proiecte, specialiștii vor dezvolta software reutilizabil care să încapsuleze cunoștințele lor

# Reutilizare. Beneficii

## Alinierea la standarde

- Anumite standarde, precum cele legate de interfețele utilizator, pot fi implementate ca și componente reutilizabile standard. În acest fel toate aplicațiile vor arăta și funcționa la fel din punct de vedere al meniurilor, ferestrelor, butoanelor etc. Ca atare scade probabilitatea greșelilor utilizatorilor și timpul de familiarizare cu o nouă interfață

## Accelerarea dezvoltării

- Lansarea pe piață cât mai repede posibil este de multe ori mai importantă decât toate costurile de dezvoltare. Prin reutilizare se poate micșora și timpul de dezvoltare și cel de testare și validare al sistemului

# Reutilizare. Probleme

## Creșterea costului de menenanță

- Dacă codul sursă a componentelor reutilizate nu este disponibil costul de menenanță poate crește datorită incompatibilității componentelor cu schimbările ce apar

## Suportul scăzut în uneltele de dezvoltare

- Nu toate uneltele CASE suportă dezvoltarea cu reutilizarea componentelor. Poate fi foarte dificil sau chiar imposibil de integrat o bibliotecă de componente în aceste unelte.

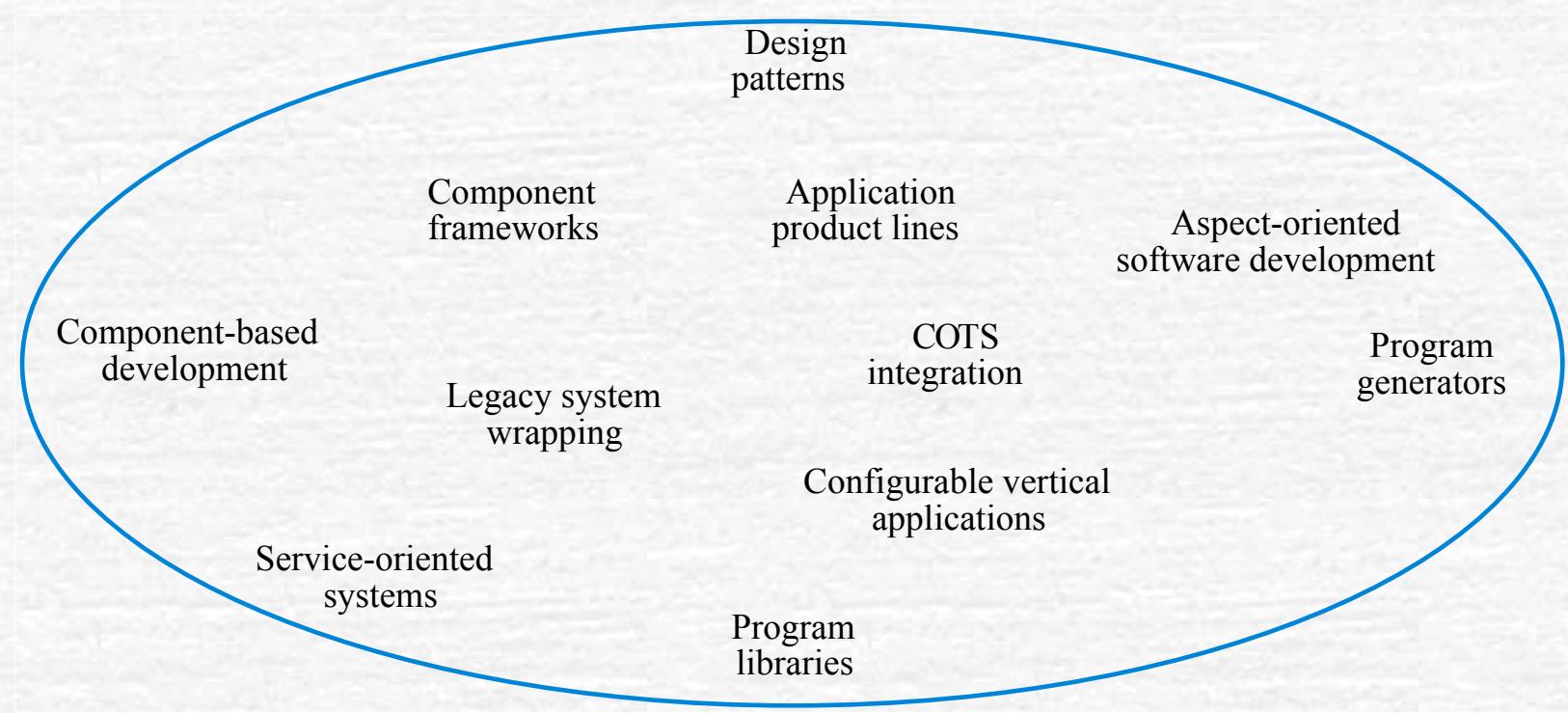
## Sindromul neîncrederii

- Unii ingineri software preferă să rescrie cod în parte pentru că aşa se poate îmbunătăți funcționalitatea, în parte pentru că e mai interesant decât utilizarea unor componente existente

## Înțelegerea și adaptarea componentelor

- Componentele trebuie descoperite în biblioteci, înțelese și adaptate

# Posibilități de Reutilizare



# Posibilități de Reutilizare

## *Design Patterns*

- Reutilizarea cunoștințelor abstracte despre probleme comune multor aplicații și soluțiile cele mai potrivite ale acestora

## *Component-based Development*

- Sistemele sunt dezvoltate prin integrarea de componente conform cu standardele unui model de compozitie

## *Application Frameworks*

- Colecții de clase abstracte și concrete care pot fi adaptate și extinse pentru a crea un sistem

## *Legacy System Wrapping*

- Sistemele moștenite pot fi înglobate prin definirea unui set de interfețe care să asigure accesul la aceste sisteme

# Posibilități de Reutilizare

## *Application Product Lines*

- Un tip de aplicație este generalizat în jurul unei arhitecturi comune astfel încât să poată fi adaptată în diverse forme pentru a satisface diferiți clienți

## *COTS Integration*

- Sistemele sunt dezvoltate prin integrarea sistemelor de aplicații existente

## *Configurable Vertical Applications*

- Este proiectat un sistem generic astfel încât să poată fi reconfigurat pentru cerințele diversilor clienți

## *Program Libraries*

- Clase și biblioteci de funcții sunt create spre a fi reutilizate

## *Program Generators*

- Pot genera sisteme sau fragmente de sisteme într-un anumit domeniu

## *Aspect Oriented Software Development*

- Asigură injectarea unor funcționalități în cod în momentul compilării

# Design Patterns

## • Nume

- Un nume sugestiv pentru identificarea tiparului

## • Descriere tipar

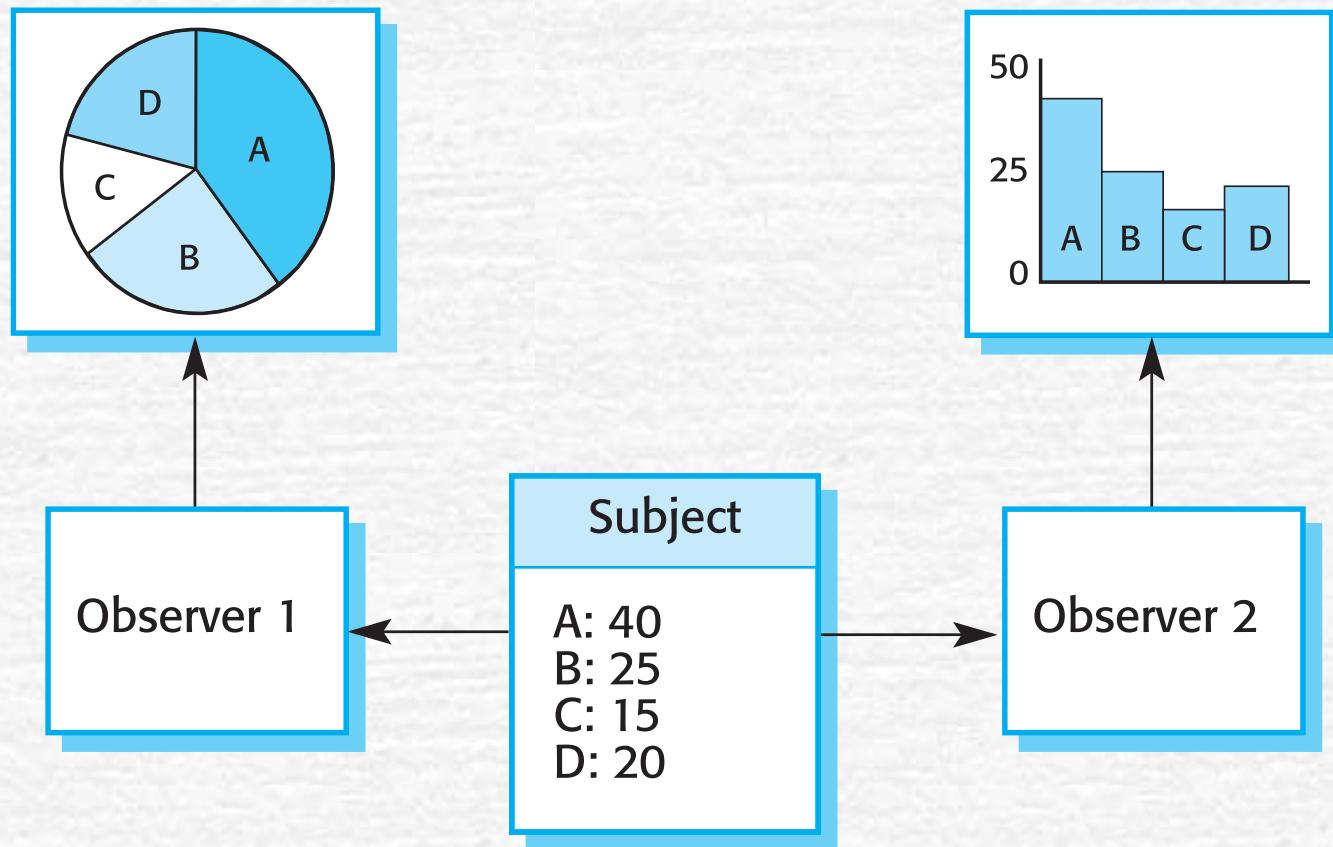
## • Descriere soluție

- Nu un proiect concret ci o machetă pentru soluția de proiectare care poate fi instanțiată în diverse forme

## • Consecințe

- Rezultatul și părțile negative ale aplicării tiparului

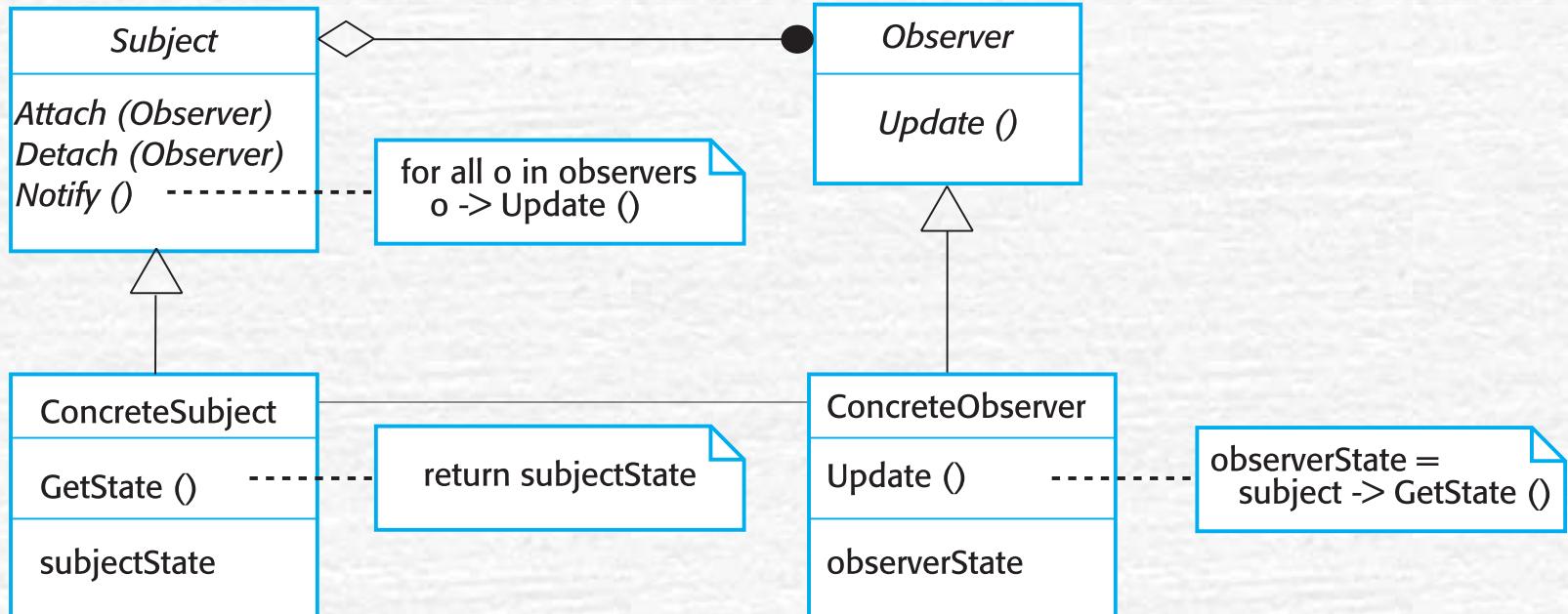
# Exemplu: Tiparul Observer



# Tiparul Observer

- Nume
  - Observer
- Descriere
  - Separă afişarea stării obiectului de obiectul însăşi
- Descrierea problemei
  - Utilizat când afişări diverse ale stării sunt necesare pentru un obiect
- Descrierea soluţiei
  - Este dată în continuare folosind UML
- Consecinţe
  - Optimizări pentru creşterea performanţei sunt nepractice

# Tiparul Observer. Descrierea soluției



# Cadre pentru Aplicații

- Cadrele (frameworks) sunt proiecte de sub-sisteme create din colecții de clase abstracte și concrete și interfețele dintre ele
- Sub-sistemul este implementat prin adăugarea componentelor pentru a umple părți din proiect și prin instanțierea claselor abstracte din cadru
- Cadrele sunt entități de mărimi moderate ce pot fi utilizate în diverse situații

# Clasele unui Cadru

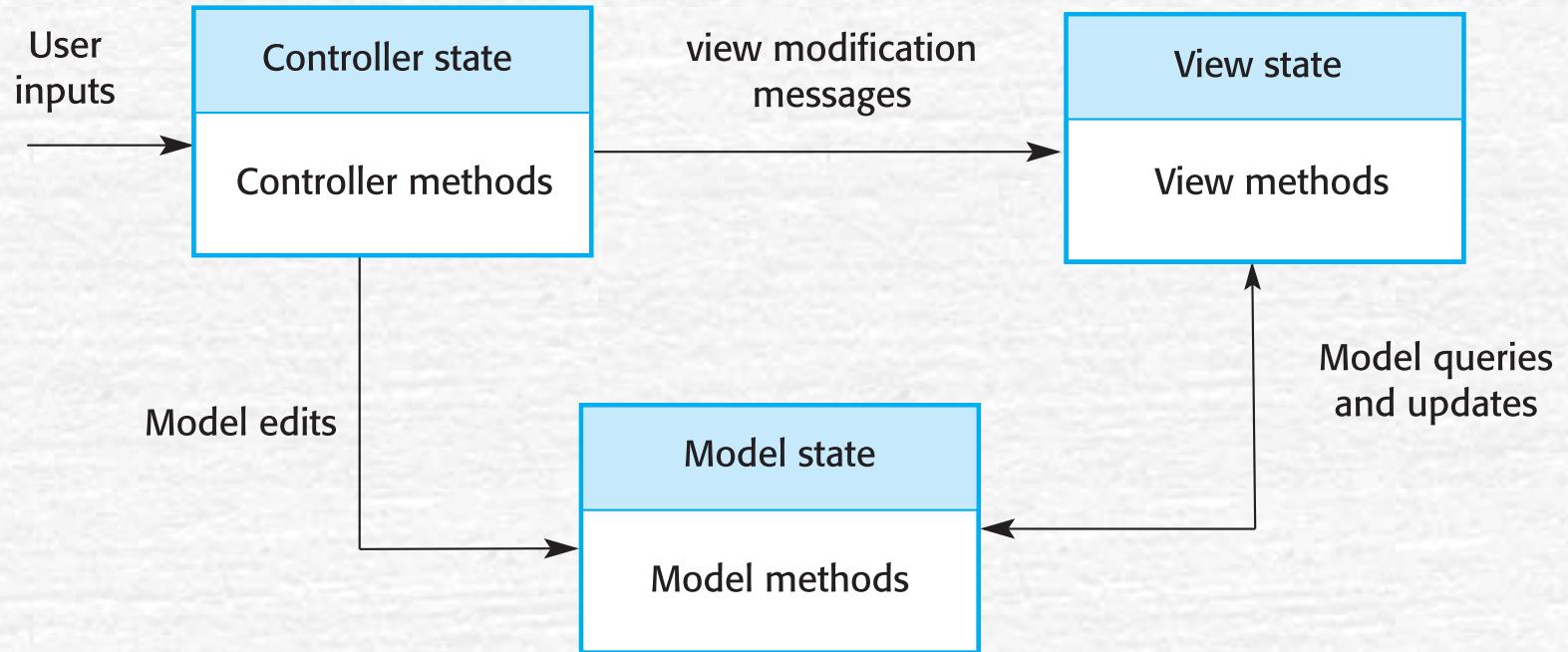
---

- ➊ Cadre pentru infrastructura sistemului
  - Sprijină dezvoltarea de infrastructuri pentru sisteme precum cele de comunicații, interfețe utilizator sau compilatoare
- ➋ Cadre pentru integrare middleware
  - Standarde și clase care sprijină comunicarea între componente și schimbul de informație
- ➌ Cadre pentru aplicații Enterprise
  - Sprijină dezvoltarea unor tipuri de aplicații specifice precum cele de telecomunicații sau sistemele financiare

## Exemplu: Model-View Controller

- ➊ Cadru de infrastructură pentru proiectarea sistemelor cu interfață grafică
- ➋ Permite prezentări multiple ale unui obiect și separarea interacțiunilor între aceste prezentări
- ➌ Cadrul MVC implică instanțierea mai multor design pattern-uri

# Model-View Controller



# Concluzii

---

- O abordare iterativă a dezvoltării software conduce la o livrare rapidă a aplicației
- Metodele Agile sunt metode iterative care urmăresc reducerea supra-încărcării la dezvoltare permitând producerea rapidă de software
- Programarea XP include practici precum: testarea sistematică, îmbunătățirea continuă și implicarea beneficiarului
- Puterea abordării testării în XP stă în testele executabile dezvoltate înainte de scrierea codului

## Concluzii

- ➊ Mediile RAD includ: limbaje de programare pentru baze de date, generatoare de formulare și legături către aplicații de birou
- ➋ Un prototip dispensabil este utilizat pentru a explora cerințele și opțiunile de proiectare
- ➌ Implementarea unui prototip dispensabil pornește cu cerințele cel mai puțin înțelese. În contrast, la dezvoltarea incrementală se pornește cu cerințele cele mai clare

# Concluzii

---

- ➊ Avantajele reutilizării sunt: costuri scăzute, accelerarea dezvoltării software și micșorarea riscului
- ➋ *Design Patterns* sunt abstracții de nivel înalt care descriu soluții de proiectare de succes
- ➌ Generatoarele de cod sunt un sprijin important pentru reutilizarea codului, cuprinzând seturi de concepte reutilizabile
- ➍ *Application Frameworks* sunt colecții de obiecte abstracte și concrete proiectate pentru a fi reutilizate prin specializare



# Fundamente de Inginerie Software

## Cap. 7

### Testare și Validare.

**Conf.Dr.Ing. Dan Pescaru**

Textbooks: Sommerville "Software Engineering 7", 2004, Cap. 22, 23

Maciaszek "Practical Software Engineering", 2005, Cap. 12

Sursă: <http://www.comp.lanes.ac.uk/computing/resources/IanS/SE7/>

<http://www.comp.mq.edu.au/books/pse/>

**2009**



# Verificare / Validare

## Verificare:

“Construim (în mod) corect produsul?”

## Software-ul trebuie să fie conform cu specificațiile

## Validare:

“Construim produsul corect (care trebuie)?”

## Software-ul trebuie să facă ceea ce utilizatorul are nevoie

# Procesul de Verificare și Validare

---

- ➊ Procesul se desfășoară pe întreaga durată de viață a sistemului – verificarea și validarea trebuie aplicate fiecărei etape a procesului software
- ➋ Are două obiective principale:
  - Descoperirea defectelor din sistem
  - Evaluarea măsurii în care sistemul este util și utilizabil într-o situație operațională

# Scopul Verificării și Validării

---

- Verificarea și validarea trebuie să stabilească gradul în care software-ul se potrivește scopului său
- Acest lucru nu înseamnă în mod necesar lipsa completă a defectelor
- Mai degrabă, el trebuie să fie suficient de bun pentru scopul în care a fost creat, iar felul utilizării va determina gradul de încredere necesar

# Gradul de Încredere

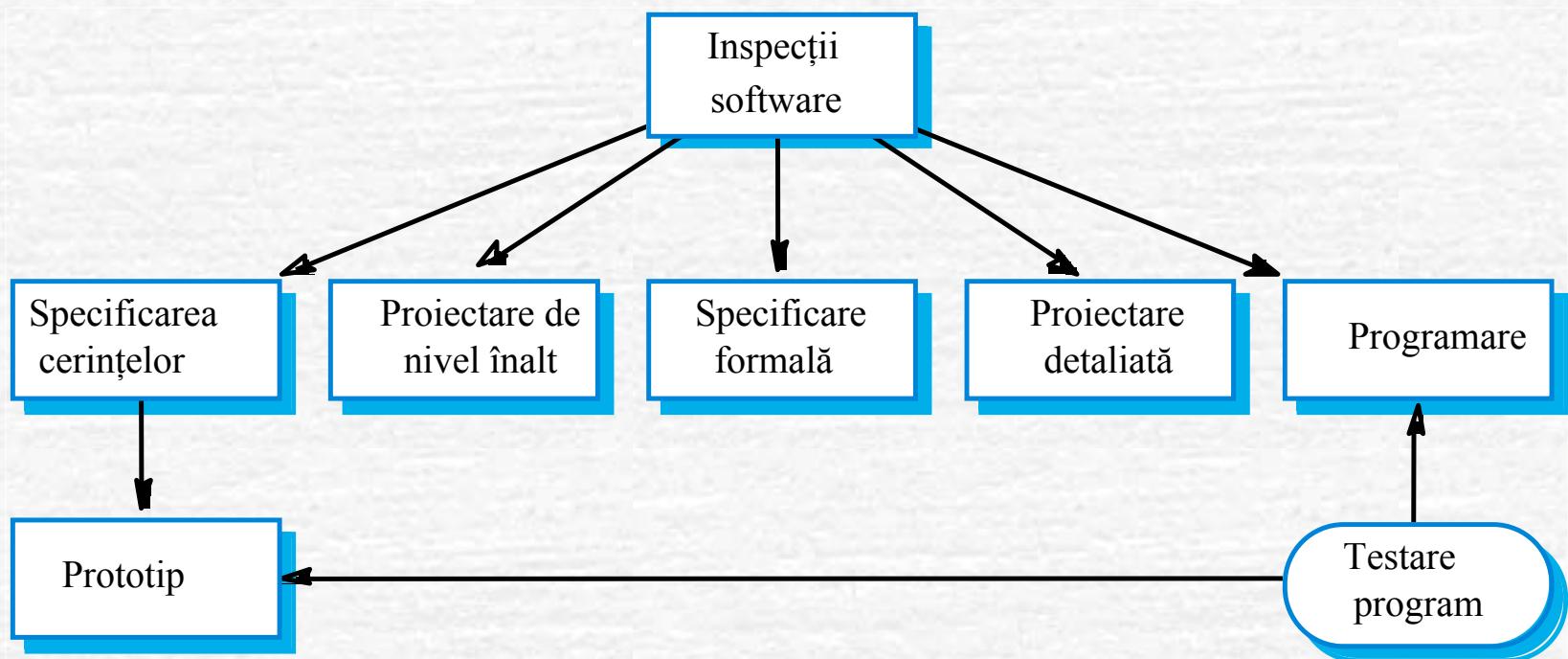
---

- ➊ Depinde de scopul sistemului, așteptările utilizatorilor și condițiilor de piață
  - Funcția software-ului în companie
    - Nivelul încrederei depinde de cât de critic este sistemul pentru organizație
  - Așteptările utilizatorilor
    - Utilizatorii pot avea așteptări nu foarte ridicate de la anumite tipuri de software
  - Condițiile de piață
    - Livrarea rapidă a unui produs software pe piață poate fi mai importantă decât găsirea tuturor defectiunilor din program

# Verificarea Statică și Dinamică

- **Inspectarea codului.** Consta în analiza statică a reprezentării sistemului pentru descoperirea eventualelor probleme (verificare statică)
  - Poate și suplimentată cu analiza automată a documentării și codului realizată prin unele dedicate
- **Testarea software.** Se preocupă de observarea comportamentului produsului la rulare (verificare dinamică)
  - Sistemul rulează cu date de test observându-se comportamentul operațional

# Verificarea și Validare Statică și Dinamică



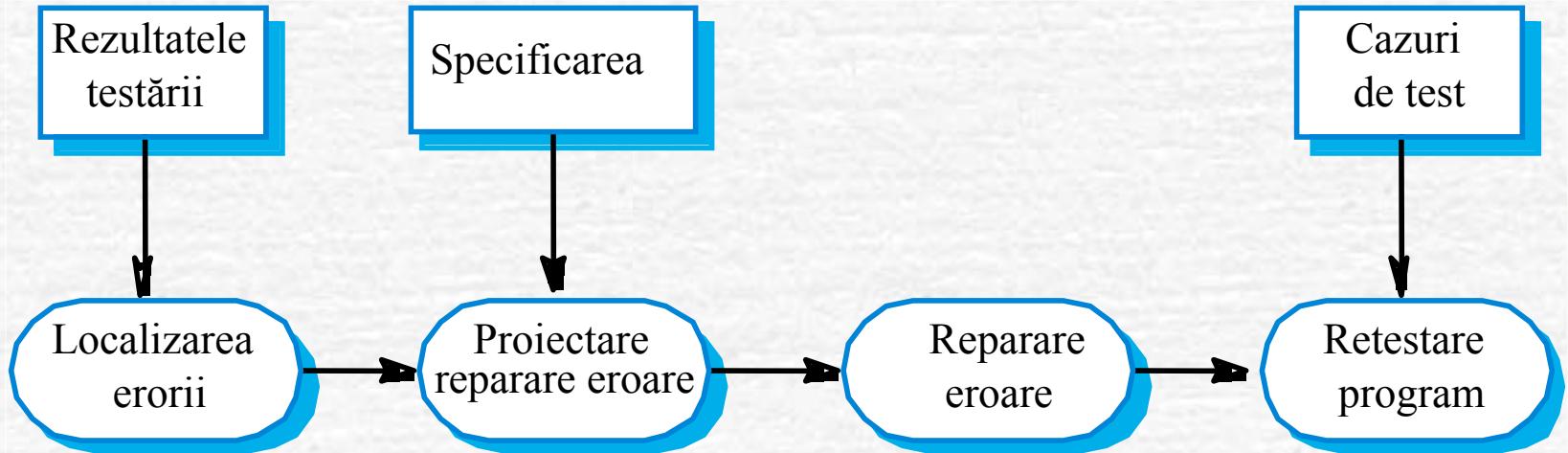
# Testarea programului

- Poate scoate în evidență prezența unor erori (NU absența acestora!)
- Singura tehnică de validare pentru cerințele non-funcționale este executarea programului și observarea comportamentului
- Trebuie utilizat în conjuncție cu verificarea statică pentru completarea procesului de verificare și validare

# Testare și Depanare

- Testarea de defecte și depanarea sunt procese distincte
- Verificarea și validarea se preocupă de stabilirea existenței defectelor în program
- Depanarea se concentrează pe localizarea și repararea erorilor
- Depanarea implică formularea unor ipoteze despre comportamentul programului care apoi sunt testate pentru a descoperi erori

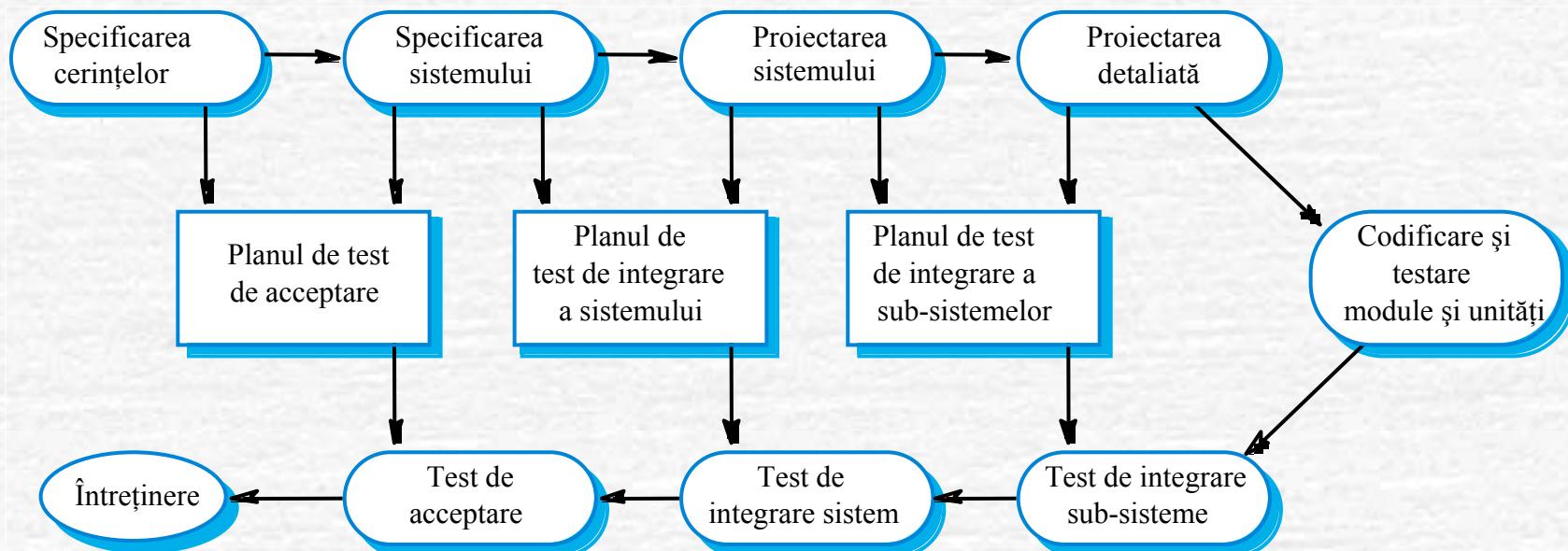
# Procesul de Depanare



# Planificarea Verificării

- ➊ Pentru a obține rezultate optime este necesară o planificare atentă a testelor și inspecțiilor
- ➋ Planificarea trebuie începută încă de la debutul procesului de dezvoltare
- ➌ Planul trebuie să identifice balansarea optimă între verificarea statică și testare
- ➍ Prin planificare se urmărește definirea unor standarde pentru procesul de testare și nu descrierea testărilor produsului

# Modelul "V" de Dezvoltare



# Structura unui Plan de Testare

- Procesul de testare. Fazele principale.
- Urmărirea cerințelor. Fiecare cerință va fi testată individual
- Elementele testate. Specificarea produselor procesului software care vor fi testate
- Planificarea testării. Se precizează și resursele alocate
- Procedurile de înregistrare a testării. Permit urmărirea corectitudinii efectuării testelor
- Cerințe hardware și software. Se precizează uneltele software utilizate pentru testare și cerințele lor
- Constrângeri. În special constrângeri legate de personalul insuficient disponibil pentru testare

# Inspectia codului

- Implică examinarea manuală a codului sursă în scopul descoperirii anomaliielor și defectelor
- Inspectiile nu necesită rularea sistemului aşa că pot fi făcute înainte de terminarea implementării
- Pot fi aplicate oricărei reprezentări a sistemului (cerințe, proiect, date de configurare, date de test etc.)
- S-a dovedit ca fiind o tehnică eficientă de descoperire a erorilor în programe

# Avantajele Inspecției

- Mai multe defecte pot fi descoperite la o singură inspecție. La testare un defect poate masca altul aşa că sunt necesare rulări succesive
- Familiaritatea cu un domeniu și cunoștințele de programare ajută recenzorii să recunoască ușor anumite tipuri de erori care apar frecvent

# Inspecția și Testarea

- Inspecția și testarea sunt tehnici de verificare complementare (și nu opuse)
- Amândouă trebuie utilizate pe timpul procesului de verificare și validare
- Inspecțiile pot verifica conformitatea cu specificațiile dar nu și cu cerințele reale ale utilizatorilor
- Inspecțiile nu pot verifica caracteristici non-funcționale precum performanța, ușurința la utilizare etc.

# Procedura de Inspecție

---

- ➊ Se începe cu o prezentare generală a sistemului echipei de inspecție
  - ➋ Codul și documentația sunt distribuite echipei încă de la început
  - ➌ Inspecția are loc și erorile descoperite sunt notate
  - ➍ Se fac modificările care să repare erorile descoperite
  - ➎ Repetarea inspecției poate fi câteodată necesară
-

# Rolul Membrilor Într-o Echipă de Inspectie

- *Autorul.* Programatorul sau proiectantul responsabil cu codul sau documentul de inspectat. Va repara defectele descoperite la inspectie
- *Inspectorul.* Găsește erorile, omisiunile și inconsistențele în cod sau document. Poate identifica și alte probleme în afara celor amintite
- *Cititorul.* Responsabil cu prezentarea codului la ședința de inspectie
- *Secretarul.* Responsabil cu notarea rezultatelor ședințelor de inspectie
- *Moderatorul.* Gestionează și facilitează inspecțiile. Raportează moderatorului șef
- *Moderatorul șef.* Responsabil cu îmbunătățirea procesului, dezvoltarea standardelor etc.

# **Lista de Inspectie**

---

- ☞ Pentru conducerea inspecției se va utiliza o listă cu erori des întâlnite
- ☞ Lista de erori este dependentă de limbajul de programare și reflectă erorile caracteristice acestuia
- ☞ În general cu cât verificările de tip sunt mai slab implementate de limbaj cu atât lista va fi mai lungă
- ☞ Exemple: inițializări, nominalizarea constantelor, cicluri infinite, indici în afara marginilor tablourilor, etc

## Ritmul Inspecției

- 500 de linii/oră în timpul privirii de ansamblu
- 125 linii de cod sursă/oră la pregătirea individuală
- 90-125 linii de cod/oră la inspectare
- Concluzie: inspecția este un proces costisitor
- Inspectia a 500 de linii de cod costa cam 40 oră-om

# Automatizarea analizei statice

- Analizoarele statice sunt ustensile software care prelucrează surse text (cod sursă).
- Ele parcurs codul programului și încearcă să descopere și să raporteze potențiale erori.
- Sunt foarte eficiente în ajutorul inspecției – totuși sunt doar un supliment și nu pot înlocui inspecția manuală.

# Exemple de teste la analiza statică

---

- Variabile utilizate înainte de initializare
- Variabile declarate dar neutilizate
- Variabile asignate de două ori dar cu prima valoare nefolosită
- Posibile depășiri de margini la indicii de tablouri
- Variabile nedeclarate
- Cod la care execuția nu poate ajunge
- Cicluri infinite
- Parametrii greșiti la apelul de funcții
- Pointeri neinitializați

# Etapele Analizei Statice

- ➊ Analiza fluxului de control. Verifică ciclurile cu intrări sau ieșiri multiple, cauță cod la care execuția nu poate ajunge etc.
- ➋ Analiza utilizării datelor. Detectează variabilele neinitializate, variabilele declarate dar neutilizate etc.
- ➌ Analiza interfețelor. Verifică consistența declarării și apelului de funcții și proceduri

# Etapele Analizei Statice

- ➊ Analiza fluxului informațional. Identifică dependențele variabilelor de ieșire. Nu detectează anomaliiile în sine dar scoate în evidență informații pentru inspectia de cod
- ➋ Analiza căilor. Identifică căile de rulare prin program și delimitează instrucțiunile executate pe acea cale. Din nou este utilă în procesul de inspecție
- ➌ Ambele etape generează o mare cantitate de informații. Ca atare trebuie utilizate cu grijă

# Utilitatea Analizei Statice

- Foarte valoroasă când limbajul utilizat este mai slab tipizat (ex. C) și ca atare multe erori trec nedetectate de compilator
- Mai puțin eficientă pentru limbaje precum Java care au o testare puternică a tipurilor și ca atare detectează multe erori în timpul compilării

# Procesul de Testare

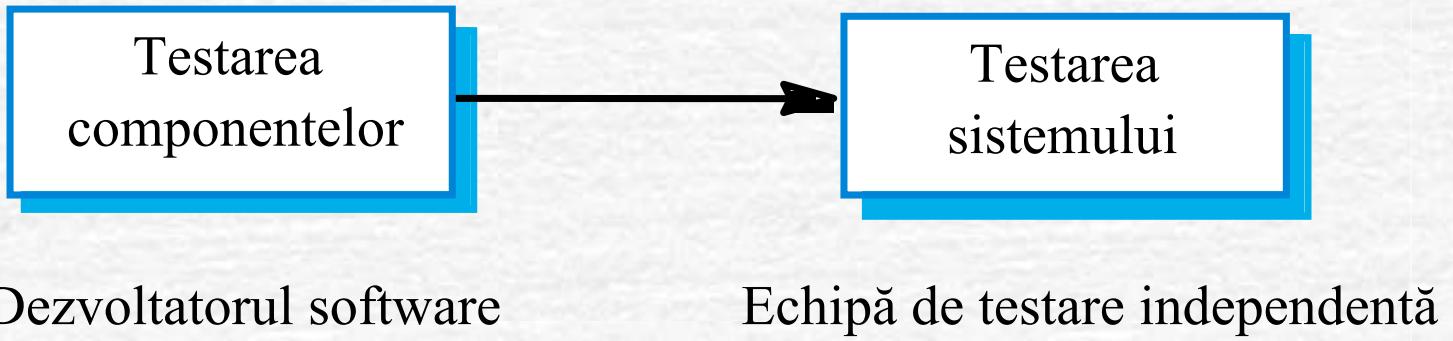
## Testarea Componentelor

- Testarea individuală a fiecărei componente a programului
- În mod obișnuit este responsabilitatea dezvoltatorului de componente (cu unele excepții la sistemele critice)
- Testele sunt derivate din experiența dezvoltatorului

## Testarea Sistemului

- Testarea grupurilor de componente integrate într-un sistem sau sub-sistem
- Responsabilitatea unei echipe de testare independente
- Testele sunt bazate pe specificarea sistemului

# Fazele Testării



# Tipuri de Testări

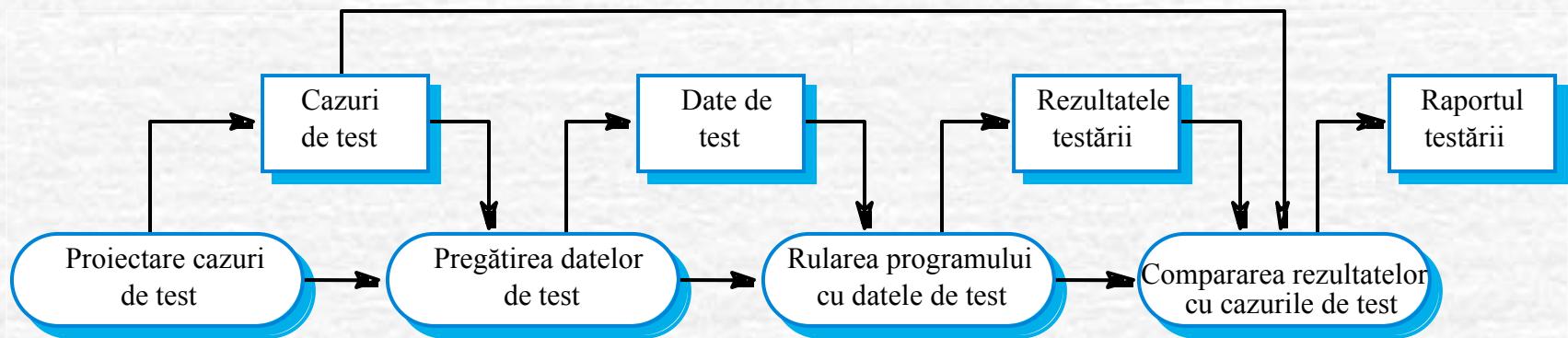
## Testarea de defecte

- Teste proiectate să descopere defectele sistemului
- Un test de defecte are succes dacă relevă prezența acestora în sistem
- Testele relevă prezența defectelor și NU absența lor

## Testarea de validare

- Are ca scop să arate că sistemul se potrivește cerințelor
- Testul se consideră că are succes dacă arată că cerințele au fost implementate corespunzător

# Procesul de Testare Software



# Politici de Testare

---

- ➊ Pentru a demonstra lipsa defectelor dintr-un program ar trebui făcută o testare exhaustivă. Cu toate acestea testarea exhaustivă este practic imposibilă.
- ➋ Politicile de testare definesc abordarea care va fi utilizată la selectarea testelor de sistem:
  - Toate funcțiile accesibile din meniu trebuie testate
  - Combinăriile de funcții accesibile din același meniu trebuie testate
  - Toate funcțiile care necesită intrări de la utilizator trebuie testate atât cu date de intrare corecte cât și incorecte

# Testarea Sistemului

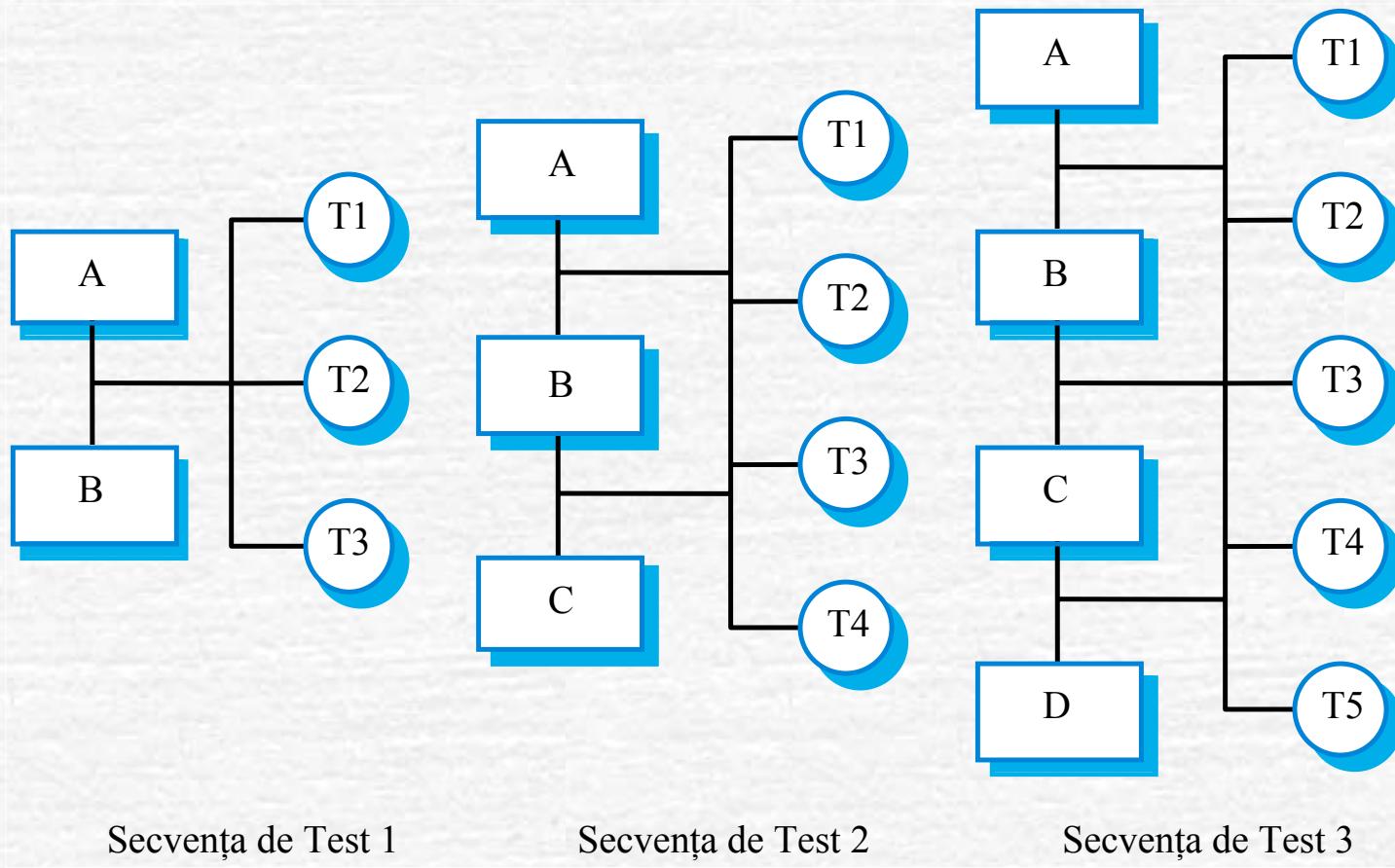
---

- Implică integrarea componentelor pentru a crea subsisteme sau sistemul respectiv
- Poate implica testarea unei variante care să fie livrate la client
- Două faze:
  - Testarea de integrare – echipa de testare are acces la codul sursă al sistemului. Sistemul este testat la nivel de componente pe măsură ce acestea sunt integrate
  - Testarea variantei finale – echipa de testare testează sistemul complet livrat ca un black-box

# Testarea de Integrare

- Implică crearea sistemului din componentele sale și testarea lui pentru a detecta problemele care apar din interacțiunea componentelor
- Integrarea top-down
  - Se dezvoltă un schelet al sistemului pe care se plasează componente
- Integrarea bottom-up
  - Se integrează mai întâi componente de infrastructură și apoi se adaugă componentele funcționale
- Pentru a simplifica localizarea erorii, sistemul trebuie integrat incremental

# Testarea cu Integrare Incrementală



# Abordări

## ✓ Validarea arhitecturii

- Testarea cu integrarea top-down este utilă la descoperirea erorilor în arhitectura sistemului

## ✓ Demonstrarea sistemului

- Testarea cu integrarea top-down permite demonstrarea limitată a funcționării sistemului încă din fazele de început

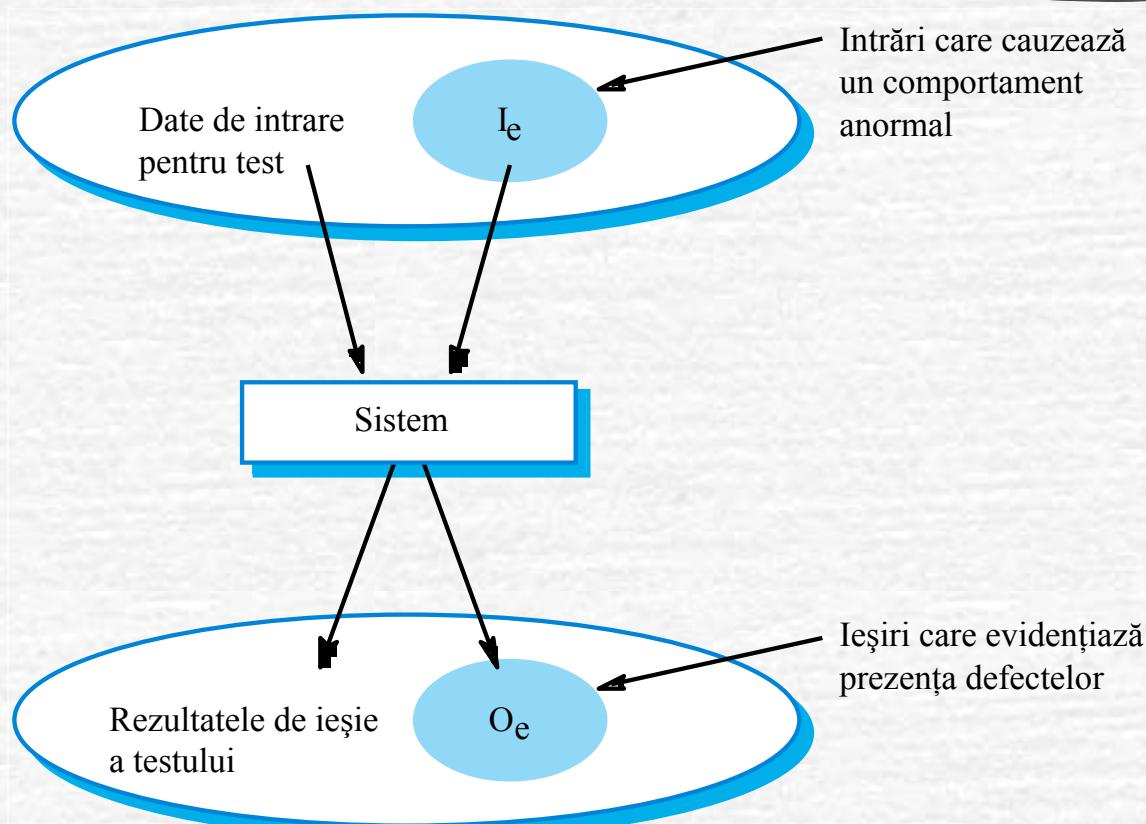
## ✓ Testarea implementării

- Deseori este mai simplă la testarea cu integrare bottom-up

## ✓ Observarea testelor

- Apar probleme în cazul ambelor tehnici. În general este nevoie de cod extern pentru a observa testele

# Testarea Black-Box



# Sfaturi Utile la Testare

---

- Aceste sfaturi sunt utile pentru echipa de testare în ai ajuta să aleagă testele care pot evidenția defecte în sistem
  - Alegerea unei intrări care forțează sistemul să genereze toate mesajele de eroare
  - Alegerea intrărilor care cauzează depășirea memoriei bufferelor
  - Repetarea aceleiași intrări sau a unei serii de intrări succesiv de câteva ori
  - Forțarea ieșirilor invalide
  - Forțarea rezultatelor calculelor să fie prea mici sau prea mari

## Teste de performanță

- O parte a testării versiunilor ce vor fi distribuite clienților implică testarea unor proprietăți emergente precum performanța sau fiabilitatea
- Testele de performanță implică în mod uzual o serie de teste la care încărcarea este mărită constant până când performanța sistemului devine inacceptabilă

## Teste de Stres

- ➊ Forțează sistemul deasupra încărcării maxime prevăzute la proiectare. Condițiile de stres scot deseori la lumină defecte ale sistemului
- ➋ În cazul unor condiții de stres sistemul ar trebui să nu dea greș în mod catastrofal. În acest fel se pot testa pierderile inacceptabile de date sau servicii
- ➌ Aceste teste sunt în deosebi relevante la sistemele distribuite care pot prezenta degradări severe dacă rețeaua devine supraîncărcată

# Testarea Componentelor

- Procesul de testare a componentelor presupune teste individuale executa asupra componentelor izolate
- Este un proces de testare la defecte
- Componentele pot fi:
  - Funcții sau metode individuale ale unui obiect
  - Clase de obiecte având câteva atribute și metode
  - Componente compuse împreună cu interfețele utilizate pentru accesarea funcționalităților lor

# Testarea Obiectelor

- ➊ Testele complete pentru o clasă implică
  - Testarea tuturor operațiilor asociate cu obiectele sale
  - Setarea și citirea tuturor atributelor unui obiect
  - Încercarea obiectului în toate stările sale posibile
- ➋ Moștenirea face mult mai dificilă proiectarea unor teste pentru clase și obiecte deoarece informația de testat nu este localizată doar la clasa respectivă

# Testarea Interfețelor

- Obiectivele sunt de a detecta erori datorate interfețelor sau presupunerilor greșite despre interfețe
- Acestea sunt importante mai ales pentru dezvoltarea orientată pe obiecte deoarece obiectele sunt definite de interfețele lor

# Tipuri de Interfețe

---

## Interfețe de parametrii

- Datele trimise de la o procedură la alta

## Interfețe cu memorie comună

- Un bloc de memorie este folosit în comun de proceduri sau funcții

## Interfețe procedurale

- Sub-sistemul înglobează un set de proceduri pentru a fi apelat de late sub-sisteme

## Interfețe cu schimb de mesaje

- Sub-sistemele solicită servicii de la alte sub-sisteme

# Erori ale Interfețelor

## Utilizare greșită a interfeței

- Apare când o componentă cheamă altă componentă și comite o eroare în utilizarea interfeței acesteia, de ex. trimite parametrii într-o ordine greșită

## Neînțelegerea interfeței

- O componentă înglobează niște presupuneri incorecte despre comportamentul componentei chemate

## Erori de sincronizare

- Componentele care interacționează operează la viteze diferite și ca atare informația accesată poate fi expirată

# Sfaturi la Testarea Interfețelor

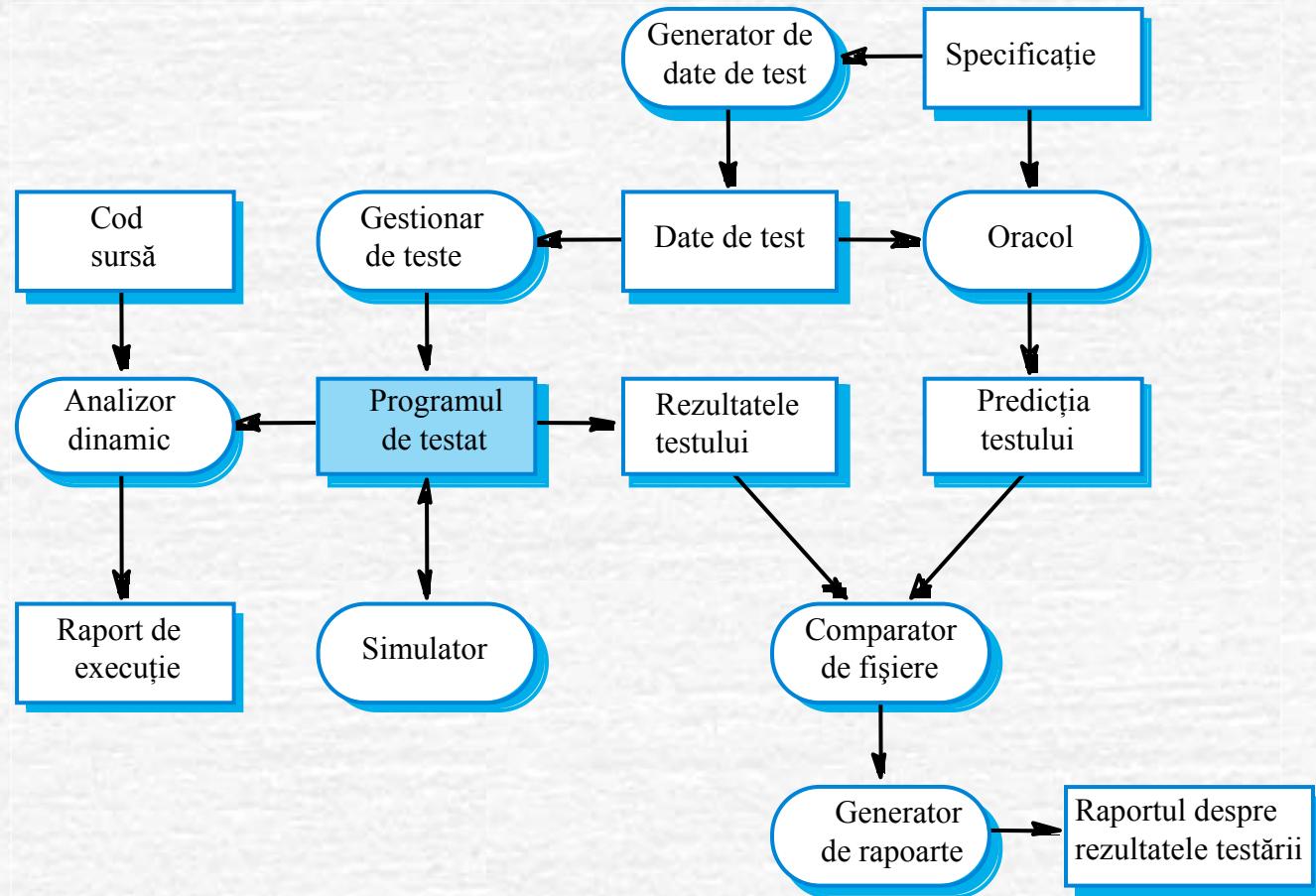
- ➊ Testele vor fi proiectate astfel încât parametrii de apel ai procedurilor să fie aleși la marginea intervalelor lor
- ➋ Parametrii pointer vor fi testați cu valori NULL
- ➌ Testele vor fi astfel proiectate ca să forțeze componenta să dea greș
- ➍ Se vor utiliza teste de stres în mecanismul de schimb de mesaje
- ➎ La sistemele cu memorie comună se va varia ordinea în care componente sunt activate

# Automatizarea Testării

---

- ➊ Testarea este o fază a procesului costisitoare. Sistemele de testare oferă o gamă variată de ustensile care reduc timpul necesar și ca atare costul total al testării
- ➋ Sisteme precum Junit sprijină execuția automată de teste
- ➌ Majoritatea sistemelor de testare sunt sisteme deschise deoarece cerințele referitoare la testare sunt specifice fiecărei organizații
- ➍ Câteodată este dificil de integrat sistemele de testare la proiectarea aplicației

# Structura unui Sistem de Testare



## Concluzii

- Verificarea și validarea nu sunt unul și același lucru.  
Verificarea arată conformitatea cu o specificație pe când validarea arată acoperirea nevoilor clienților
- Pentru a ghida procesul de testare este nevoie de un plan de testare
- Tehnicile de verificare statică implică examinarea și analiza codului programului pentru detectarea erorilor
- Testarea poate arăta prezența erorilor în sistem dar nu poate demonstra lipsa erorilor

# Concluzii

- Dezvoltatorii componentelor sunt responsabili de testarea acestora. Testarea sistemului este responsabilitatea unei echipe separate
- Testarea de integrare se ocupă cu testarea variantelor incrementale ale sistemului
- Testarea de defecte se proiectează pornind de la experiența echipei și a unor linii de ghidaj generale
- Testarea interfețelor are ca scop descoperirea defectelor interfețelor componentelor compuse



# Fundamente de Inginerie Software

## Cap. 8

### Managementul Dezvoltării Produselor Software.

**Conf.Dr.Ing. Dan Pescaru**

Textbooks: Sommerville "Software Engineering 7", 2004, Cap. 25, 26, 27

Maciaszek "Practical Software Engineering", 2005, Cap. 3

Sursă: <http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/>

<http://www.comp.mq.edu.au/books/pse/>

**2009**



# Activități de Management

---

- ➊ Managementul Proiectului. UTELTE de management specifice.
- ➋ Managementul Personalului. Managementul echipelor implicate în dezvoltarea unui proiect software.
- ➌ Estimarea Costurilor. Estimarea de preturi, costuri și a productivității procesului de dezvoltare software
- ➍ Managementul Calității. Standarde, măsurători și metrici referitoare la calitatea procesului software

# Managementul Proiectului

---

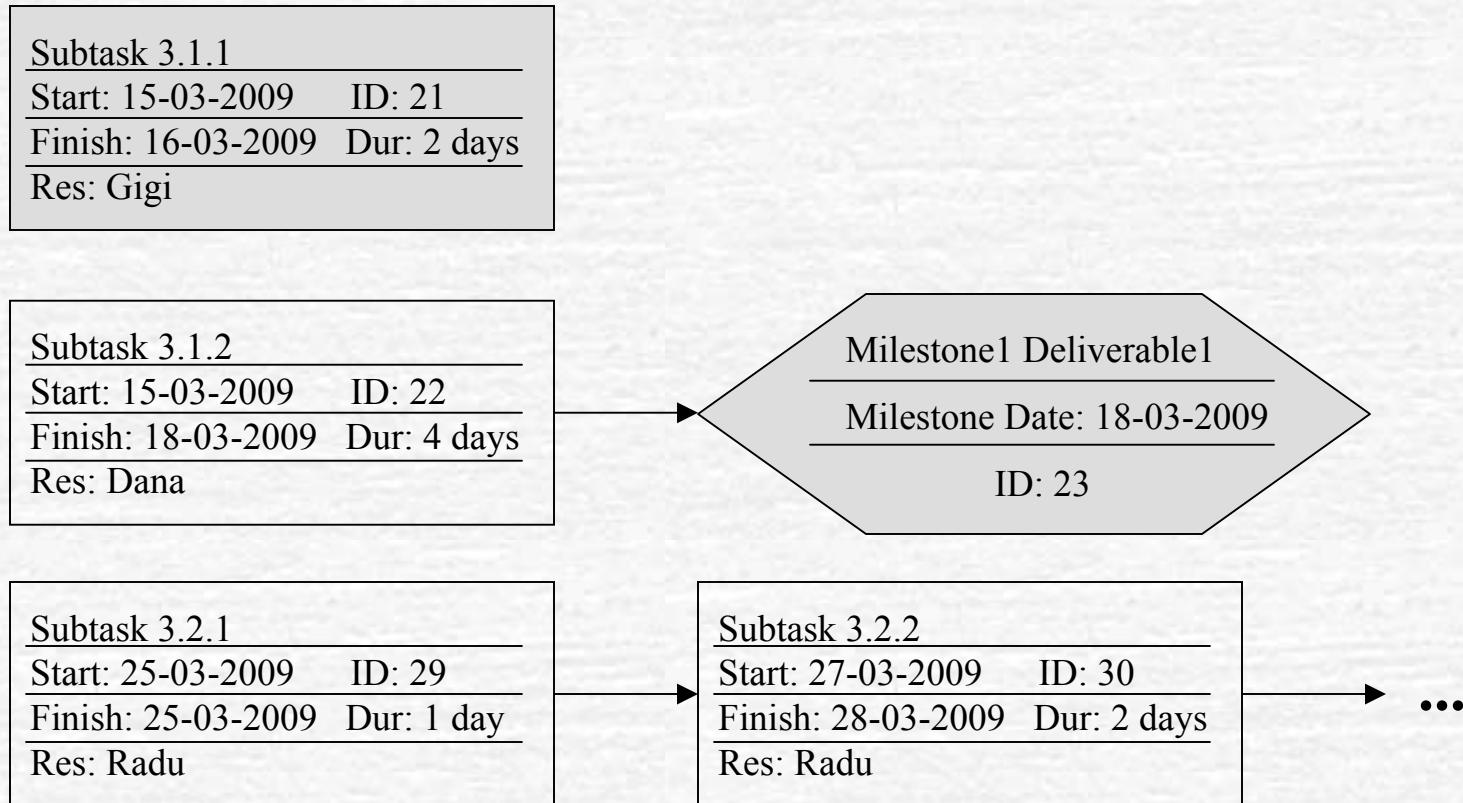
- Managementul proiectului se referă la distribuția și controlul bugetului, timpului și personalului.
- Moto: "Ce nu poți planifica nu poți nici realiza"
- Managementul eficient implică utilizarea unor unelte pentru planificarea și controlul activităților, pentru estimarea costurilor, pentru colectarea metricilor etc.
- Uneltele actuale sunt incluse frecvent în medii integrate pentru management care leagă managementul cu planificarea strategică, modelarea afacerii, gestiunea portofoliului, gestiunea documentelor, gestiunea fluxului de producție etc.

# Planificarea și Controlul Proiectului

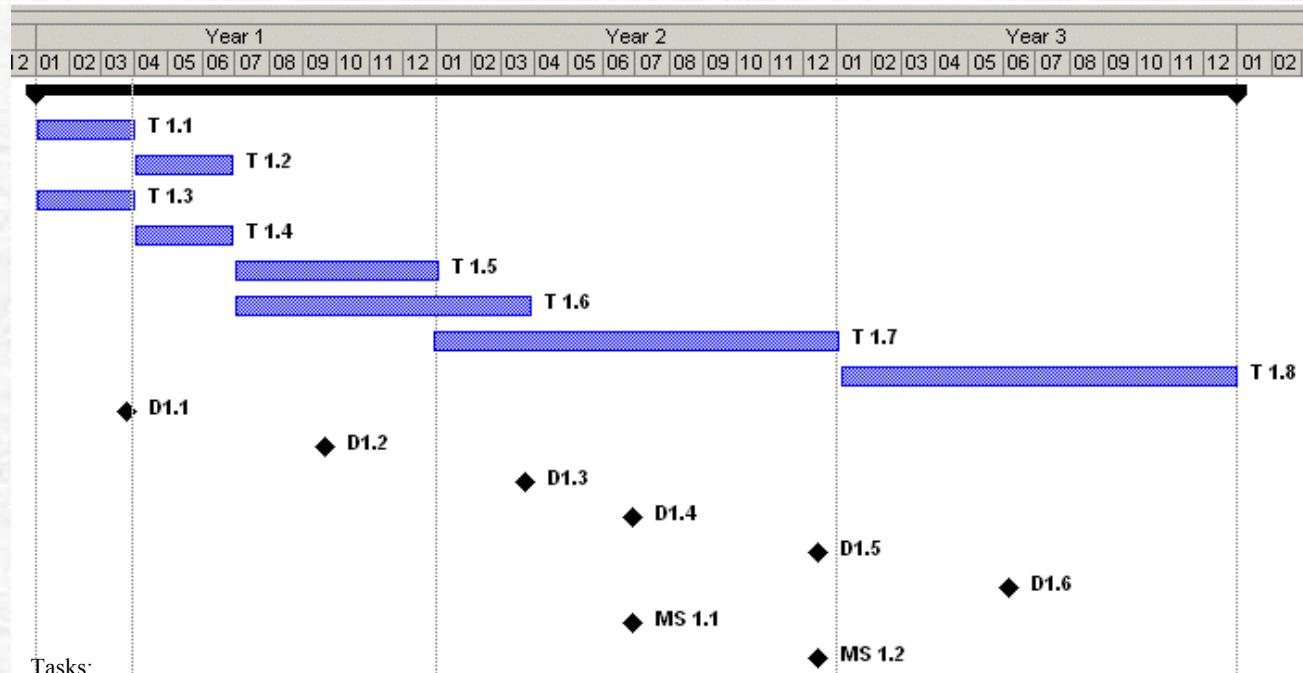
---

- *Graficele Rețelelor de Activități* – identifică evenimentele care trebuie să apară înainte ca o activitate să înceapă, specifică durata și momentul de încheiere al fiecărei activități, împarte personalul și celelalte resurse între toate activitățile etc.
  - Rețele **CPM** (Critical Path Method)
  - Diagrame **Gantt**
- Aceste grafice sunt generate de către unelte de management al proiectului furnizând informații despre proiect
- Unelte de management sunt capabile să ajusteze sau să refacă automat planificarea ori de câte ori apar schimbări la nivelul unei activități sau resurse

# Exemplu de Rețea CPM



# Exemplu de Diagramă Gantt



# Gestionarea Portofoliului

- *Gestiunea Portofoliului* este procesul care permite companiei să-și alinieze resursele și investițiile cu obiectivele strategice. În general se folosesc mediile bazate pe Web care permit:
  - Discuții, urmărirea și alocarea activităților la proiect
  - Ședințe “virtuale” între membrii aflați în diverse locații
  - Calendare și orare accesibile pentru toată echipa
  - Punerea în comun a documentelor și menținerea facilă a versiunilor de către întreaga echipă
  - Automatizarea fluxului de lucru și urmărirea deciziilor
  - Planificare prin diagrame Gantt și CPM
  - Machete de document comune pentru echipă
  - Alinierea proiectelor cu planurile strategice și modelele afacerilor
  - Gestionarea și alocarea resurselor individuale și comune
  - Generarea de rapoarte și urmărirea stării proiectului

# Utilizare Metriici În Managementul Proiectului

---

- ➊ În management, *Metricile* se referă la măsurarea procesului de dezvoltare software prin colectarea de informații care să ajute planificarea viitoare
- ➋ Proiectarea arhitecturală trebuie susținută prin metrici care să asigure înțelegerea, mentenabilitatea și scalabilitatea sistemului
- ➌ Unelte de colectare a metricilor trebuie să fie ușor de utilizat
- ➍ Analizele ***what-if*** permit identificarea dependențelor între o entitate (clasă) și restul entităților din program

# Managementul Personalului

---

- ➊ Cea mai importantă resursă a unei organizații sunt oamenii
- ➋ Principala preocupare a unui manager se referă la oamenii din organizație. Nu poate exista un management de succes dacă nevoile acestora nu sunt înțelese
- ➌ Un management slab reprezintă o cale sigură către falimentul proiectului

# Factorii care Influențează Managementul Personalului

---

## Consistență

- Membrii echipei trebuie tratați toți în aceeași manieră, fără favoruri și discriminări

## Respectul

- Diversi membri ai echipei au talente diferite iar aceste diferențe trebuie respectate

## Includerea

- Toți membrii echipei trebuie implicați și toate punctele de vedere trebuie luate în considerare

## Onestitatea

- Lucrurile care merg bine sau rău într-un proiect trebuie recunoscute cu onestitate de către managerul de proiect

# Selectia Personalului

- O sarcină importantă de management este aceea de a selecta personalul
- Informațiile utilizate la selecție provin din:
  - Informațiile prezentate de candidați
  - Informațiile obținute prin interviuri și discuții cu candidații
  - Recomandări și comentarii de la alți oameni care îi cunosc sau care au lucrat cu candidații

# Observații

---

- ➊ Managerii dintr-o companie nu doresc să-și piardă oamenii pentru un nou proiect. Lucru part-time la proiecte este inevitabil
- ➋ Abilități precum proiectarea interfeței utilizator sau interfațarea componentelor hardware sunt utile dar nu și suficiente
- ➌ Proaspeții absolvenți pot să nu aibă cunoștințe specifice dar reprezintă o cale de introducere a noi cunoștințe
- ➍ Prințeperea tehnică poate fi mai puțin importantă decât abilitățile sociale

# Factori la Selecția Personalului

---

- *Experiența în domeniul aplicației.* Deoarece succesul proiectului depinde de cunoașterea domeniul aplicației, măcar câțiva membrii ai echipei trebuie să aibă experiență în acel domeniu
- *Experiența în utilizarea platformei.* Este esențială pentru proiecte implicând programare la nivel scăzut, în celealte situații nu este un atribut critic
- *Experiența în limbajul de programare.* Este semnificativă de obicei pentru proiecte de durată scurtă unde nu este timp pentru a învăța un nou limbaj. De obicei învățarea ia câteva luni pentru cunoașterea în amănunt a bibliotecilor și componentelor
- *Abilitatea de rezolvare a problemelor.* Este foarte importantă pentru programatori care au constant de rezolvat probleme tehnice. Cu toate acestea, este dificil de judecat fără o cunoaștere prealabilă a persoanei respective

# Factori la Selecția Personalului

---

- ➊ *Educația.* Poate fi un indicator al cunoștințelor fundamentale ale candidatului și a abilității sale de a învăța. Devine mai puțin relevant pe măsură ce se câștigă experiență din lucru la proiecte
- ➋ *Abilități comunicационale.* Sunt importante deoarece frecvent apare necesitatea de a comunica direct sau în scris cu alți membri ai echipei, cu managerii și cu clienții.
- ➌ *Adaptabilitatea.* Poate fi judecată după varietatea tipurilor de experiențe avute de candidat. Acest atribut arată abilitatea de a învăța lucruri noi
- ➍ *Atitudinea și personalitatea.* Membrii echipei trebuie să aibă o atitudine pozitivă față de muncă și să dorească să învețe lucruri noi. De asemenea contează mult și atitudinea membrilor echipei unul față de celălalt. Sunt attribute greu de evaluat

# Motivarea Personalului

- ➊ Un rol important al managerului este acela de a motiva personalul care lucrează la proiect
- ➋ Motivarea este o problemă complexă care poate fi abordată din unghiuri diverse:
  - Nevoi primare (salariu, mâncare, odihnă etc.)
  - Nevoi personale (respect, apreciere de sine etc.)
  - Nevoi sociale (să fie acceptat ca parte a unui grup etc.)

# Tipuri de Personalități

## ➤ Orientate pe muncă

- Motivația de a munci este munca în sine

## ➤ Orientate către sine

- Munca este un scop de a obține avantaje personale – bani, posibilitatea de a călătorii etc.

## ➤ Orientate spre interacțiune

- Motivația principală este munca în echipă, prezența colegilor de echipă. Oamenii merg la lucru deoarece le place să meargă la lucru

# Managementul Grupurilor

- Cea mai mare parte a ingineriei software este o activitate de grup
  - Planificarea dezvoltării majorității proiectelor non-triviale este de așa natură încât nu poate fi făcută de o singură persoană
- Interacțiunile la nivel de grup este esențială pentru performanța grupului ca întreg
- Flexibilitatea în compoziția grupului este limitată
  - Managerii trebuie să obțină maximul de rezultate cu personalul pe care îl are la dispoziție

# Compoziția Grupului

- Compoziția unui grup din membri cu aceeași personalitate poate fi o problemă:
  - Orientați pe muncă – toți vor să facă doar munca lor și atât
  - Orientați către sine – toți vor să conducă
  - Orientați către interacțiune – multă vorbă, puțin lucru efectiv
- Un grup eficient trebuie să balanseze aceste tipuri
- Este greu de obținut deoarece programatorii sunt de obicei orientați către muncă
- Persoanele orientate către interacțiune sunt foarte importante deoarece ei pot dezamorsa tensiunile care apar în mod obișnuit în orice grup

# Conducerea unui Grup

---

- Conducerea este o noțiune care ține de respect și nu este pur și simplu un titlu
- Pot fi atât un conducător tehnic cât și unul administrativ
- Conducerea democratică este mai eficientă decât cea autocratică

# Coeziunea Grupului

- ➊ Într-un grup coeziv, membrii vor considera grupul mai important decât orice individ din el
- ➋ Avantajele grupurilor coeziive sunt:
  - Pot fi dezvoltate standarde de calitate la nivel de grup
  - Membrii grupului lucrează apropiat aşa că vor fi minimezate inhibițiile cauzate de ignoranță
  - Membrii unei echipe învață unii de la alții și ajung să cunoască munca celorlalți
  - Se poate practica programarea lipsită de egoism, în care programatorii se străduiesc să-și îmbunătățească unii altora programele

# Comunicarea În Interiorul Grupului

---

## ■ Mărimea grupului

- Dificultatea comunicării cu ceilalți membri crește odată cu dimensiunea grupului

## ■ Structura grupului

- Comunicarea este mai bună în grupurile structurate informal decât în cele structurate ierarhic

## ■ Compoziția grupului

- Comunicarea este mai bună când sunt tipuri de personalități diferite și când sunt formate și din bărbați și din femei

## ■ Mediul fizic de lucru

- Un loc de muncă bine organizat poate încuraja comunicarea

# Organizarea Informală a Grupurilor

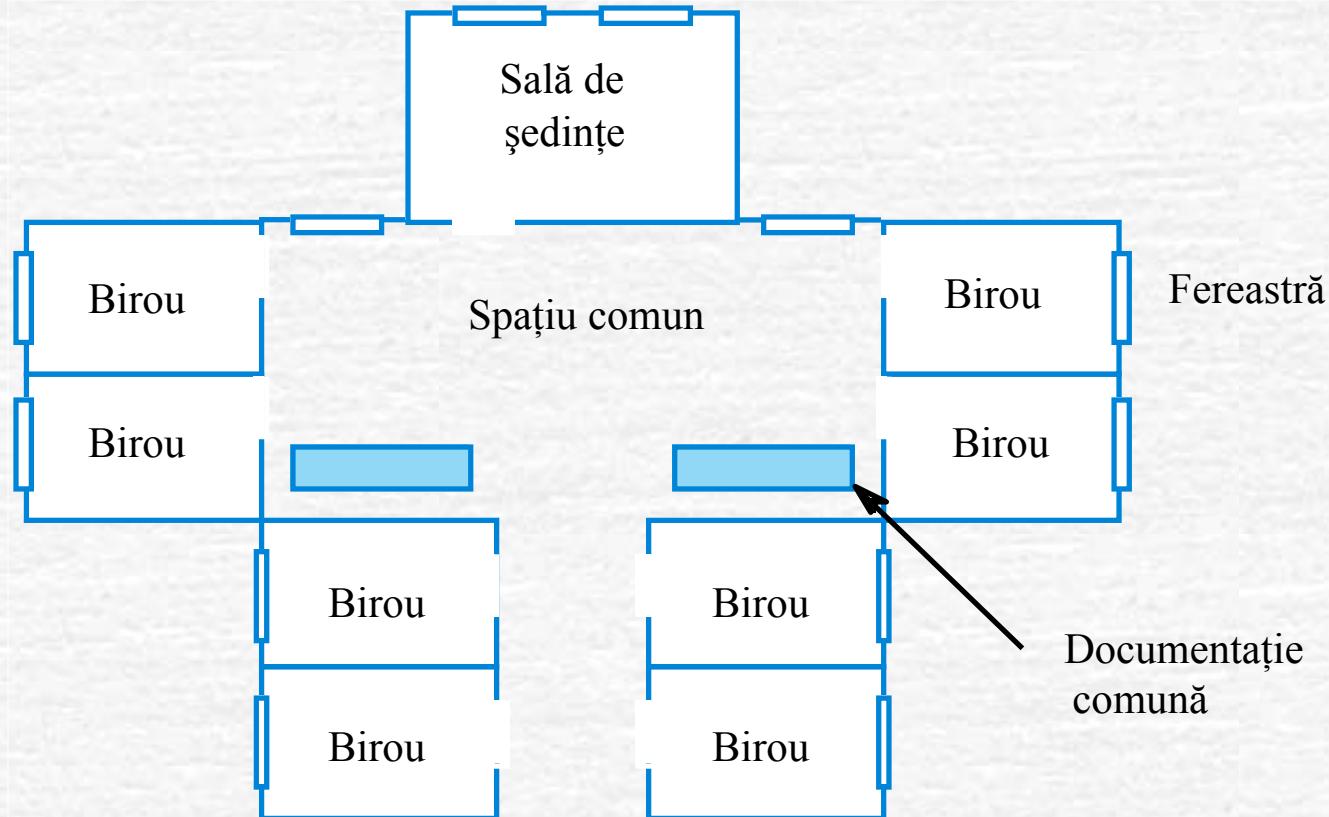
---

- Grupul trebuie să acționeze ca un întreg și să ajungă la un consens în privința deciziilor care afectează sistemul
- Conducătorul grupului servește ca interfață cu exteriorul pentru întregul grup
- Deciziile privind alocarea muncii sunt luate în grup și sarcinile sunt alocate ținând cont de abilități și experiență
- Acest mod de organizare este unul de succes pentru unde toți membrii grupului au experiență și sunt la fel de competenți

# Organizarea Mediului de Lucru

- ➊ Locul de muncă are un impact important în productivitatea și satisfacția individuală prin:
  - Confort
  - Spațiu personal, confidențial – să poată lucra fără a fi întrerupt
  - Facilități
- ➋ Considerentele legate de sănătate și siguranță trebuie de asemenea luate în considerare:
  - Iluminare – de preferat lumină naturală
  - Încălzire
  - Mobilier

# Exemplu de Organizare Optimală



# Elemente de Estimare a Costurilor

---

- ➊ Cât efort este necesar pentru efectuarea activității?
- ➋ Cât timp este necesar pentru completarea activității?
- ➌ Care este costul total al unei activități?
- ➍ Estimarea și planificarea proiectului sunt activități de management întrețesute

# Componentele Costului Software

---

- Costul efectiv pentru hardware și software
- Costul pentru mobilități și training
- Costul efortului (factor dominant în majoritatea proiectelor)
  - Salariile inginerilor implicați în proiect
  - Costuri sociale și asigurări
- Trebuie avute în vedere și costurile adiționale
  - Costuri de chirii, încălzire și iluminare
  - Costul rețelelor și comunicațiilor
  - Costul facilităților comune (bibliotecă, restaurant, etc.)

# Relația dintre Preț și Cost

- Estimările sunt folosite pentru a descoperi costurile producerii sistemului software
- Nu există o relație simplă între costul de dezvoltare și prețul cerut clientului
- La stabilirea prețului trebuie avute în vedere considerente mai largi precum organizaționale, economice, politice și de afaceri

# Factori care Influențează Prețul

- *Oportunitățile de piață* – poate fi considerat un preț mai mic dacă se dorește intrarea pe un alt segment al pieței. Acceptarea unui profit mai scăzut pe acest proiect poate aduce beneficii importante în viitor. Un beneficiu este și experiența acumulată
- *Imprecizia estimării costurilor* – dacă există dubii cu privire la estimarea costurilor prețul va fi crescut peste profitul propus pentru a nu lucra în pierdere dacă se depășesc costurile
- *Termenii contractuali* – trebuie avut în vedere cine va detine proprietatea asupra codului dezvoltat: clientul sau dezvoltatorul
- *Volatilitatea cerințelor* – dacă este previzibilă schimbarea cerințelor, se poate scădea prețul pentru a câștiga contractul. După aceasta se poate cere prețuri mai mari pentru modificări
- *Situația financiară* – în situații financiare dificile, dezvoltatorii pot scădea prețul pentru a câștiga contractul.

# Măsurarea Productivității

## ➤ Măsuri dimensionale

- Se bazează pe ieșirea procesului software- Acestea pot fi: numărul de liniile de cod sursă livrate, numărul de instrucțiuni în limbaj mașină etc.

## ➤ Măsuri funcționale

- Se bazează pe estimarea funcționalității software-ului livrat. Cea mai cunoscută măsură este *function-points*. Acestea depind de intrările/ieșirile externe, interacțiunea cu utilizatorul, interfețe externe și fișiere utilizate

# Tehnici de Estimare a Costului

---

- Nu există o cale simplă de a crește acuratețea estimării efortului necesar la dezvoltarea unui sistem software
  - Estimările inițiale se bazează pe informații inadecvate la definirea cerințelor utilizatorului
  - Software-ul poate să ruleze pe o platformă diferită sau utilizând o nouă tehnologie
  - Persoanele implicate în proiect pot să nu se cunoască
- Estimarea costului proiectului se poate auto-impune
  - Estimarea definește un buget și produsul este ajustat să se încadreze în acel buget

# Feluri de Estimări de Cost

- ➊ Orice abordare poate fi utilizată fie top-down fie bottom-up
- ➋ Top-down
  - Pornește de la nivelul sistemului și stabilește funcționalitatea generală a sistemului și cum va fi ea livrată de sub-sisteme. Atenție: se pot subestima costurile rezolvării problemelor complexe de nivel scăzut
- ➌ Bottom-up
  - Pornește de la nivelul componentelor și estimează efortul necesar pentru fiecare componentă. Acestea sunt apoi însumate pentru a găsi estimarea finală. Atenție: poate subestima costurile pentru integrare și documentație

# Managementul Calității

---

- Se ocupă de asigurarea atingerii unui anumit nivel de calitate al produsului software
- Implică definirea unor standarde de calitate și proceduri care apoi sunt urmărite spre a fi respectate
- Își propune dezvoltarea unei “culturi a calității”, în care calitatea este văzută ca responsabilitatea fiecăruia

# Ce este Calitatea

---

- ➊ Calitatea, în mod simplist, înseamnă că un produs trebuie să se conformeze specificațiilor sale
- ➋ Acest lucru este problematic aici pentru că
  - Există o tensiune între cerințele de calitate ale clientului (eficiență, fiabilitate etc.) și cerințele de calitate ale dezvoltatorului (mentenabilitate, reutilizabilitate etc.)
  - Anumite cerințe de calitate sunt dificil de specificat într-un mod neambiguu
  - Specificațiile software sunt de obicei incomplete și deseori inconsistent

# Scopul Managementului Calității

- Managementul calității este foarte important pentru sisteme mari și complexe. Documentația despre calitate este o înregistrare a progreselor înregistrate și asigură continuarea dezvoltării dacă se schimbă echipa care lucrează la proiect
- Pentru sisteme mici, managementul calității necesită mai puțină documentație și trebuie focalizat în stabilirea “culturii calității”

# Activități ale Managementului Calității

---

## Asigurarea calității

- Stabilește proceduri și standarde de calitate la nivelul organizației

## Planificarea calității

- Selectează procedurile și standardele aplicabile pentru un anumit proiect și le modifică când este necesar

## Controlul calității

- Asigură faptul că procedurile și standardele sunt urmate de către echipa de dezvoltare

## Managementul calității trebuie separat de managementul proiectului pentru a asigura independentă (a nu ceda unor constrângeri bugetare)

---

# Calitatea Procesului și a Produsului

---

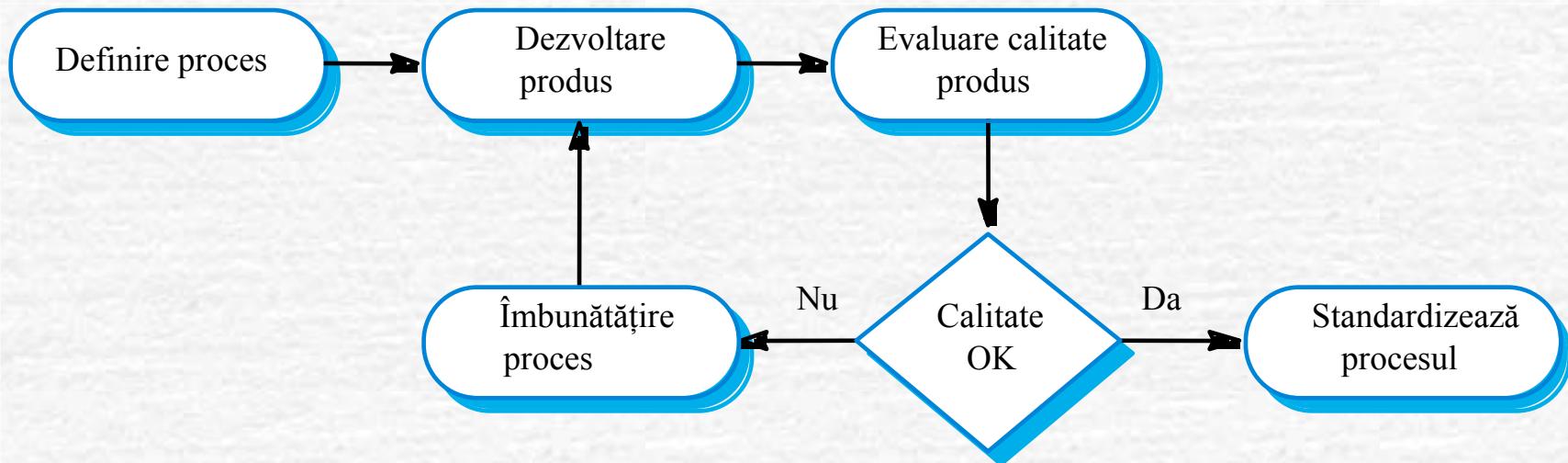
- Calitatea produsului dezvoltat este influențată de calitatea procesului de producție
- Acest lucru este important la dezvoltarea software cât timp anumite atrbute de calitate ale produsului sunt greu de evaluat
- Cu toate acestea, relația dintre procesul software și calitatea produsului este complexă și insuficient înțeleasă

# Calitatea Bazată pe Proces

---

- Există o legătură directă între procesul de producție și produse în orice bunuri produse industrial
- Lucrurile stau mai complicat la software deoarece:
  - Aplicarea abilităților individuale și experienței este deosebit de importantă la dezvoltarea software
  - Factori externi precum noutatea unei aplicații sau necesitatea unui proces de dezvoltare accelerat pot avea impact în calitatea produsului
- Trebuie avut grijă să nu se impună standarde de proces neadecvate - acestea pot cauza reducerea calității produsului în loc să o crească

# Calitatea Bazată pe Proces



# Utilizarea Standardelor

---

- Încapsulează practicile corecte – ajută la evitarea repetării erorilor din trecut
- Reprezintă un cadru de aplicare a procesului de asigurare a calității – implică verificarea alinierii la respectivele standarde
- Asigură continuitatea – angajații noi vor înțelege organizarea producției prin prisma standardelor care sunt utilizate

# Probleme la Aplicarea Standardelor

---

- ➊ Pot fi privite cu neîncredere de către inginerii software – aceştia le pot considera irelevante sau învechite
- ➋ Implică de multe ori multă muncă birocratică pentru completarea formularelor și rapoartelor
- ➌ Dacă nu sunt suportate de uneltele software utilizate, atunci vor implica o muncă plăcătoare pentru a menține documentația asociată cu aceste standarde

# ISO 9000

- Un set internațional de standarde pentru managementul calității
- Aplicabil la o mare varietate de organizații de la cele de producție până la cele de prestări servicii
- ISO 9001 este aplicabil organizațiilor care proiectează, dezvoltă și asigură mențenanța diverselor produse
- ISO 9001 este un model generic pentru procesul calității. El trebuie instantiat pentru fiecare organizație în parte utilizând standardul general

# ISO 9001

## Responsabilități management

- Controlul produselor ne-conforme
- Manevrarea, stocarea, împachetarea și livrare
- Produse furnizate clientului
- Controlul procesului
- Echipament de inspecție și testare
- Recenzie contract
- Control document
- Audituri interne de calitate
- Întreținere

## Quality system

- Controlul proiectului
- Achiziții
- Identificarea și urmărirea produsului
- Inspectare și testare
- Stare inspectii și testare
- Acțiuni corective
- Înregistrări despre calitate
- Instruire
- Tehnici statistice

# Certificare ISO 9000

---

- Standardele de calitate și procedurile trebuie documentate într-un manual al calității la nivel de organizație
- Un organism extern poate certifica că o manualul calității dintr-o organizație se conformează standardului ISO 9000
- Anumită clienți pot cere furnizorilor fie certificați ISO 9000, totuși este recunoscută necesitatea unei flexibilități în acest caz

# Standarde pentru Documentație

---

- Deosebit de important – documentele trebuie să manifestarea palpabilă a software-ului
- Standarde pentru procesul de scriere a documentației
  - Se referă la cum trebuie create, validate și menținute documentele
- Standarde referitoare la documente
  - Se axează pe conținutul, structura și forma documentelor
- Standarde pentru schimbul de documente
  - Se ocupă de compatibilitatea documentelor electronice

# Planificarea Calității

- Un plan pentru calitate stabilește calitățile necesare ale produsului și cum sunt apoi evaluate. Definește de asemenea cele mai semnificative atrbute de calitate
- Planul pentru calitate trebuie să definească și procesul de evaluare a calității
- Trebuie să stabilească și care dintre standardele organizației trebuie aplicate, și, unde este necesar să definească noi standarde ce trebuie utilizate

# Structura unui Plan pentru Calitate

## Structura planului

- Scurtă descriere a produsului
- Planuri de produs
- Descriere proces
- Ținte de calitate
- Riscuri și managementul riscului

## Planurile pentru calitate trebuie să fie documente scurte și succinte

- Dacă sunt prea lungi nu le va citi nimeni

# Atribute de Calitate ale Software-ului

- Cele mai importante atribute de calitate sunt:

Siguranța	Claritatea	Portabilitatea
Securitatea	Testabilitatea	Utilitatea
Fiabilitatea	Adaptabilitatea	Reusabilitatea
Rezistența	Modularitatea	Eficiența
Robustețea	Complexitatea	Ușurința de învățare

# Controlul Calității

---

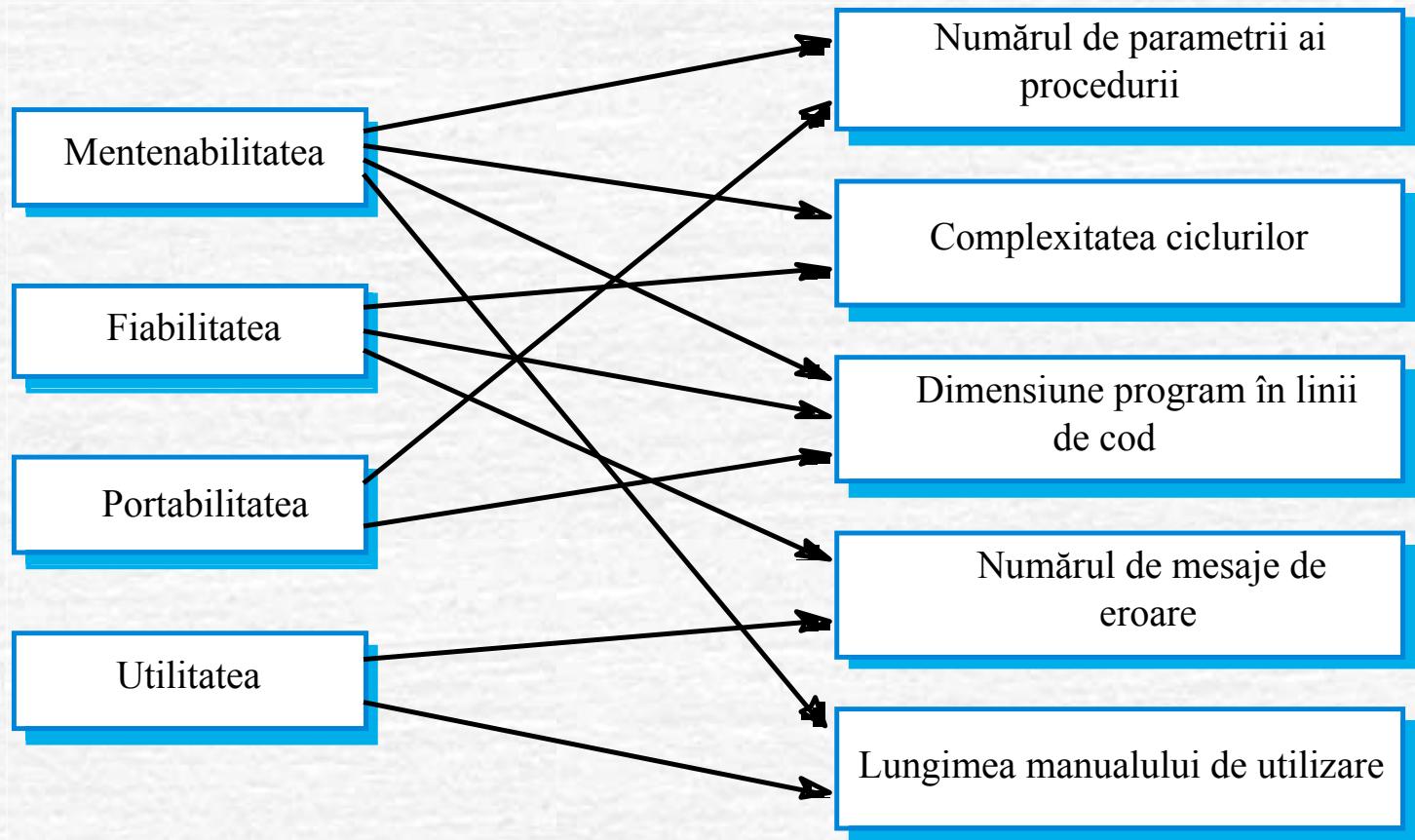
- Implică verificarea procesului de dezvoltare software pentru a asigura că procedurile și standardele sunt urmate întocmai
- Există două abordări posibile pentru controlul calității
  - Recenzii ale calității pentru proiect, cod, specificații, standarde și documentație
  - Evaluarea și măsurarea automată prin software

# Metrici Software

---

- Orice tip de măsurători legate de un sistem software, proces sau documentație
  - Ex: număr de linii de cod, numărul de persoane/zi pentru dezvoltarea unei componente etc.
- Permit calificarea unui produs sau proces software
- Pot fi folosite pentru a prezice atrbute ale produsului sau pentru a controla procesul software
- Metricile pot fi folosite pentru predicții generale sau pentru identificarea componentelor anormale

# Atribute Interne și Externe



# Concluzii

- Factorii care influențează selecția personalului sunt: nivelul educației, experiența în domeniul aplicației, adaptabilitatea și personalitatea.
- Oamenii sunt motivați prin interacțiune, recunoașterea meritelor și dezvoltare personală
- Grupurile de dezvoltare software trebuie să fie mici și coeze. Conducătorii trebuie să fie competenți și să asigure sprijin atât tehnic cât și administrativ
- Comunicarea în cadrul grupului este afectată de dimensiune, organizare și de compoziția personalităților membrilor

# Concluzii

---

- ➊ Mediul de lucru trebuie să includă spații private de lucru și spații de discuții
- ➋ Nu există o relație simplă între prețul unui sistem software și costurile de dezvoltare
- ➌ Sistemul software poate fi oferit la un preț mai scăzut pentru a câștiga un contract și funcționalitatea sa ajustată mai acestui preț
- ➍ Managementul calității software are ca și scop asigurarea că software-ul îndeplinește anumite standarde

# Concluzii

---

- Procedurile de asigurare a calității trebuie documentate într-un manual al calității specific organizației
- Standardele software încapsulează cele mai bune practici
- Recenzia este cea mai utilizată abordare pentru evaluarea calității software
- Măsurătorile software strâng informații și despre procesul software și despre produsul software
- Metricile de calitate trebuie utilizate pentru a identifica componentele potențial cu probleme