

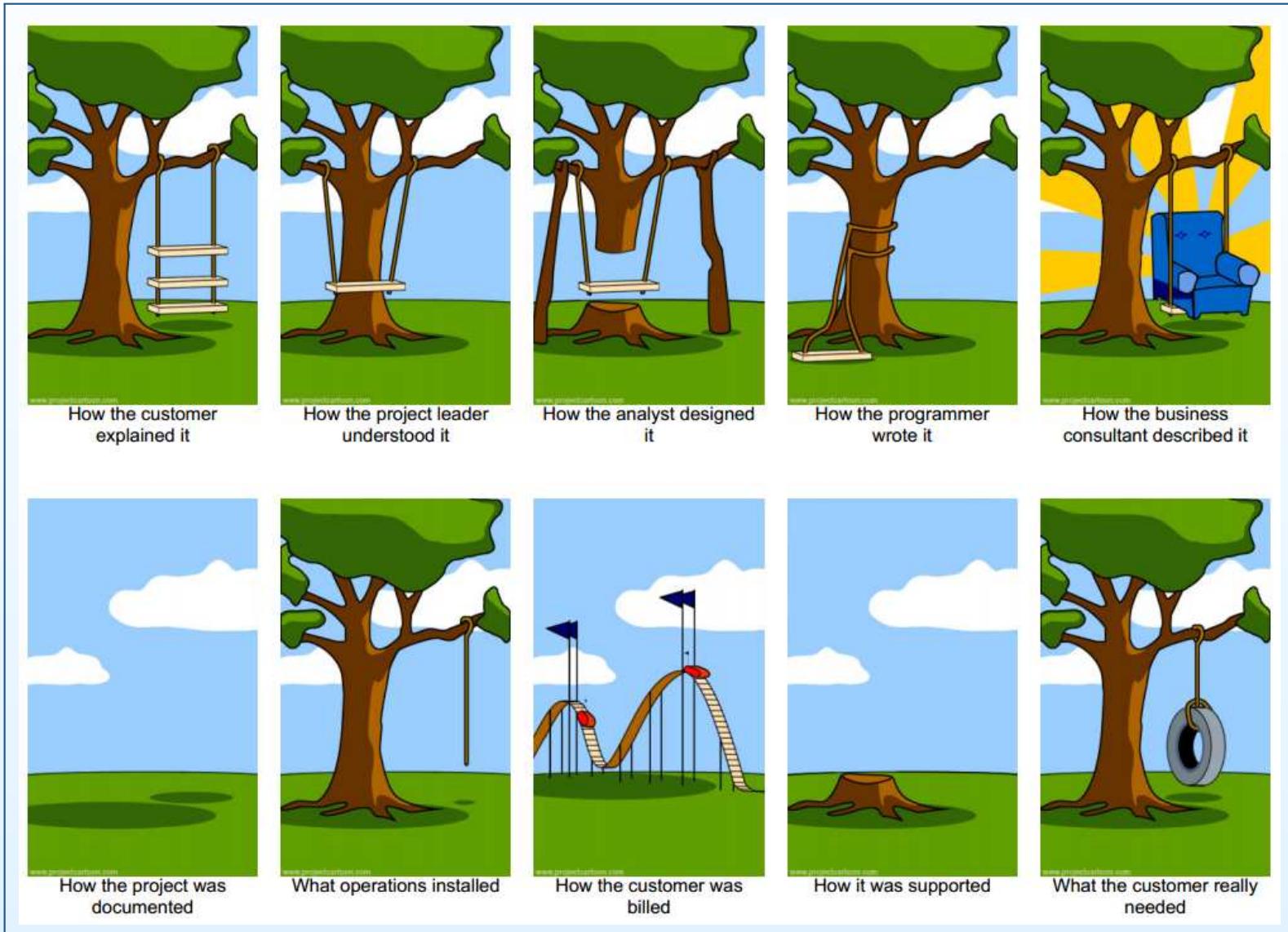
# *Ingineria Sistemelor Soft*

## *Curs 1*

### *Introducere în Ingineria Sistemelor Soft*

*Suport de curs bazat pe B. Bruegge and A.H. Dutoit,  
"Object-Oriented Software Engineering using UML, Patterns, and Java"*

# Communication in SE - a Tree Swing story!



## Obiective curs

---

- Înțelegerea conceptelor legate de modelarea softului
- Cunoașterea și aplicarea tehniciilor de dezvoltare a softului pe baza modelelor
- Familiarizarea cu limbajul UML
- Abilitatea de a utiliza instrumente CASE
- Cunoașterea etapelor ciclului de viață al softului și a modelelor de procese soft
- Familiarizarea cu unele dintre metodologiile de dezvoltare, tradiționale sau agile
- Însușirea unor aspecte de bază legate de gestiunea softului

## Utile

- Evaluare - colocviu

Modalitate de evaluare	Procent din nota finală
Proiect de laborator	40%
Examen scris	60%
Activitate seminar	bonus (în limita unui punct)

- Resurse (notițe curs, materiale seminar, cerințe laborator)  
Team: Ingineria sistemelor soft RO 2023, Files/Class Materials  
Team code: mov3xvw

## Bibliografie

---

- [1] Bruegge, B., Dutoit, A.H., *Object-Oriented Software Engineering using UML, Patterns, and Java (3rd edition)*, Prentice Hall, 2010.
- [2] Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd edition)*, Addison-Wesley, 2004.
- [3] Sommerville, I., *Software Engineering (10th edition)*, Pearson, 2015.
- [4] Pressman, R.S., *Software Engineering - A Practitioners Approach (8th edition)*, McGraw-Hill, 2014.
- [5] Schach, S.R., *Object-Oriented and Classical Software Engineering, (8th edition)*, McGraw-Hill, 2010.
- [6] IEEE Computer Society, *SWEBOK v3.0: Guide to the Software Engineering Body of Knowledge*, 2014.
- [7] Pârv, B., *Analiza si proiectarea sistemelor*, Univ. Babeş-Bolyai, CFCID, Facultatea de Matematică și Informatică, Cluj-Napoca, 2004.
- [8] Seidl, M., Scholz, M., Huemer, C., Kappel, G., *UML @ Classroom - An Introduction to Object-Oriented Modeling*, Springer, 2015.

## Bibliografie (cont.)

---

- [9] Fowler, M., *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd edition), Addison Wesley, 2003.
- [10] Rumbaugh, J., Jacobson, I., Booch, G., *The Unified Modeling Language Reference Manual* (2nd edition), Addison Wesley, 2010.
- [11] Booch, G., Rumbaugh, J., Jacobson, I., *The Unified Modeling Language User Guide*, V.2.0, Addison Wesley, 2005.
- [12] Object Management Group, *UML 2.4.1 Infrastructure & Superstructure Specification - 2012 ISO standard*,  
<https://www.omg.org/spec/UML/ISO/19505-1/PDF>.  
<https://www.omg.org/spec/UML/ISO/19505-2/PDF>.
- [13] Object Management Group, *OCL 2.3.1 Specification - 2012 ISO standard*, <https://www.omg.org/spec/OCL/ISO/19507/PDF>.
- [14] Object Management Group, *UML 2.5.1 Specification*, 2017,  
<https://www.omg.org/spec/UML/2.5.1/PDF>.
- [15] Object Management Group, *OCL 2.4 Specification*, 2014,  
<https://www.omg.org/spec/OCL/2.4/PDF>.

## Bibliografie (cont.)

---

- [16] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1996.
- [17] Fowler, M. et al., *Refactoring - Improving the Design of Existing Code*, Addison-Wesley, 1999.
- [18] Eckel, B., *Thinking in Java* (4th edition), Prentice Hall, 2006.
- [19] Beck, K., *Test Driven Development: By Example*, Addison-Wesley, 2002.
- [20] Rubin, K.S., *Essential Scrum - A Practical Guide to the Most Popular Agile Process*, Addison-Wesley, 2012.
- [21] Brambilla, M., Cabot, J., Wimmer, M., *Model-Driven Software Engineering in Practice (2nd edition)*, Morgan and Claypool Publishers, 2017.

# Structura curs

---

- Curs 1: *Introducere în ingineria sistemelor soft*
- Curs 2: *Specificarea modelelor folosind UML*
- Curs 3: *Colectarea cerințelor*
- Curs 4: *Analiza cerințelor*
- Curs 5: *Proiectarea de sistem*
- Curs 6: *Proiectarea obiectuală - Şablonane de proiectare*
- Curs 7: *Proiectarea obiectuală - Specificarea interfeţelor folosind OCL*
- Curs 8: *Implementarea sistemului. Transformarea modelelor în cod*
- Curs 9: *Testarea sistemului*
- Curs 10: *Ciclul de viaţă al sistemelor soft*
- Curs 11: *Gestiunea proiectelor soft*
- Curs 12: *Metodologii de dezvoltare a sistemelor soft*
- Curs 13: *Model-Driven Software Engineering*
- Curs 14: *Colocviu*

# Sumar Curs 1

---

- Ingineria sistemelor soft:
  - Motivație
  - Definiții
  - Evoluție
  - Eșecuri celebre
  - Provocări
- Ce presupune ingineria sistemelor soft?
  - Modelare
  - Rezolvare de probleme (eng. *problem solving*)
  - Acumulare de cunoștințe (eng. *knowledge acquisition*)
  - Argumentare (eng. *rationale*)
- Concepte ale ingineriei sistemelor soft
  - Participanți și roluri
  - Sisteme și modele
  - Produse (eng. *work products*)
  - Activități, sarcini și resurse
  - Cerințe funcționale și nefuncționale
  - Notații, metode și metodologii

## Sumar Curs 1 (cont.)

---

- Activități ale procesului de dezvoltare a softului
  - Colectarea cerințelor
  - Analiza cerințelor
  - Proiectarea de sistem
  - Proiectarea obiectuală
  - Implementarea
  - Testarea

# Ingineria sistemelor soft: Motivație

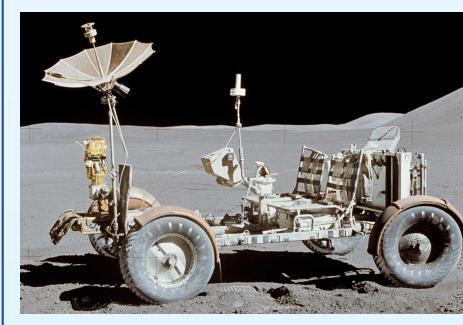
- *Criza software de la sfârșitul anilor '60*, generată de *dezvoltarea artizanală* a programelor
  - calitate slabă a softului - rezultată din cerințe nerespectate, lipsă de fiabilitate, întreținere dificilă
  - nerespectarea termenelor de livrare
  - depășirea bugetului alocat
- Dezvoltatorii de soft erau incapabili să stabilească obiective concrete, să anticipateze resursele necesare și să gestioneze așteptările clientilor

Promisă:



Luna

Dezvoltat:



Vehicul lunar

Livrat:



O pereche de roți pătrate

# Ingineria sistemelor soft: Definiții

- 1968: prima conferință NATO de Software Engineering - Bavaria
  - nevoie unei *abordări sistematice și disciplinate* a dezvoltării softului
- **Definiția 1.1 [Ingineria Programării, eng. Software Engineering]:** *Stabilirea și punerea în practică a unor principii inginerești solide, care să producă aplicații soft fiabile și care să funcționeze eficient pe mașini reale.* (Fritz Bauer, 1968)
- **Definiția 1.2 [Ingineria Programării]:** *Aplicarea unei abordări sistematice, disciplinate și cuantificabile la dezvoltarea, operarea și întreținerea softului, mai exact aplicarea ingineriei la soft.* [6]

## Ingineria sistemelor soft: Evoluție

- "Software and cathedrals are very much the same. First we build them, then we pray." (Sam Redwine, 1988)
- "Despite 50 years of progress, the software industry remains years, perhaps decades, short of the mature engineering discipline needed to meet the needs of an information-age society." (Scientific American, 1994)
- În 2002, Institutul Nord American pentru Standarde și Tehnologie a estimat costul total al erorilor soft în economia americană la 59 miliarde USD
- Studiile actuale indică faptul că majoritatea sistemelor soft dezvoltate au un număr relativ mare de erori, conducând la depășirea termenelor de predare, a bugetului alocat, precum și la probleme de utilizabilitate
- Costurile de întreținere a softului reprezintă 2/3 din costurile totale de dezvoltare; o eroare de specificare este de 20 de ori mai costisitor de reparat în fază de testare, față de descoperirea ei la începutul dezvoltării sistemului

# Ingineria sistemelor soft: Eșecuri celebre

---

- *Therac-25 (1985-1987)*
  - Între 1985 și 1987, sistemul medical Therac-25, utilizat în terapia pacienților cu cancer, a fost implicat în 6 accidente (cu decese și răniri grave), aplicând pacienților supradoze mari de radiații (de peste 100 de ori mai mari decât limita admisă).
- *Ariane 5 (1996)*
  - Pe 4 iunie 1996, racheta Ariane 5 a explodat la nici 40 de secunde de la lansare. Pierderile înregistrate au fost de 500 milioane USD (racheta + cargo), plus un deceniu de dezvoltare estimat la aproximativ 7 miliarde USD.
- *Eroarea procesoarelor Pentium (1994)*
  - În 1994 s-a descoperit o eroare în unitatea de împărțire cu virgulă mobilă a procesoarelor Pentium. Pierderile înregistrate au fost de circa 400 milioane USD, plus prejudiciile de imagine.
- *Eroarea anului 1900 (1992)*
  - În anul 1992, Mary din Winona, Minnesota a primit invitația de a se înscrie la o grădiniță. Mary avea 104 ani atunci.

# Ingineria sistemelor soft: Eșecuri celebre (cont.)

- *Eroarea anului bisect* (1988)
  - Un supermarket a fost amendat cu 1000 USD pentru că a ținut produse pe raft o zi în plus în februarie 1988. Programul care a imprimat data expirării pe etichete nu a ținut cont de fapul că anul era bisect.
- *Eroarea de utilizare a interfeței* (1990)
  - Pe 10 aprilie 1990, în Londra, un metrou a pornit din stație fără conductor. Acesta a apăsat butonul de pornire, știind că sistemul nu permitea plecarea trenului cu ușile deschise. Ulterior, a părăsit trenul pentru a închide o ușă care se blocate. Când aceasta s-a închis, trenul a plecat pur și simplu.
- *Neîncadrare în timp și buget* (1995)
  - În 1995, defecțiuni în sistemul aeroportului internațional din Denver au determinat distrugerea bagajelor clienților. Aeroportul s-a deschis 16 luni mai târziu, cu 3.2 miliarde USD peste buget și un sistem de gestiune a bagajelor preponderent manual.

# Ingineria sistemelor soft: Provocări

---

- **Complexitate**

- Sistemele soft au multiple funcționalități
- Sunt construite să îndeplinească un număr mare de obiective
- Constanță dintr-un număr mare de componente
- Mulți participanți, cu pregătiri diferite, iau parte la dezvoltarea lor
- Procesul de dezvoltare durează mulți ani

- **Schimbare**

- Cerințele se modifică, la inițiativa clientului sau pe măsură ce dezvoltatorii înțeleg domeniul problemei
- În cazul în care proiectul durează mulți ani, dezvoltatorii se pot schimba
- Sistemul se schimbă, ca urmare a greșelilor descoperite la testare
- Tehnologia se schimbă, de multe ori înainte de finalizarea proiectului

- **Definiția 1.3 [Ingineria Programării]:** *O colecție de tehnici, metodologii și instrumente care ajută la producerea unor sisteme soft de calitate, dezvoltate cu un buget dat și cu încadrare în termene, în contextul unor permanente schimbări.* [1]

# Ce presupune ingineria sistemelor soft?

---

- Modelare (eng. *Modeling*)
  - Permite gestionarea *complexității*, prin focusarea pe aspectele relevante și ignorarea detaliilor
  - Pe parcursul procesului de dezvoltare, se construiesc diverse modele ale domeniului problemei și ale sistemului
- Rezolvare de probleme (eng. *Problem-solving*)
  - *Modelele* sunt folosite pentru găsirea unei soluții
  - Evaluarea diferitelor alternative este, de multe ori, empirică
- Acumulare de cunoștințe (eng. *Knowledge acquisition*)
  - Modelarea presupune colectare de informații, organizarea și formalizarea lor
  - Acumularea de cunoștințe nu este un proces secvențial; o informație nouă poate invalida modelele anterioare
- Argumentare (eng. *Rationale*)
  - Orice decizie luată ar trebui documentată prin menționarea contextului, a alternativelor posibile și a argumentelor aferente
  - Aceasta permite înțelegerea impactului unei eventuale schimbări asupra sistemului

# Modelare

---

- Scopul științelor, în general, este *descrierea și înțelegerea sistemelor complexe*
  - științe naturale (clasice), ex.: fizică, chimie, biologie
  - științe sociale (clasice), ex.: psihologie, sociologie
  - științe "ale artificialului" (recente), ex.: știința calculatoarelor (eng. *computer science*)
- Tehnica principală folosită de științele clasice pentru gestionarea complexității este *modelarea*
- **Definiția 1.3 [Model]:** Un *model* este o reprezentare abstractă a unui sistem, care ne permite să răspundem unor întrebări cu privire la acel sistem. [1]
- **Definiția 1.4 [Modelare, model]:** *Modelarea* este procesul de reprezentare a elementelor sau esenței unui sistem sau fenomen. *Modelul* este o reprezentare simplificată a unui sistem sau fenomen, împreună cu orice ipoteze necesare pentru a descrie sistemul sau a explica fenomenul. [7]

## Modelare (cont.)

- Modele sunt utile pentru studiul sistemelor

- prea mari/complexă
- prea mici
- costisitor/periculos de experimentat în realitate
- care nu mai există, care se presupune că ar exista sau care urmează a fi construite



- Inginerii soft au nevoie să construiască

- *modele ale domeniului problemei* (eng. *application domain models*) - pentru a înțelege mediul în care sistemul operează
- *modele ale domeniului soluției* (eng. *solution domain models*)
  - pentru a înțelege sistemul care va fi construit și pentru a evalua soluțiile și alternativele posibile

- Avantajul major al ingineriei software orientate-obiect

- *modelul domeniului soluției se obține natural ca o transformare (rafinare) a modelului domeniului problemei*

# Rezolvare de probleme

- *Ingineria este o activitate de rezolvare de probleme în 5 pași*
  - I1. Formularea problemei
  - I2. Analiza problemei
  - I3. Căutarea de soluții
  - I4. Alegerea unei soluții potrivite
    - evaluări empirice, cu resurse limitate și informații incomplete
  - I5. Specificarea soluției alese
- Dezvoltarea de soft orientată obiect include, în general, 6 activități
  - D1. Colectarea cerințelor (corespondent I1)
  - D2. Analiza cerințelor (corespondent I2)
  - D3. Proiectarea de sistem (corespondent I3, I4)
  - D4. Proiectarea obiectuală (corespondent I3, I4)
  - D5. Implementarea (corespondent I5)
  - D6. Testarea
- Ingineria softului este o activitate *inginerească, nu algoritmică*
  - presupune experimentare, reutilizare de şabloane, evoluția iterativă a soluției
  - ceea ce o diferențiază de rezolvarea de probleme din alte domenii ingineresci sunt *schimbările ce au loc în domeniul problemei și al soluției în timpul rezolvării problemei*

## Acumulare de cunoștințe

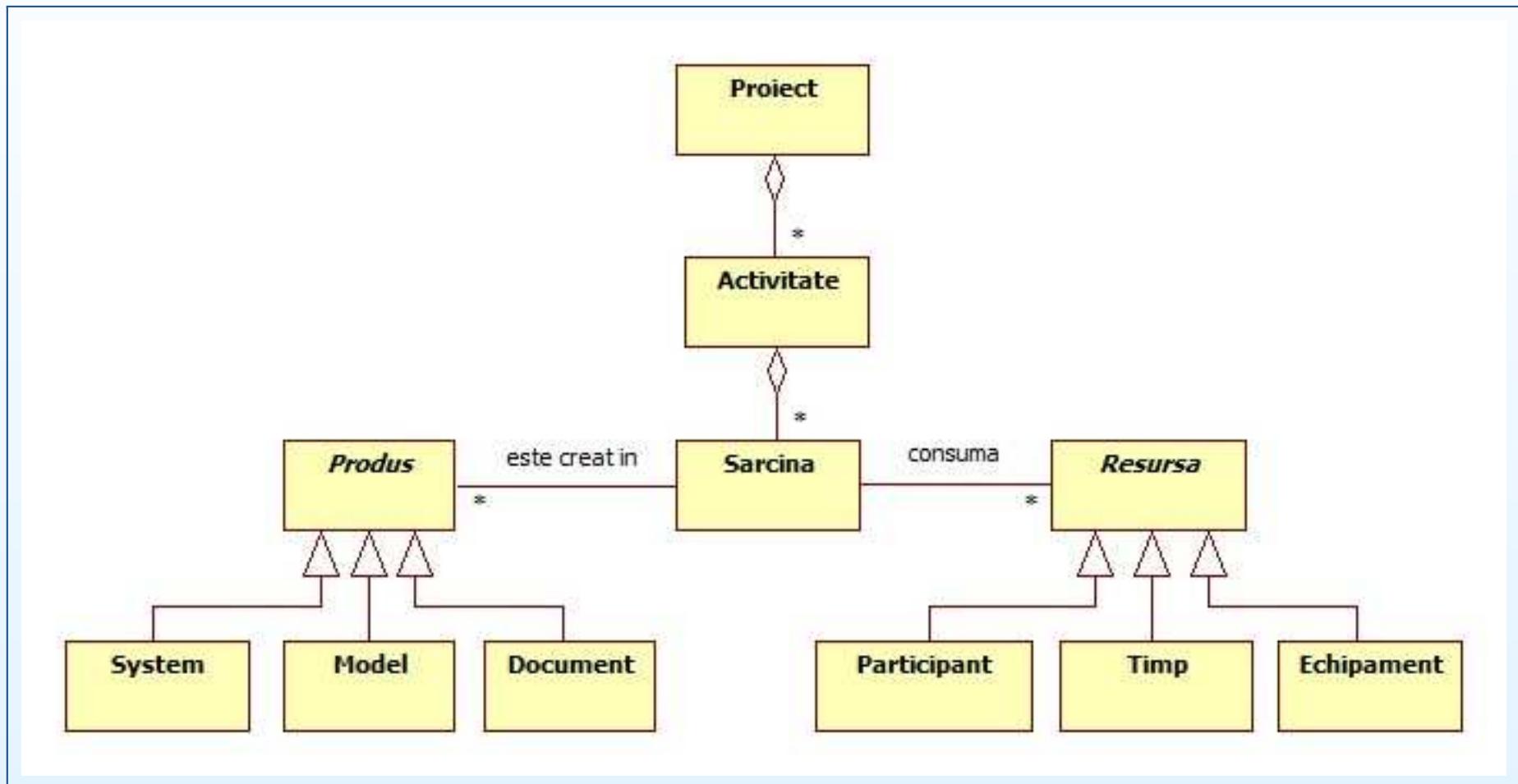
---

- Nu este un proces secvențial, întrucât o informație nou apărută poate invalida toate modelele anterior dezvoltate
  - Chiar dacă sistemul este 90% terminat, trebuie să existe disponibilitatea psihologică de a o lua de la capăt
- Acumularea secvențială de cunoștințe corespunde aşa numitei *bucket theory of the mind*
- În dezvoltarea softului, *bucket theory of the mind* se traduce în *modelul cascadă* al procesului soft
- Procese care evită dezavantajele modelului secvențial/cascadă: *risk-based development, issue-based development*
- Problema majoră a modelelor de dezvoltare non-secvențiale constă în dificultatea de a fi gestionate eficient

## Argumentare

- Deși modelul domeniului se stabilizează odată ce dezvoltatorii capătă o bună înțelegere a problemei, modelul soluției continuă să suferă modificări, ca urmare a
  - erorilor de proiectare și implementare descoperite la testare
  - problemelor de utilizabilitate raportate de clienți
  - apariției unor noi tehnologii
- Modificarea softului necesită nu doar înțelegerea componentelor și comportamentului curent, ci și cunoașterea raționamentelor din spatele deciziilor luate
- Înregistrarea și accesarea acestor raționamente este un proces netrivial, ca urmare a
  - volumului mare de informație: fiecărei decizii îi corespund, cel mai probabil, diferite alternative ce au fost considerate, discutate, evaluate
  - caracterului implicit al unor decizii luate pe baza experienței, fără o evaluare explicită a alternativelor posibile

# Concepte ale ingineriei sistemelor soft



# Participanți și roluri

- Dezvoltarea unui sistem soft presupune colaborarea a multe persoane, cu interese și specializări diferite, referite ca și *participanți*
  - *Clientul* comandă și achită sistemul
  - *Dezvoltatorii* construiesc sistemul
  - *Project managerul* planifică proiectul și coordonează dezvoltatorii și clientul
  - *Utilizatorii* folosesc sistemul
- Un *rol* referă o mulțime de responsabilități în cadrul proiectului
  - Un rol este asociat cu o mulțime de sarcini și atribuit unui participant
  - Același participant poate îndeplini mai multe roluri
- **Exemplu:** sistemul AutomatBilete
  - AutomatBilete (AB) este un automat care distribuie bilete de tren. Călătorii pot opta pentru un bilet valabil pentru o singură călătorie sau pentru un card valabil o zi sau o săptămână. Sistemul calculează prețul biletului funcție de zona în care va călători clientul și de tipul acestuia - adult sau copil. Sistemul trebuie să gestioneze diferite excepții, printre care tranzacții nefinalizate de client, plăți cu bancnote mari, precum și pană de resurse - bilete, rest sau curent.

## Participanți și roluri (cont.)

Rol	Responsabilități	Exemple
Client	furnizarea cerințelor sistemului, fixarea datei de livrare, a bugetului alocat și a criteriilor de calitate	Companie feroviară
Utilizator	oferearea de cunoștințe specifice domeniului (eng. <i>domain knowledge</i> )	Călători
Manager	gestionarea resurselor oferite de client, contractarea de personal, instruirea acestuia, atribuirea sarcinilor de lucru, monitorizarea progresului	Andrew (PM)
Dezvoltator	construcția sistemului, incluzând specificație, proiectare, implementare, testare	John (analist) Marc(programator) Zoe(tester)
Specialist HCI	utilizabilitatea sistemului	Zoe (specialist HCI)
Responsabil documentație tehnică	documentația livrată clientului; discută cu dezvoltatori, manageri și utilizatori, pentru a înțelege sistemul	John

# Sisteme și modele

---

- *System* = un ansamblu de părți interconectate
  - Ex.: Sistemul AutomatBilete
  - Ex.: Proiectul de dezvoltare al sistemului AutomatBilete
- *Model* = orice abstractizare a sistemului
  - Ex.: Specificația sistemului AutomatBilete
  - Ex.: Scheme ale circuitelor electrice ale acestuia
  - Ex.: Modele ale softului său
  - Ex.: Planul proiectului AutomatBilete
  - Ex.: Alocarea bugetul său
  - Ex.: Termenele stabilite

# Produse

---

- *Produs* (eng. *work product*) = artefact realizat în timpul procesului de dezvoltare
  - Ex.: un document sau o componentă soft pentru dezvoltatori sau client
  - Clasificare
    - Produse de lucru interne (eng. *internal work products*) - utilizate doar în cadrul proiectului
    - Produse livrabile (eng. *deliverables*) - realizate pentru client și specificate în contract

Produs	Tip	Descriere
Specificație	Livrabil	Descrie detaliat sistemul AB din perspectiva utilizatorului. Este utilizată ca și contract între client și dezvoltator.
Manual de operare	Livrabil	Este utilizat de către angajații companiei feroviare responsabili cu instalarea și configurarea sistemului AB. Descrie, spre exemplu, modul în care poate fi schimbat prețul biletelor.
Raport de stare	Intern	Descrie, pentru o dată anume, sarcinile terminate și cele în lucru. Produs pentru manager, inaccesibil clientului.
Manual de teste	Intern	Înregistrează defectele prototipului AB și starea lor curentă (eliminate, în lucru). Produs de tester, inaccesibil clientului.

# Activități, sarcini și resurse

- **Activitate** = o mulțime de sarcini realizate cu un anumit scop.  
Activitățile pot fi compuse din alte (sub)activități
  - Colectarea cerințelor este o activitate având drept scop definirea funcționalităților sistemului și a constrângerilor impuse asupra lui
  - Gestiunea proiectului este o activitate având drept scop monitorizarea și controlul proiectului, astfel încât acesta să-și atingă obiectivele (termen, buget, calitate)
  - Predarea la beneficiar este o activitate având drept scop instalarea sistemului la o locație operațională
    - Activitatea de predare include o activitate de instalare a softului și una de instruire a operatorilor
- **Sarcină** = o unitate atomică, gestionabilă de lucru
  - Managerul o atribuie dezvoltatorului, dezvoltatorul o îndeplinește, managerul urmărește progresul și finalizarea sarcinii
  - Sarcinile consumă resurse, generează produse și depind de produsele altor sarcini
- **Resursă** = bun utilizat pentru realizarea unor sarcini
  - Ex.: timp, echipamente, persoane

## Activități, sarcini și resurse (cont.)

Exemplu	Tip	Descriere
Colectarea cerințelor	Activitate	Include obținerea și validarea cerințelor și a cunoștințelor legate de domeniul problemei de la client și utilizatori. Produce specificația sistemului.
Dezvoltarea cazului de test "Pană rest"	Sarcină	Se referă la verificarea comportamentului sistemului AB în cazul în care rămâne fără bani și nu poate returna rest. Include specificarea contextului, a datelor de intrare și a rezultatelor așteptate. Atribuită lui Zoe (tester).
Revizuirea cazului de utilizare "Accesare help online" pentru evaluarea utilizabilității	Sarcină	Se referă la detectarea problemelor de utilizabilitate în accesarea funcționalităților de help online. Atribuită lui Zoe (specialist HCI).
Baza de date "Tarife"	Resursă	Include un exemplu de tarifare și o structurare pe zone a rețelei. Oferită de client pentru colectarea cerințelor și testare.

## Cerințe funcționale și nefuncționale

---

- *Cerință funcțională* = specificare a unei funcționalități pe care sistemul va trebui să o ofere
  - Ex.: Utilizatorul va putea să achiziționeze bilete
  - Ex.: Utilizatorul va putea să acceseze informații legate de tarifare
- *Cerință nefuncțională* = constrângere legată de operarea sistemului (fără a fi asociată unei funcționalități anume)
  - Ex.: Sistemul va oferi feedback utilizatorului în mai puțin de o secundă
  - Ex.: Culorile utilizate în interfața grafică vor fi cele ale companiei
  - Ex.: Se va folosi o anumită platformă hardware
  - Ex.: Se va asigura compatibilitatea cu un sistem mai vechi

# Notății, metode, metodologii

- *Notăție* = mulțime de reguli, textuale sau grafice, folosite pentru reprezentarea modelelor
  - Ex.: UML (Unified Modeling Language) - notație grafică pentru reprezentarea modelelor orientate obiect
  - Ex.: OCL (Object Constraint Language) - notație textuală pentru reprezentarea constrângerilor pe modelele UML
  - Ex.: diagrama de flux de date, diagrama de flux de control - notații grafice pentru reprezentarea modelelor în modelarea clasică / structurată
  - Ex.: B - notație formală textuală pentru reprezentarea sistemelor, bazată pe logică și teoria mulțimilor
- *Metodă* = tehnică cu caracter repetabil, constând într-o succesiune de pași aplicați în scopul rezolvării unei anumite probleme
  - Ex.: o rețetă este o metodă de a găti un anumit fel de mâncare
  - Ex.: un algoritm de sortare este o metodă de ordonare a unui sir
- *Metodologie* = colecție de metode folosite pentru rezolvarea unei anumite clase de probleme, incluzând informații referitoare la modalitatea de aplicare a acestora (când și cum ar trebui aplicate)

## Notății, metode, metodologii (cont.)

---

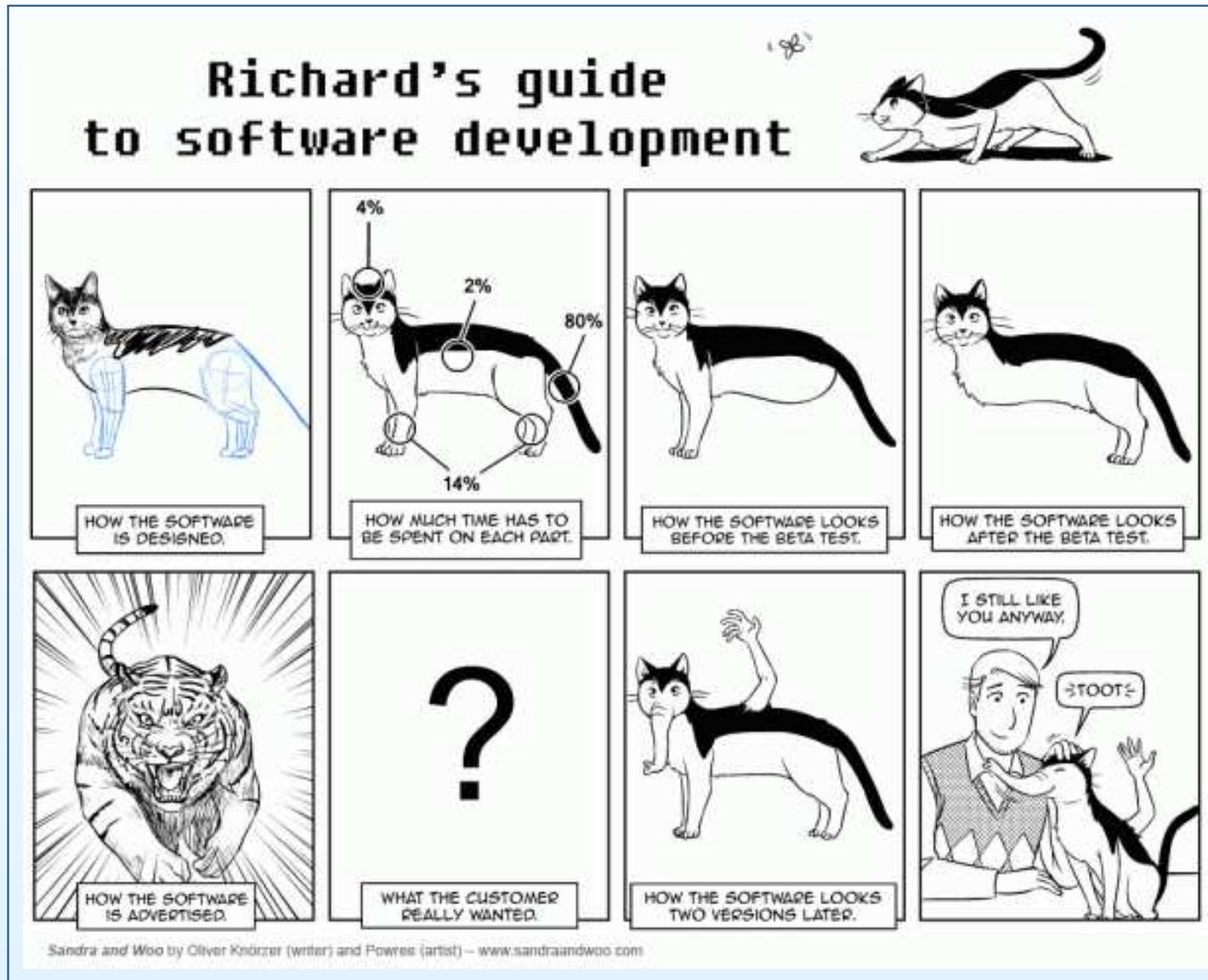
- Ex.: o carte de bucate raw reprezintă o metodologie de preparare a alimentelor raw, în cazul în care conține indicații legate de utilizarea ingredientelor sau substituirea acestora
- Ex.: OMT (Object Modeling Technique), OOSE (Object Oriented Software Engineering), metodologia Booch, USDP (Unified Software Development Process) sau Catalysis sunt metodologii de dezvoltare a softului
- Metodologiile de dezvoltare a softului descompun procesul în activități
  - OMT oferă metode pentru 3 activități
    - Analiză
    - Proiectare de sistem
    - Proiectare obiectuală
  - USDP oferă metode pentru 3 activități
    - Colectarea cerințelor
    - Analiză
    - Proiectare

# Activități ale procesului de dezvoltare a softului

---

- Activități tehnice ale ingineriei soft orientate obiect (conform metodologiei din [1])
  - Colectarea cerințelor (eng. *Requirements Elicitation*)
  - Analiza cerințelor (eng. *Analysis*)
  - Proiectarea de sistem (eng. *System Design*)
  - Proiectarea obiectuală (eng. *Object Design*)
  - Implementarea (eng. *Implementation*)
  - Testarea (eng. *Testing*)
- Aceste activități gestionează complexitatea prin construirea și validarea de modele ale domeniului problemei și domeniului soluției

# Software Engineering, now with cats



# Colectarea cerințelor

- Finalitatea acestei etape o constituie definirea, de către client și dezvoltatori, a scopului/cerințelor sistemului
- Produse
  - *Modelul funcțional*
    - descriere a sistemului în termeni de actori și cazuri de utilizare
      - *Actor* = rol jucat de o entitate externă sistemului, care interacționează cu sistemul (utilizator uman, alt calculator/dispozitiv/sistem)
      - *Caz de utilizare* = secvență generală de evenimente descriind toate interacțiunile posibile între un actor și sistem, pentru îndeplinirea unei anumite funcționalități
    - instrumente de reprezentare
      - *diagramă UML a cazurilor de utilizare*
      - sabloane pentru descrierea textuală explicită a cazurilor de utilizare
  - *Cerințele nefuncționale*

## Colectarea cerințelor (cont.)

- Ex.: cazul de utilizare *Achiziționează bilet pentru o călătorie*

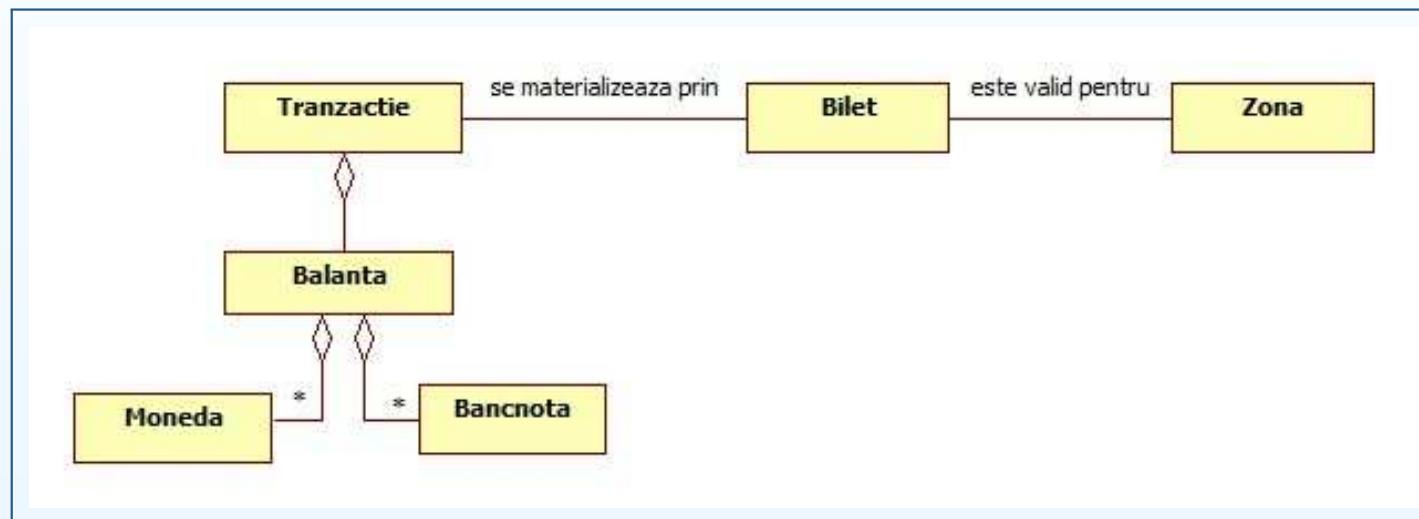
<b>Nume caz de utilizare</b>	<i>Achiziționează bilet pentru o călătorie</i>
<b>Actor</b>	<i>Inițiat de Călător</i>
<b>Flux de evenimente (scenariu normal)</b>	<ol style="list-style-type: none"><li><i>Călătorul selectează zona stației destinație.</i></li><li><i>Sistemul AutomatBilete afișează prețul biletului.</i></li><li><i>Călătorul inserează o sumă de bani cel puțin egală cu prețul biletului.</i></li><li><i>Sistemul AutomatBilete oferă biletul solicitat și restul.</i></li></ol>
<b>Condiție de intrare</b>	<i>Călătorul se află în fața automatului, localizat în stația de plecare sau într-o altă stație.</i>
<b>Condiție de ieșire</b>	<i>Călătorul primește biletul solicitat și restul.</i>
<b>Cerință de calitate</b>	<i>În cazul în care tranzacția nu se finalizează, după un minut de inactivitate, sistemul restituie călătorului întreaga sumă inserată.</i>

# Analiza cerințelor

- În cadrul acestei etape, analiștii derivă din descrierea cazurilor de utilizare ale etapei precedente un model obiectual al sistemului
  - Scopul este realizarea unui model *complet, consistent și neambiguu*
  - În cadrul procesului de realizare a modelului obiectual de analiză, analiștii identifică inconsistențe, incompletitudini sau ambiguități în modelul funcțional, pe care le rezolvă cu clientul
- Modelul obiectual de analiză poate fi descris
  - În termenii structurii sale  $\implies$  *model conceptual*, reprezentat printr-o *diagramă de clase*
  - În termenii comportamentului său  $\implies$  *model dinamic*, reprezentat prin *diagrame de interacțiune*
- Modelul obiectual de analiză conține *doar concepte caracteristice domeniului problemei* (fără referire la domeniul soluției) !
  - În colectarea și analiza cerințelor răspundem la întrebarea "ce?" (funcționalități oferă sistemul / constrângerile se impun asupra lui / concepte manipulează), fără a face vreo referire la modul "cum?" se realizează acele cerințe (probleme de proiectare)

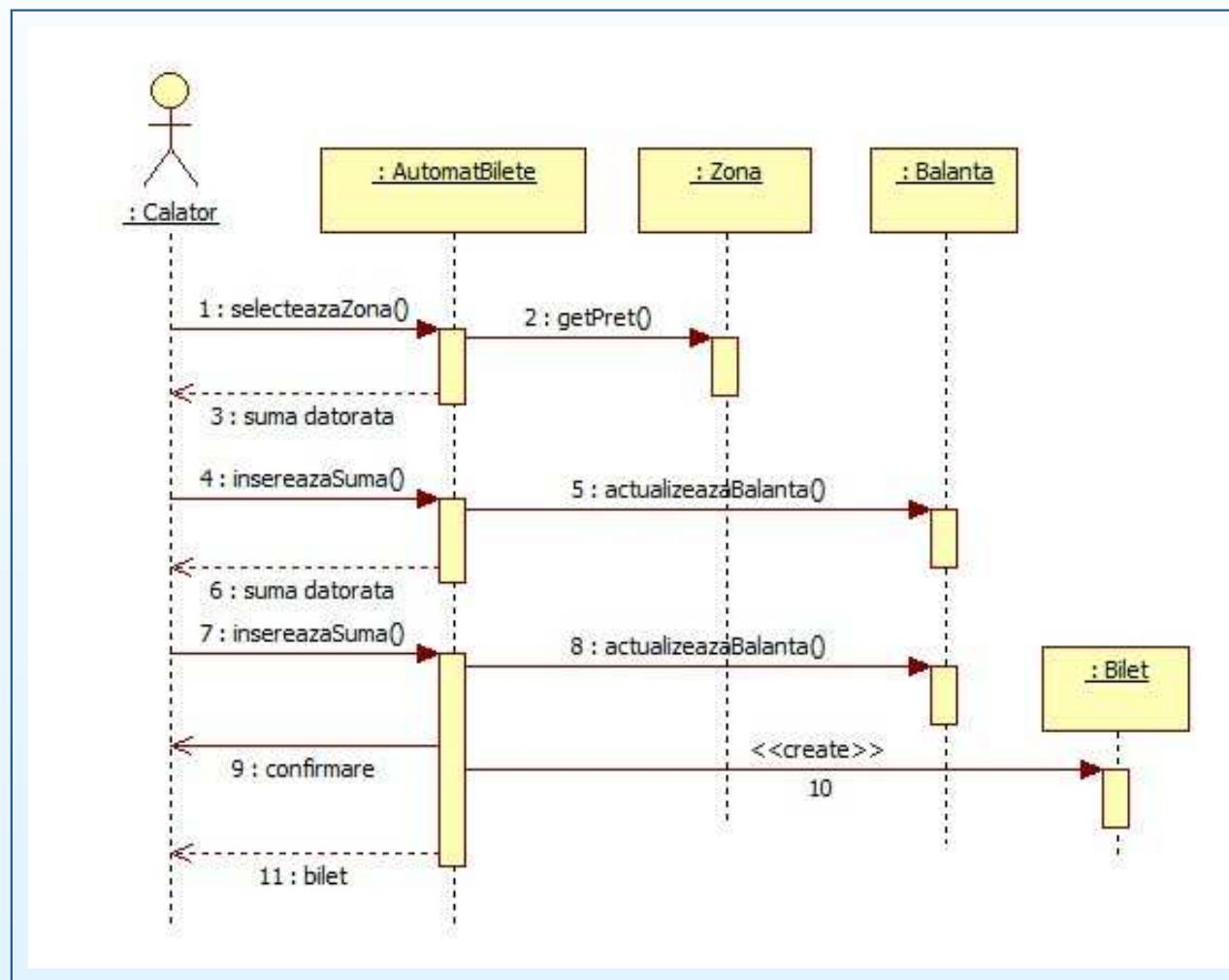
## Analiza cerințelor (cont.)

- Ex.: model conceptual pentru sistemul *AutomatBilete* (reprezentat printr-o diagramă UML de clase)



## Analiza cerințelor (cont.)

- Ex.: model dinamic pentru sistemul *AutomatBilete* (reprezentat printr-o diagramă UML de secvență)

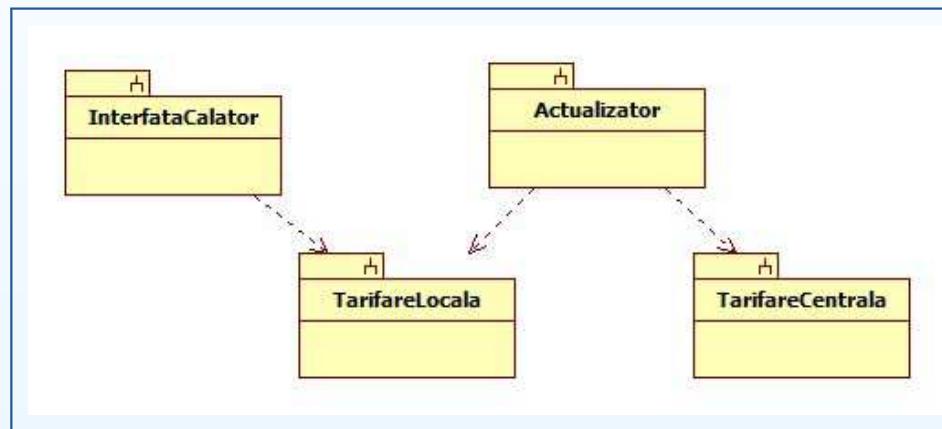


# Proiectarea de sistem

- Scopul acestei etape constă în definirea obiectivelor de proiectare și descompunerea sistemului în subsisteme
  - În cadrul acestei activități, se decide asupra strategiilor utilizate pentru dezvoltarea sistemului
    - platforma hardware/software pe care va rula sistemul
    - modalitatea de asigurare a persistenței datelor
    - fluxul de control global
    - politicile de control a accesului
- Rezultatul proiectării de sistem conține descrierea explicită a strategiilor mai sus menționate, descompunerea sistemului și o *diagramă de repartiție a resurselor* (eng. *deployment diagram*) ilustrând maparea hardware/software aferentă sistemului
- Ca și primă etapă a proiectării, proiectarea de sistem trece din domeniul problemei în domeniul soluției (răspunde întrebării "cum?" și introduce concepte nefamiliale clientului)

## Proiectarea de sistem (cont.)

- Ex.: descompunere a sistemului *AutomatBilete* (diagramă de clase UML - pachetele corespund subsistemelor, liniile punctate reprezintă dependențe)



- Subsistemul *InterfațăCălător* colectează inputul călătorului și asigură feedback (afișare preț bilete, returnare rest, etc.)
- Subsistemul *TarifareLocală* calculează prețul biletelor utilizând o bază de date locală
- Subsistemul *TarifareCentrală* păstrează o copie centralizată a bazei de date cu tarife
- Subsistemul *Actualizator* este responsabil cu actualizarea bazelor de date locale ale automatelor în condițiile schimbării prețului biletelor

# Proiectarea obiectuală

- Scopul proiectării obiectuale este definirea de obiecte/clase din domeniul soluției, pentru a realiza legătura dintre modelul obiectual de analiză și platforma hardware/software stabilită la proiectarea de sistem
- Proiectarea obiectuală include
  - descrierea detaliată a interfețelor obiectelor și subsistemelor
  - selectarea componentelor ce urmează a fi reutilizate
  - restructurarea modelului obiectual pentru a satisface obiectivele de proiectare (generalitate/extensibilitate)
  - optimizarea modelului obiectual în vederea asigurării obiectivelor de performanță
- Rezultatul proiectării obiectuale îl constituie un model obiectual detaliat, îmbogățit cu constrângerii și descrieri precise ale tuturor elementelor componente

# Implementarea și testarea

- Implementarea presupune translatarea modelului obiectual de proiectare (modelul domeniului soluției) în cod sursă
- Testarea presupune identificarea diferențelor între sistemul implementat și modelele sale, ca urmare e execuției sistemului (sau a unor părți ale acestuia) cu seturi de date de test
  - *Testarea unitară* - se compară modelul obiectual de proiectare cu fiecare obiect și subsistem
  - *Testarea de integrare* - se integrează diferite subsisteme, comparându-se cu modelul corespunzător proiectării de sistem
  - *Testarea de sistem* - se compară comportamentul sistemului în ansamblu cu modelul cerințelor
- Scopul testării este identificarea a cât mai multe defecte, permîțând remedierea lor anterior predării sistemului la beneficiar
- Planificarea testelor se face simultan cu dezvoltarea sistemului
  - Testele de sistem se planifică în timpul colectării și analizei cerințelor
  - Testele de integrare se planifică în timpul proiectării de sistem
  - Testele unitare se planifică în timpul proiectării obiectuale

## *Curs 2*

### *Specificarea modelelor folosind UML*

*Suport de curs bazat pe B. Bruegge and A.H. Dutoit*

*"Object-Oriented Software Engineering using UML, Patterns, and Java"*

## Sumar Curs 2

---

- Notații/limbaje. UML
- Tipuri de modele ale sistemelor soft
- Diagrame de cazuri de utilizare
- Diagrame de clase/obiecte
- Diagrame de interacțiune
- Diagrame de tranziție a stărilor
- Diagrame de activități

## Notății / limbaje. UML

- O notăție (*un limbaj*) reprezintă o mulțime de reguli, textuale sau grafice, folosite pentru specificarea modelelor
- În cadrul unui proiect soft, o notăție este un instrument de comunicare (de idei, decizii de analiză, proiectare, etc.)
- Pentru a permite o comunicare eficientă, o notăție trebuie
  - să aibă o semantică bine definită
  - să fie adecvată reprezentării celor aspecte pentru care este folosită
  - să fie bine înțeleasă de către toți participanții la proiect
- *UML (Unified Modeling Language)*
  - limbajul standard adoptat de industrie pentru reprezentarea modelelor orientate obiect
  - a rezultat prin unificarea notățiilor utilizate de metodologiile
    - OMT (Object Modeling Technique - [Rumbaugh et al., 1991])
    - Booch ([Booch, 1994])
    - OOSE (Object Oriented Software Engineering - [Jacobson et al., 1992])
  - oferă un spectru larg de notății pentru reprezentarea diferitor tipuri de sisteme și a diferitor aspecte/vederi ale unui sistem

# Tipuri de modele ale sistemelor soft

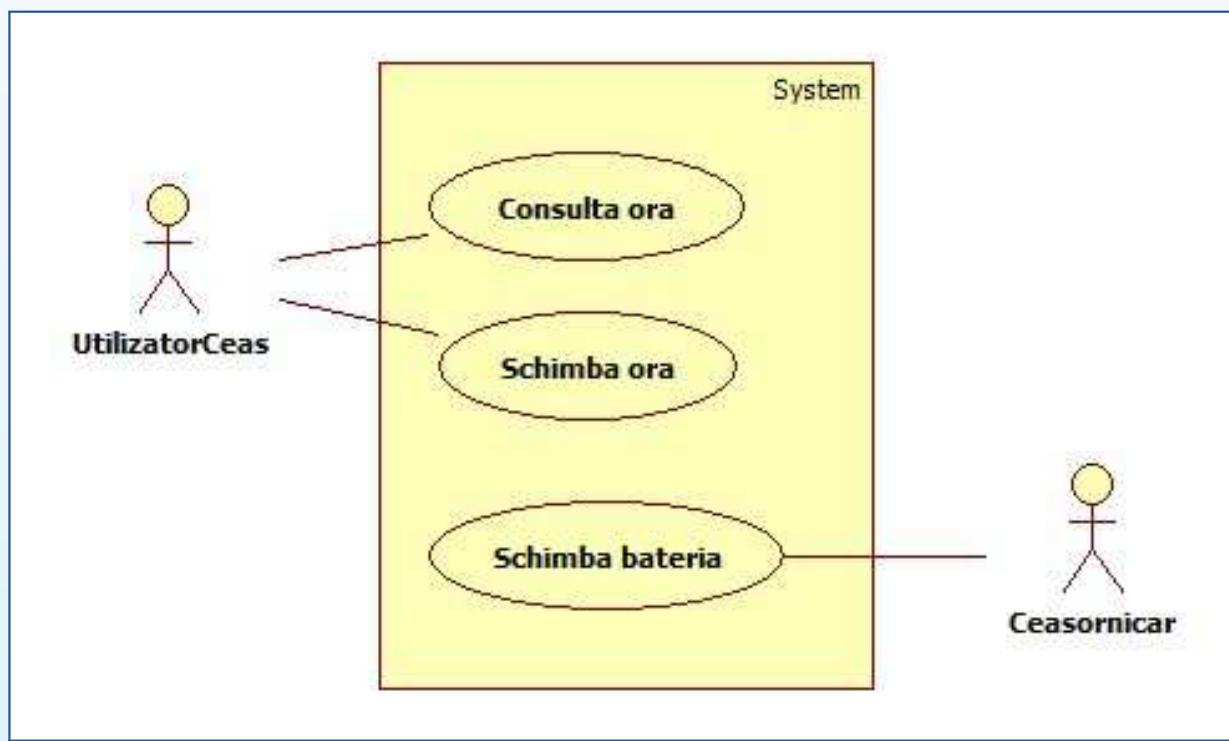
- Modelul funcțional (eng. *functional model*)
  - descrie funcționalitatea sistemului din perspectiva utilizatorului
  - reprezentat în UML folosind *diagrame de cazuri de utilizare*
- Modelul obiectual (structural) (eng. *object model*)
  - descrie structura sistemului în termeni de clase, atribute, asocieri și operații
  - reprezentat în UML folosind *diagrame de clase*
  - evoluția modelului obiectual
    - *modelul obiectual de analiză* sau *modelul conceptual* (eng. *analysis object model*) - descrie conceptele din domeniul problemei relevante pentru sistemul studiat
    - *modelul obiectual corespunzător proiectării de sistem* (eng. *system design object model*) - rafinare a modelului conceptual, ce include descrieri ale interfețelor subsistemelor
    - *modelul obiectual de proiectare* (eng. *object design model*) - rafinare a modelului obiectual aferent proiectării de sistem, incluzând descrierea detaliată a obiectelor din domeniul soluției
- Modelul dinamic (eng. *dynamic model*)
  - descrie comportamentul intern al sistemului
  - reprezentat în UML folosind
    - *diagrame de interacțiune* - secvențe de mesaje schimbate între obiecte
    - *diagrame de tranziție a stărilor* - stările obiectelor și tranzițiile între stări
    - *diagrame de activități* - fluxuri de date și de control

# Diagrame de cazuri de utilizare

- *Cazurile de utilizare* (eng. *use cases*) sunt folosite în cadrul activităților de colectare și analiză a cerințelor, pentru a reprezenta funcționalitățile sistemului
  - Cazurile de utilizare surprind comportamentul sistemului din perspectiva utilizatorilor externi
  - Un caz de utilizare descrie o funcție oferită de către sistem, care are rezultate tangibile pentru un actor
- *Actorii* (eng. *actors*) reprezintă roluri jucate de entități externe sistemului, care interacționează cu acesta
  - utilizatori umani (administrator, client ATM, abonat bibliotecă)
  - alte sisteme soft, echipamente hardware, etc.
- Identificarea actorilor și a cazurilor de utilizare permite definirea *frontierei* sistemului (eng. *system boundary / subject*), mai precis diferențierea între sarcinile îndeplinite de sistem și cele îndeplinite de mediul său
  - Actorii sunt înafara frontierei, cazurile de utilizare sunt înăuntrul acesteia

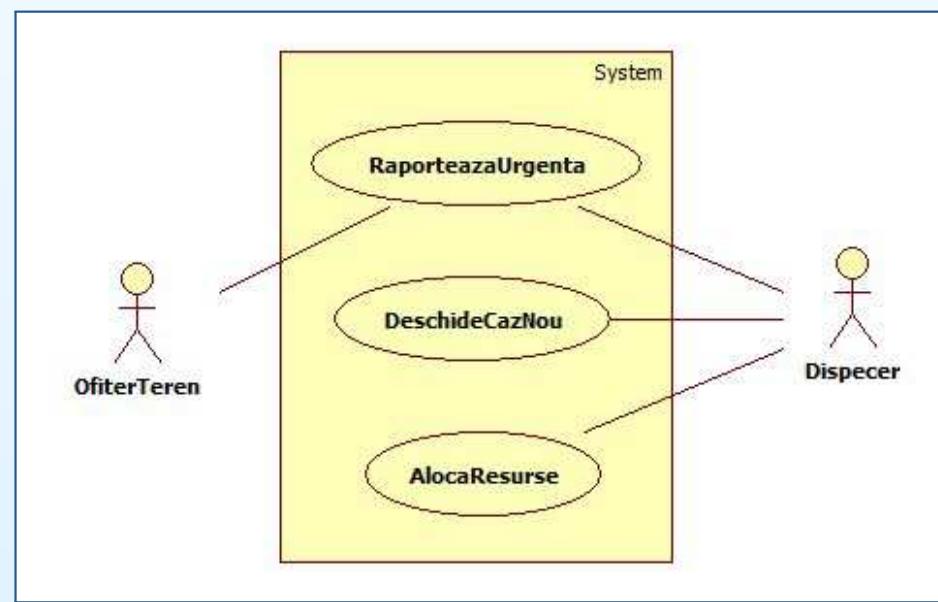
## Diagrame de cazuri de utilizare (cont.)

- Ex.: diagramă a cazurilor de utilizare descriind funcționalitatea unui ceas simplu
  - Sintaxa UML: actor - eng. *stick figure*, caz de utilizare - elipsă, frontieră sistemului - dreptunghi (eng. *box*), etichetate cu denumirile aferente
  - ! Cazurile de utilizare se denumesc cu expresii verbale sugestive pentru funcționalitatea oferită



## Diagrame de cazuri de utilizare (cont.)

- Ex.: diagramă de cazuri de utilizare aferentă unui sistem de gestiune a accidentelor (SGA)
  - Un ofițer din teren (de poliție/pompieri) are posibilitatea invocării cazului de utilizare *RaporteazaUrgenta*, pentru a notifica un dispecer relativ la un nou incident. Ca și răspuns, dispecerul invocă *DeschideCazNou*, pentru a iniția gestiunea incidentului. Dispecerul introduce în baza de date informațiile preliminare primite de la ofițer și alocă resurse (mașini, resursa umană), prin intermediul cazului de utilizare omonim.



## Cazuri de utilizare

- Conținutul unui caz de utilizare poate fi descris textual, folosind un şablon cu următoarele elemente
  - **Nume** - numele cazului de utilizare, unic în sistem
  - **Actori participanți** - actorii care comunică cu acel caz de utilizare
  - **Flux de evenimente** - secvența de interacțiuni între actori și sistem, care definește cazul de utilizare. Fluxul normal și fluxurile alternative (erori, condiții speciale) se vor descrie separat, pentru claritate
  - **Condiții de intrare (precondiții)** - condiții care trebuie satisfăcute anterior inițierii cazului de utilizare
  - **Condiții de ieșire (postcondiții)** - condiții ce trebuie satisfăcute după finalizarea cazului de utilizare
  - **Cerințe de calitate** - constrângeri privind performanța sistemului, implementarea lui, platforma hardware folosită, etc.
- În cadrul şablonului, descrierea cazului de utilizare se face în limbaj natural, susținând comunicarea facilă cu clienții și utilizatorii sistemului

## Cazuri de utilizare (cont.)

- Ex.: Descriere textuală a cazului de utilizare *RaporteazăUrgență*

<b>Nume</b>	<i>RapoteazăUrgență</i>
<b>Actori</b>	<i>Inițiat de OfițerTeren Comunică cu Dispecerul</i>
<b>Flux de evenimente (scenariu normal)</b>	<ol style="list-style-type: none"><li>1. <i>Ofițerul</i> activează funcția <i>Rapotează urgență</i> a terminalului.</li><li>2. Sistemul SGA afișează un formular <i>Ofițerului</i>.</li><li>3. <i>Ofițerul</i> completează formularul, inserând nivelul de alertă, tipul, locația și o scurtă descriere a situației. Propune și posibile soluții la situația de urgență. După completare, <i>Ofițerul</i> trimite formularul.</li><li>4. Sistemul primește formularul și notifică <i>Dispecerul</i>.</li><li>5. <i>Dispecerul</i> consultă informația primită și apelează cazul de utilizare <i>DeschideCazNou</i>. <i>Dispecerul</i> optează pentru una dintre soluțiile propuse și confirmă primirea formularului.</li><li>6. Sistemul afișează confirmarea și soluția aleasă <i>Ofițerului</i>.</li></ol>
<b>Condiții de intrare</b>	<i>Ofițerul</i> este logat în sistem.
<b>Condiții de ieșire</b>	<i>Ofițerul</i> a primit confirmarea de la <i>Dispecer</i> SAU un mesaj referitor la o eroare de comunicare.
<b>Cerințe de calitate</b>	Confirmarea <i>Dispecerului</i> ajunge în maxim 30 de sec. după trimitere.

## Scenarii

---

- Un caz de utilizare este o abstractizare ce acoperă toate scenariile posibile aferente funcționalității descrise
- Un *scenariu* este o instanță a unui caz de utilizare, ce descrie o secvență concretă de acțiuni/evenimente
  - scenariile sunt exemple ce ilustrează situații tipice - sunt focusate pe inteligibilitate
  - cazurile de utilizare sunt folosite pentru a surprinde toate situațiile posibile - sunt focusate pe completitudine
- Scenariile pot fi descrise folosind un şablon cu trei câmpuri
  - **Nume scenariu** - numele scenariului, pentru o referire neambiguă (subliniat, pentru a indica faptul că e o instanță)
  - **Instanțele actorilor participanți** (subliniate)
  - **Fluxul de evenimente**

## Scenarii (cont.)

- Ex.: Scenariul *IncendiereDepozit* al cazului de utilizare *RaporteazăUrgență*

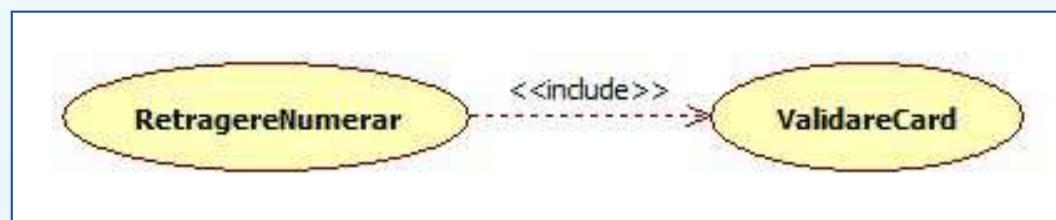
<b>Nume</b>	<u>IncendiereDepozit</u>
<b>Instanțe</b>	<u>bob, alice : OfițerTeren</u>
<b>actori</b>	<u>john : Dispecer</u>
<b>Flux de evenimente</b>	<ol style="list-style-type: none"><li>1. Trecând prin dreptul unui depozit, Bob simte miros de fum. Partenera sa, Alice, activează funcția <i>Raportează urgență</i> pe terminalul SGA.</li><li>2. Alice introduce adresa clădirii, o scurtă descriere a locației curente și un nivel de alertă. Zona fiind aglomerată, solicită o echipă de pompieri și mai multe de medici. Trimit formularul și așteaptă confirmarea dispecerului.</li><li>3. John, dispecerul, este alertat de un semnal sonor al stației sale de lucru. Citește informațiile trimise de Alice și confirmă primirea lor. Alocă o echipă de pompieri și două de medici și ii trimit lui Alice ora estimată a sosirii acestora.</li><li>5. Alice primește confirmarea și estimarea.</li></ol>

# Relații

- Relații posibile în diagramele de cazuri de utilizare
  - comunicare
  - incluziune
  - extindere
  - generalizare
- *Relația de comunicare* (eng. *communication relationship*)
  - un caz de utilizare și un actor comunică atunci când între aceștia există schimb de informație
  - Ex.: Actorii *Dispecer* și *OfițerTeren* comunică cu cazul de utilizare *RaporteazaUrgenta*; doar *Dispecer* comunică cu *DeschideCazNou* și *AlocaResurse*
  - comunicarea dintre un actor și un caz de utilizare se reprezintă ca o asociere UML binară,
- *Relația de incluziune* (eng. *include relationship*)
  - reprezintă o dependență între două cazuri de utilizare, semnificația fiind inserarea comportamentului descris de cazul de utilizare inclus în cadrul comportamentului descris de cazul de utilizare care include/de bază (localizarea exactă a acestei inserări se face la nivelul descrierii textuale a cazului care include)

## Relații (cont.)

- cazul de utilizare inclus nu este optional, execuția sa este obligatorie pentru finalizarea cu succes a cazului care include (analogie cu apelul de subprogram); ca urmare, sensul dependenței este de la cazul care include către cel inclus
- permite reducerea complexității și eliminarea redundanțelor prin factorizarea secvențelor de interacțiuni comune mai multor cazuri de utilizare
- Ex.: În cadrul unui sistem de tip ATM, cazul de utilizare *RetragereNumerar* include cazul de utilizare *ValidareCard* (definit independent)

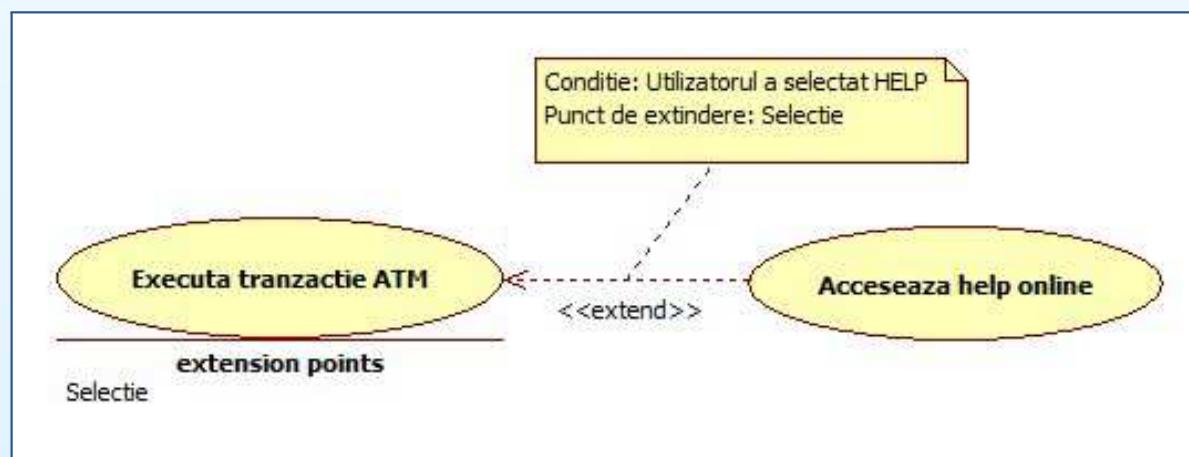


- *Relația de extindere (eng. extend relationship)*

- reprezintă o dependență între două cazuri de utilizare, care precizează *când* (în ce condiții) și *unde* anume poate fi inserat comportamentul aferent cazului de utilizare care extinde în cadrul comportamentului descris de cazul de utilizare extins/de bază

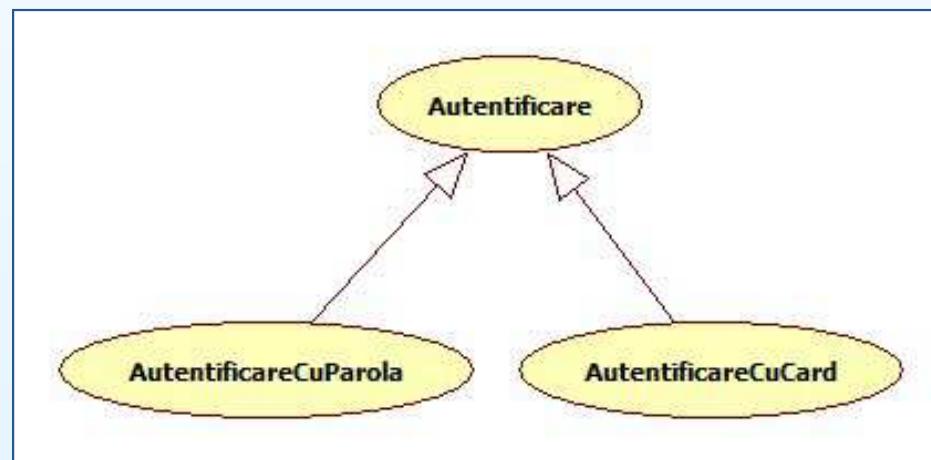
## Relații (cont.)

- cazul de utilizare extins este definit independent și nu depinde de cel care extinde (este de sine stătător)
- cazul de utilizare care extinde poate să nu aibă semnificație de sine stătătoare (poate defini doar un increment comportamental care extinde unul sau mai multe alte cazuri de utilizare, în anumite condiții)
- sensul dependenței este de la cazul care extinde către cel extins
- extinderea are loc la unul sau mai multe puncte de extindere definite în cazul de utilizare de bază (prin nume și variante de localizare)
- Ex.: Cazul de utilizare *Accesează help online* extinde cazul *Execută tranzacție ATM*



## Relații (cont.)

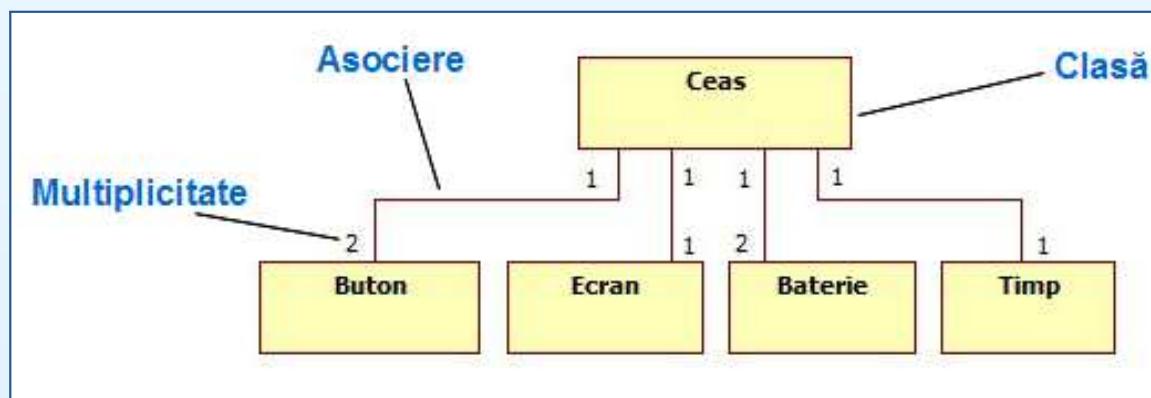
- *Relația de generalizare* (eng. *generalization relationship*)
  - între cazuri de utilizare sau între actori, cu semantica uzuală
  - un caz de utilizare/actor poate specializa un altul, mai general, prin adăugarea de detalii suplimentare
  - Ex.: un caz de utilizare *Autentificare*, identificat inițial în faza de analiză a cerințelor, poate fi ulterior rafinat, rezultând două noi cazuri de utilizare *AutentificareCuParola* și *AutentificareCuCard*, specializări ale primului



- În reprezentarea textuală, cazurile specializate moștenesc actorul ce inițiază interacțiunea, precum și condițiile de intrare și ieșire de la cazul general

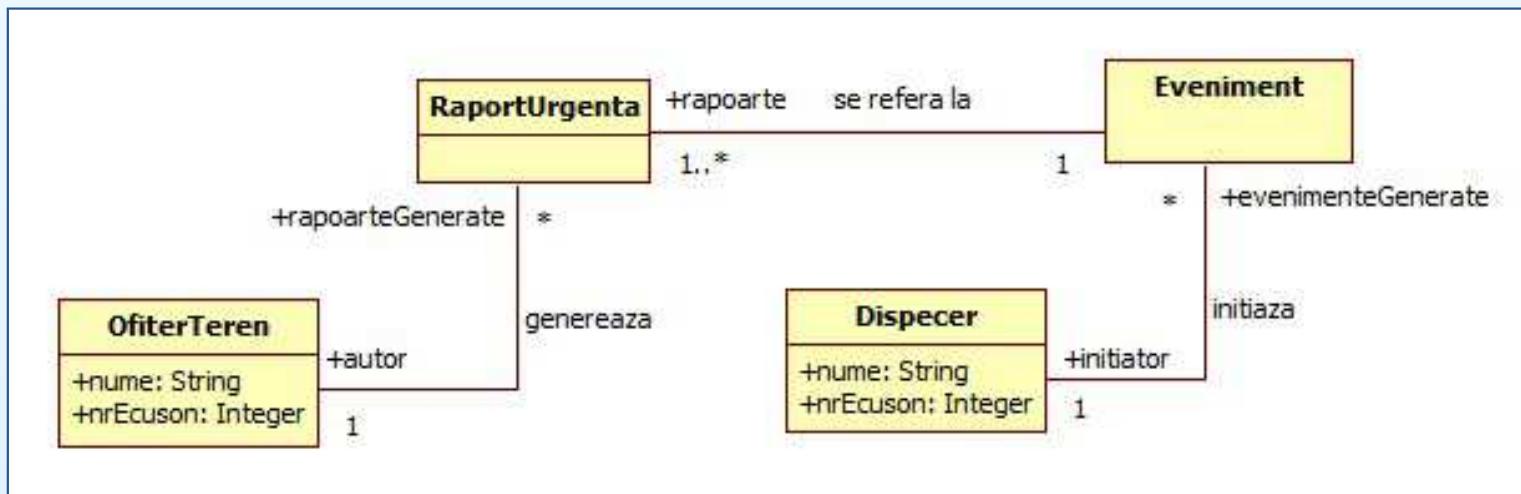
# Diagrame de clase/obiecte

- *Diagramele de clase/obiecte* (eng. *class/object diagrams*) sunt utilizate pentru descrierea structurii unui sistem
  - Clasele sunt abstractizări ce specifică structura și comportamentul comune unor mulțimi de obiecte
  - Obiectele sunt instanțe ale claselor, create, modificate și distruse pe parcursul execuției unui sistem
    - Un obiect este caracterizat prin *starea sa*, ce include valorile atributelor sale și legăturile cu alte obiecte
    - Fiecare obiect are o *identitate* - poate fi referit individual și diferențiat de alte obiecte
- Ex.: diagramă de clase descriind structura unui ceas simplu



# Clase și obiecte

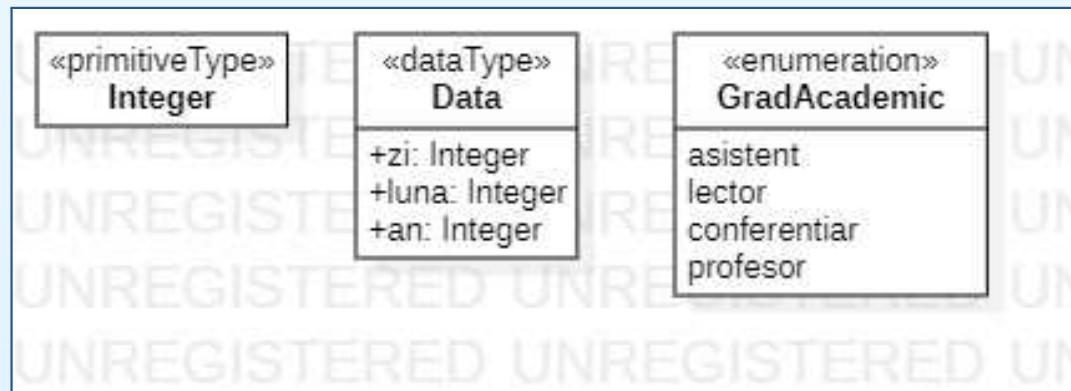
- *Clase* (eng. *classes*)
  - Sunt reprezentate în UML folosind dreptunghiuri cu 3 compartimente (primul compartiment - numele, al doilea - atributele, ultimul - operațiile)
  - Compartimentele pentru atrbute și operații pot fi omise
  - Convenție: numele unei clase este un substantiv la singular, prima literă fiind majusculă
  - Atributele sunt caracterizate prin *tip*, operațiile prin *signură*
- Ex.: diagramă de clase ilustrând entitățile implicate în cazul de utilizare *RaporteazăUrgență* și asocierile între acestea



# Clase și obiecte

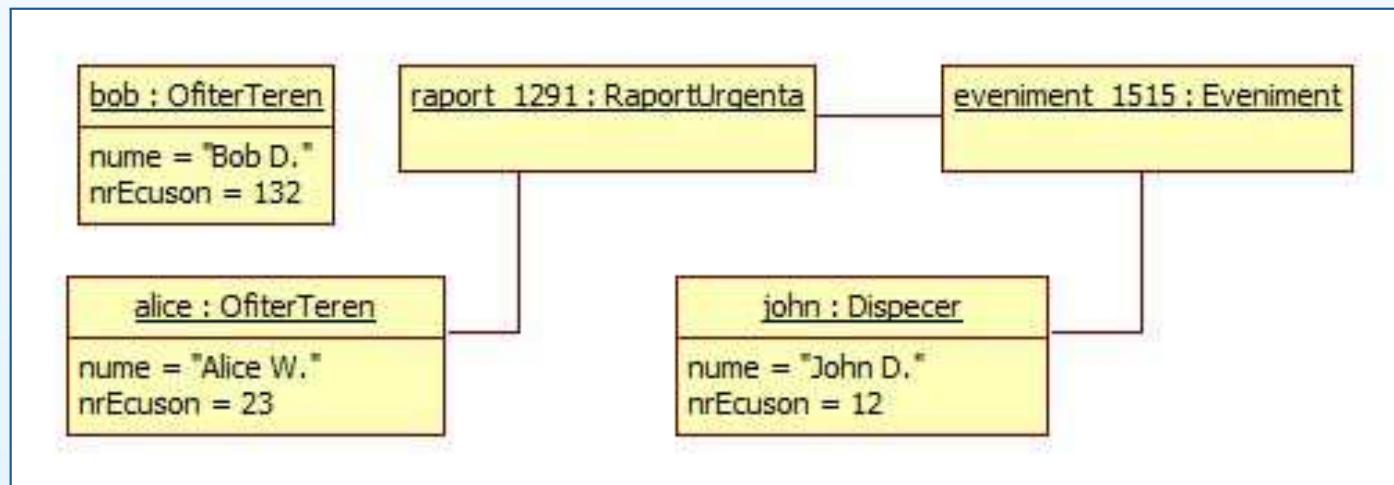
## • Data Type

- Spre deosebire de instanțele unei clase, instanțele unui *data type* sunt identificate doar prin valoare (nu au identitate proprie, toate instanțele cu aceeași valoare sunt considerate identice)
- Clasificare
  - primitive (nestructurate) - stereotip «primitive»
    - Primitive UML predefinite: Integer, UnlimitedNatural, Real, String, Boolean
  - enumerări (mulțimea valorilor definită ca o listă de literali) - stereotip «enumeration»
  - structurate (au atribută) - stereotip «dataType»



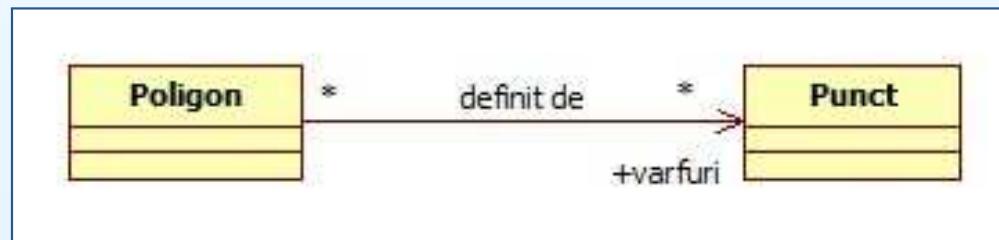
## Clase și obiecte (cont.)

- *Obiecte* (eng. *objects*)
  - Pot primi nume în diagramele de obiecte (pentru a ușura referirea lor), sau pot fi *anonyme* (se precizează doar clasa corespunzătoare)
  - Convenție: numele obiectelor se scriu cu litere mici
  - Se precizează valorile atributelor aferente *sloturilor* obiectelor
- Ex.: diagramă de obiecte (eng. *object diagram*) ilustrând obiectele implicate în scenariul *IncendiereDepozit* și legăturile între acestea



# Asocieri și legături

- O legătură (eng. *link*) reprezintă o conexiune între două obiecte
- Asocierile (eng. *associations*) sunt relații între clase și reprezintă mulțimi de legături
  - Asocierile pot fi uni/bi-direcționale sau cu navigabilitate nespecificată
    - Asocierile unidirecționale indică navigabilitate într-un singur sens - cel precizat de săgeată
    - Asocierile bidirecționale indică navigabilitate în ambele sensuri - se pot reprezenta ambele săgeți, însă, de obicei, se omit
- Ex.: Asociere unidirectională între clasele *Poligon* și *Punct*

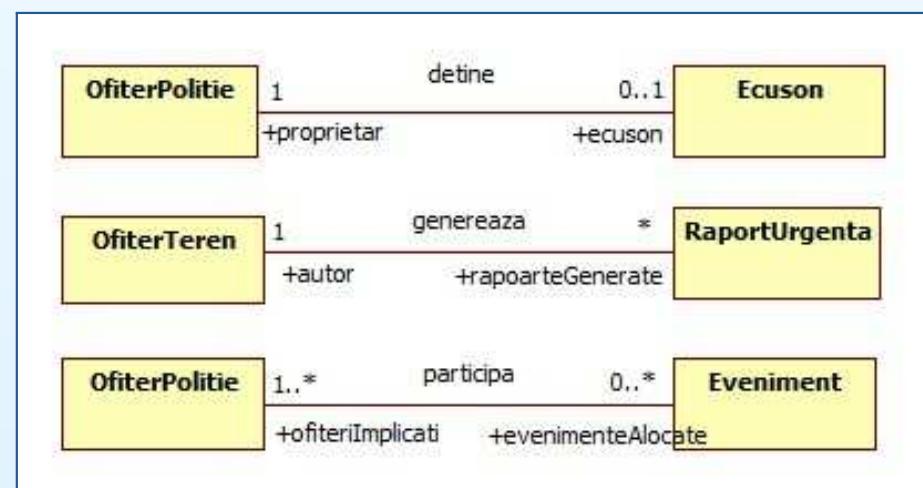


- Conform săgeții, se permite doar navigarea dinspre poligon înspre punct
  - Dat fiind un poligon, se poate interoga colecția punctelor ce definesc acel poligon
  - Dat fiind un punct, NU se pot afla (direct) poligoanele din care face parte

# Multiplicități și roluri

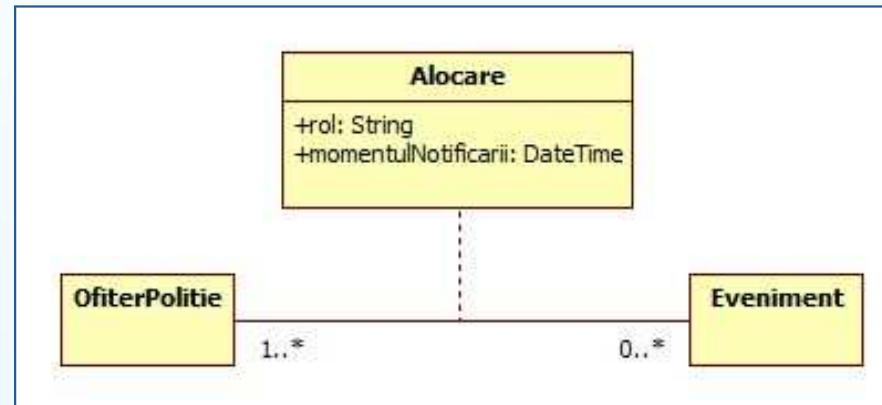
- Fiecare dintre capetele unei asocieri poate fi etichetat cu un *nume de rol* (eng. *role*)
  - Rolurile clarifică scopul asocierii și permit diferențierea între asocieri diferite ce conectează aceeași clase
- Fiecare capăt al unei asocieri poate fi etichetat de o mulțime de întregi = *multiplicitatea capătului repectiv* (eng. *multiplicity*)
  - Multiplicitatea unui capăt de asociere indică numărul de legături pe care o instanță a clasei de la capătul opus îl poate avea cu instanțe ale clasei de la acel capăt
  - Tipuri uzuale de asocieri

- eng. *one-to-one*  
*one* = 1 or 0..1
- eng. *one-to-many*  
*many* = 1..\*, 0..\* or \*
- *many-to-many*

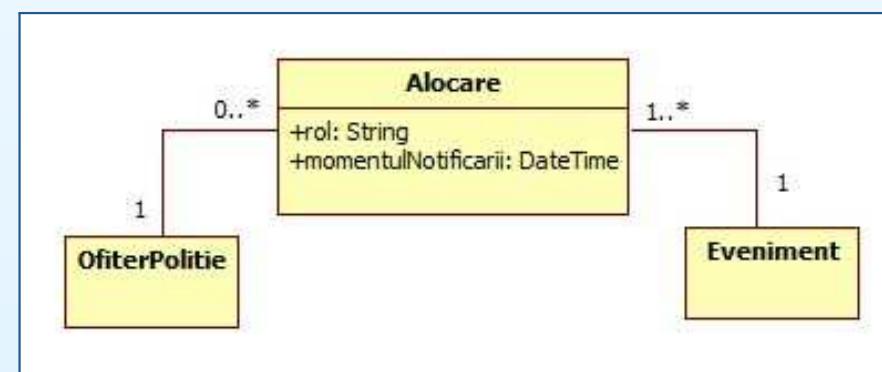


# Clase asociere

- Similar claselor, asocierilor li se pot ataşa atribute şi operaţii => **clase asociere** (eng. *association classes*)
  - Sintaxă



- Clasă asociere  $\Leftrightarrow$  clasă + asocieri simple + constrângere de unicitate !

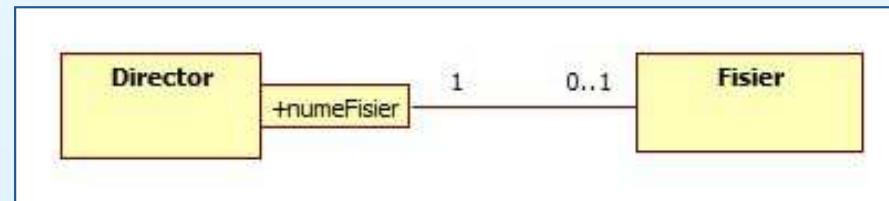


# Asocieri calificate

- *Calificarea* (eng. *qualification*) este o metodă de reducere a multiplicităților, prin utilizarea cheilor (= attribute ce oferă identificare unică)
  - Utilizarea asocierilor calificate simplifică și mărește claritatea diagramelor
- Ex.:
  - Inițial: un director conține mai multe fișiere, fiecare fișier fiind identificat în mod unic în cadrul directorului prin numele său

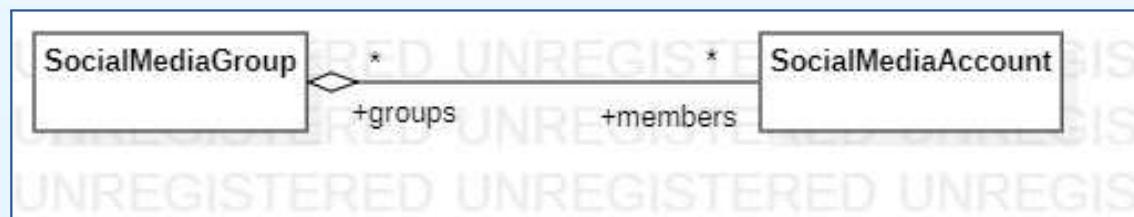


- Reducerea multiplicității asocierii (introducerea unei asocieri calificate) prin alegerea atributului aferent numelui de fișier ca și calificator (cheie)



# Agregare

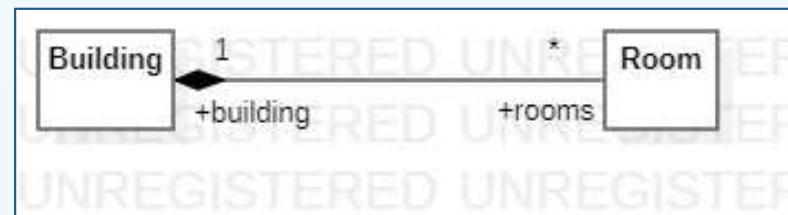
- Reprezintă un caz particular de asociere, care denotă o relație de tip parte - întreg
  - Se traduce prin: *are* (eng. *has*), *constă din*, *este format din*
  - Sintactic, se reprezintă folosind un romb, pe capătul dinspre întreg
- Variante
  - *agregare partajată* (eng. *shared aggregation*)
    - apartenență slabă a părților la întreg, părțile pot exista și independent
    - multiplicitatea capătului dinspre întreg poate fi mai mare decât 1 (partea poate apartine mai multor întregi simultan)
    - sintactic, romb gol (engl. *hollow diamond*)



## Agregare (cont.)

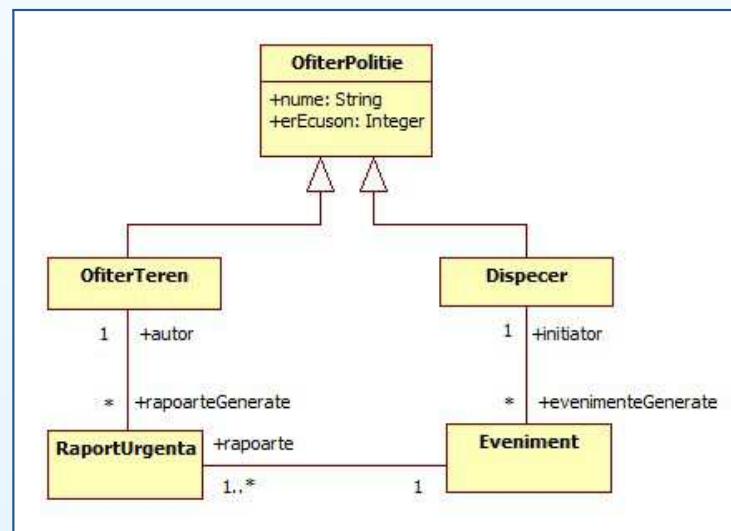
- *componere* (eng. *composition*)

- o parte poate fi conținută în cel mult un întreg la un moment dat (multiplicitatea maximă pe capătul dinspre întreg e 1)
- întregul controlează durata de viață a părților (distrugere întregului rezultă în distrugerea părților)
- sintactic, romb plin (engl. *solid diamond*)



# Generalizare / specializare

- Generalizarea este o relație care permite factorizarea atributelor și operațiilor (stării și comportamentului) comune unei mulțimi de clase
  - Clasele *derivate* sau *subclasele* moștenesc attributele și operațiile clasei de bază sau *superclasei*



- UML distinge între conceptul de *operație* și cel de *metodă*
  - operație - specificarea comportamentului
  - metodă - implementarea comportamentului

# Utilizarea diagramelor de clase

- În etapa de *analiză a cerințelor*
  - Permit formalizarea cunoștințelor legate de domeniul problemei
  - Clasele corespund entităților din domeniul problemei, iar asocierile relațiilor stabilite între acestea
  - Stabilirea tipurilor atributelor și a signaturilor operațiilor poate fi amânată pentru etapa de proiectare, la fel și deciziile privind navigabilitatea
- În *proiectarea de sistem și obiectuală*
  - Diagramele de clase din analiză sunt rafinate, prin introducerea claselor corespunzătoare entităților din domeniul soluției
  - Clasele sunt grupate în subsisteme cu interfețe bine definite

# Diagrame de interacțiune

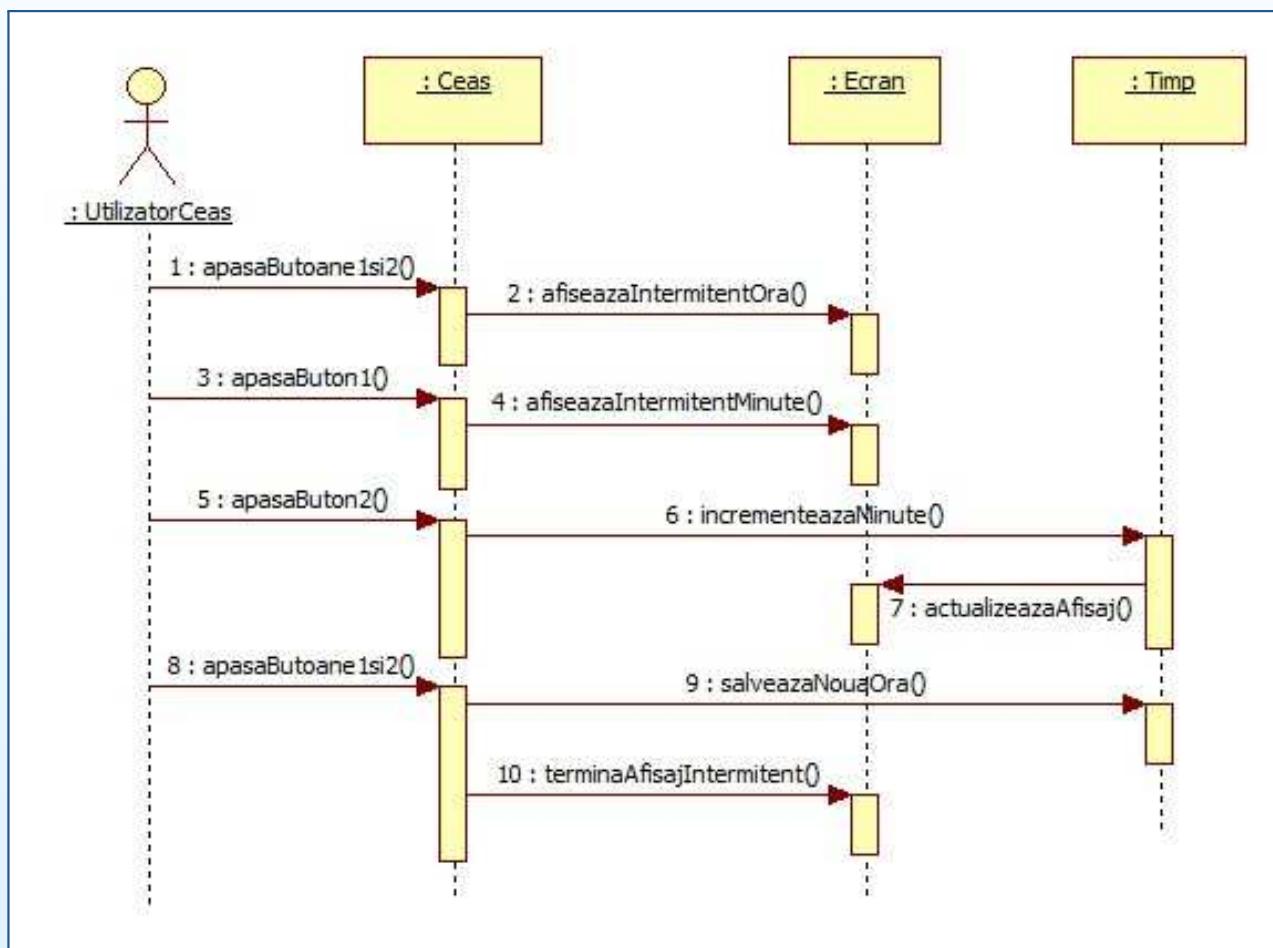
- *Diagramele de interacțiune* (eng. *interaction diagrams*) descriu şabloane de comunicare într-o mulțime de obiecte care interacționează
  - Obiectele interacționează între ele prin schimb de *mesaje*
  - Recepționarea unui mesaj de către un obiect declanșează execuția unei *metode* a obiectului în cauză, fapt ce determină, de obicei, trimitera unor mesaje către alte obiecte
  - Un mesaj trimis unui obiect poate avea *argumente* asociate - acestea corespund parametrilor metodei aferente a obiectului destinație
- Tipuri de diagrame de interacțiune
  - Diagrame de secvență (eng. *sequence diagrams*)
  - Diagrame de comunicare (eng. *communication diagrams*)
    - Cele două tipuri de diagrame de interacțiune sunt echivalente: dată fiind o diagramă de secvență, se poate construi diagrama de colaborare echivalentă și reciproc

## Exemplu: ceas electronic

- Considerăm exemplul unui ceas electronic simplu, cu două butoane. Pentru a modifica timpul curent, utilizatorul trebuie să apese simultan cele două butoane, moment în care se intră în modul de lucru *Setare timp*. În acest mod de lucru, ceasul afișează intermitent componenta modificată (oră, minut, secundă, zi, lună sau an). Imediat după intrarea în modul *Setare timp*, va fi afișată intermitent ora. În cazul în care utilizatorul apasă primul buton, va fi afișată intermitent următoarea componentă. Dacă este apăsat al doilea buton, componenta curentă afișată intermitent va fi incrementată cu o unitate (atingerea valorii finale a intervalului aferent determină resetarea la valoarea inițială a componentei respective). Modul *Setare timp* poate fi părăsit prin apăsarea simultană a ambelor butoane.

# Diagrame de secvență

- Ex.: diagramă de secvență corespunzătoare incrementării minutelor curente ale unui ceas electronic cu o unitate (scenariu aferent cazului de utilizare *Schimba ora*)

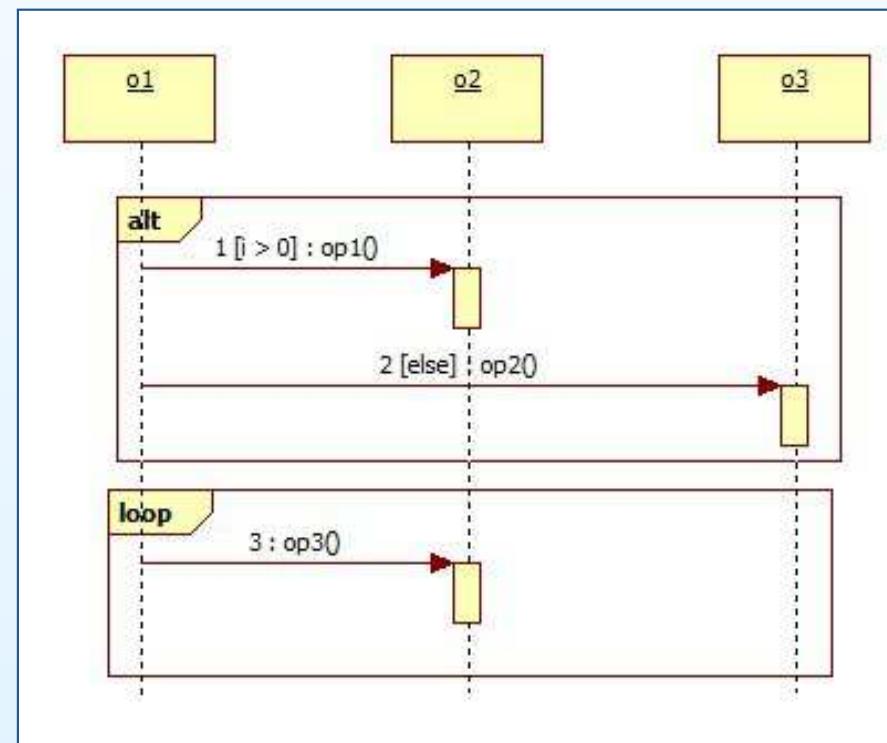


## Diagrame de secvență (cont.)

- Primează perspectiva temporală
- Obiectele participante la interacțiune se reprezintă pe orizontală, iar timpul pe verticală
  - Fiecare coloană corespunde unui obiect participant la interacțiune
  - Coloana cea mai din stânga corespunde unei instanțe a actorului care declanșează interacțiunea
- Transmiterea mesajelor este reprezentată prin săgeți etichetate
  - Etichetele indică numele mesajelor și eventualele argumente
- Activarea (execuția unei metode) este reprezentată prin dreptunghiuri verticale
- Mesajele inițiate de către actori corespund unor interacțiuni descrise în cazul de utilizare aferent
  - Deși, pentru simplitate, interacțiunile dintre obiecte și cele între sistem și actori sunt reprezentate uniform prin mesaje, cele două tipuri de interacțiune sunt de natură diferită!

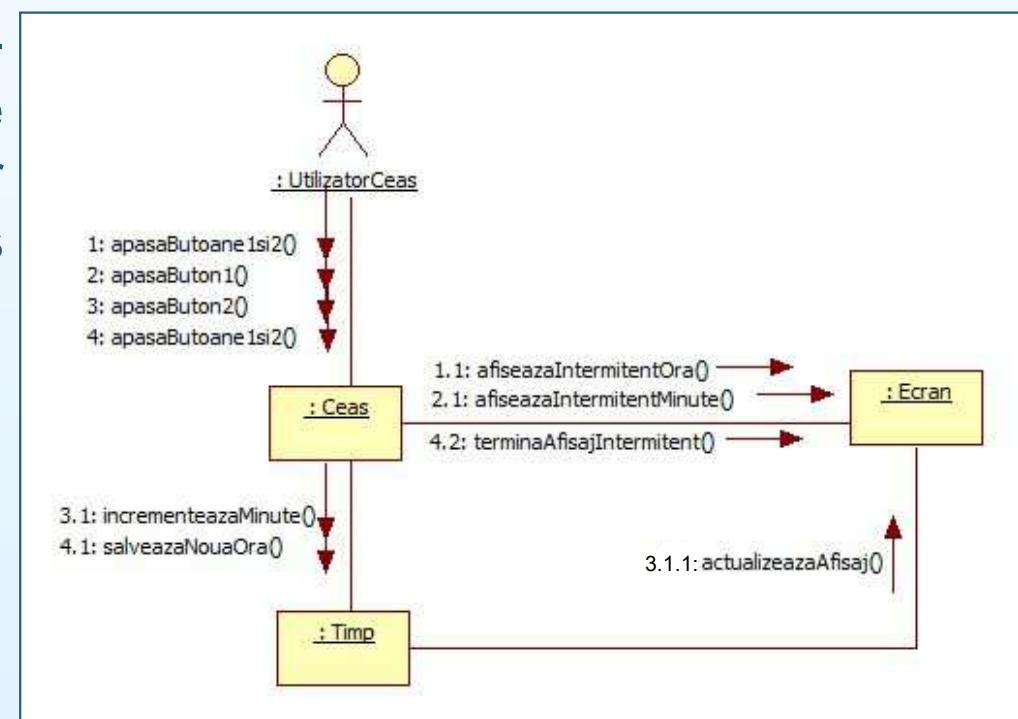
## Diagrame de secvență (cont.)

- Diagramele de secvență pot fi utilizate pentru descrierea unor interacțiuni specifice (scenarii) sau a unora generale (cazuri de utilizare)
  - Corespunzător celei de-a doua situații, există notații specifice pentru condiționări și cicluri (eng. *combined fragments*)



# Diagrame de comunicare

- Ilustrează aceeași informație ca și diagramele de secvență, însă cu accent pe colaborările între obiectele participante la interacțiune (în diagramele de secvență accentul e pe secvențierea mesajelor în timp)
  - Avantaje: aspect compact al diagramei
  - Dezavantaje: secvențierea mesajelor e dificil de urmărit
- Ex.: diagramă de comunicare corespunzătoare incrementării minutelor curente ale unui ceas electronic cu o unitate



# Utilizarea diagramelor de interacțiune

- Motivul principal al construirii diagramelor de interacțiune îl reprezintă identificarea responsabilităților claselor existente deja în diagrama de clase, precum și identificarea de noi clase
- În mod uzual, se realizează cel putin câte o diagramă de interacțiune pentru fiecare caz de utilizare, axată pe fluxul normal de evenimente + diagrame aferente fluxurilor de excepție
  - Sunt identificate obiectele care participă la cazul respectiv de utilizare și se atribuie fragmente din comportamentul ce definește cazul de utilizare acestor obiecte, sub forma operațiilor
- După definirea diagramei inițiale de clase, aceasta și diagramele de interacțiune se dezvoltă în tandem
  - Acet proces are ca și efect rafinarea cazurilor de utilizare (rezolvarea ambiguităților, adăugarea unor noi elemente de comportament), cu introducerea de noi obiecte și servicii

# Diagrame de tranziție a stărilor

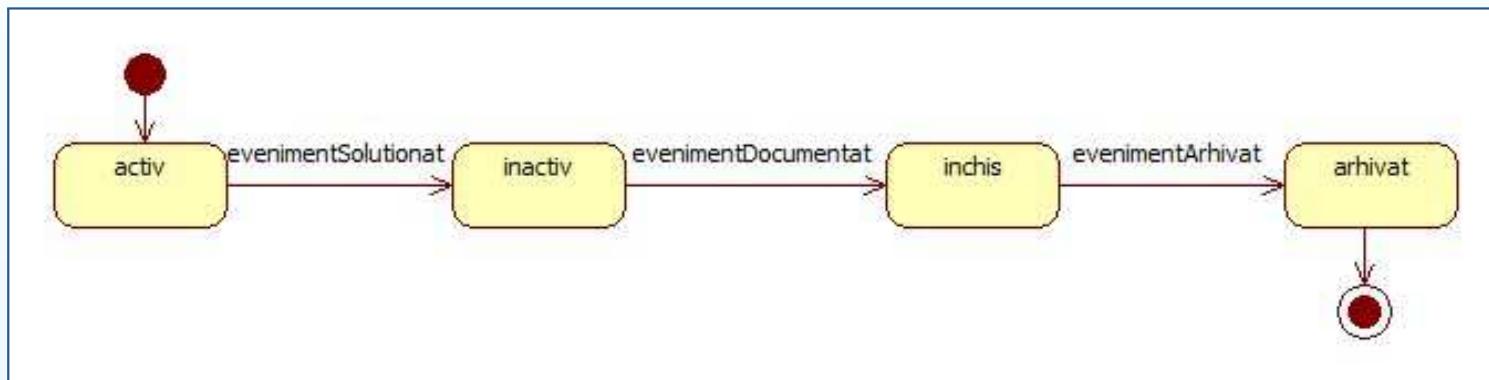
- O mașină cu stări UML (eng. *UML state machine*) reprezintă o notație folosită pentru a descrie succesiunea de stări prin care trece un obiect sub acțiunea evenimentelor externe
- Originea *diagramelor de tranziție a stărilor UML* (eng. *UML state machine diagrams*) este reprezentată de teoria automatelor finite, extinsă cu
  - o notație pentru imbricarea stărilor și a mașinilor cu stări (o stare poate fi descrisă prin intermediul unei mașini cu stări)
  - o notație ce permite etichetarea tranzițiilor cu mesaje trimise și condiții impuse asupra obiectelor
- Mașinile cu stări UML sunt bazate în principal pe diagramele de stări (eng. *statecharts*) introduse de Harel [Harel, 1987]

## Stări și tranziții

- O *stare* (eng. *state*) reprezintă o condiție satisfăcută de valorile atributelor unui obiect
  - Ex.: un obiect de tip *Eveniment* din sistemul SGA poate fi în una din patru stări posibile:
    - *activ* - denotă o situație care necesită a fi soluționată (ex.: un incendiu, un accident rutier)
    - *inactiv* - denotă o situație care a fost rezolvată, dar nu a fost încă documentată corespunzător (ex.: incendiul a fost stins, dar nu au fost estimate încă pagubele)
    - *închis* - denotă o situație care a fost rezolvată și documentată corespunzător
    - *arhivat* - denotă un eveniment închis a cărui documentație a fost arhivată
  - Aceste patru stări pot fi reprezentate adăugând în clasa *Eveniment* un atribut *status*, ce poate avea, la un moment dat, una din aceste patru valori: *activ*, *inactiv*, *închis*, *arhivat*
- O *tranziție* (eng. *transition*) reprezintă o schimbare de stare, care poate fi provocată de declanșarea unor evenimente, îndeplinirea unor condiții sau de trecerea unui anumit interval de timp

## Stări și tranziții (cont.)

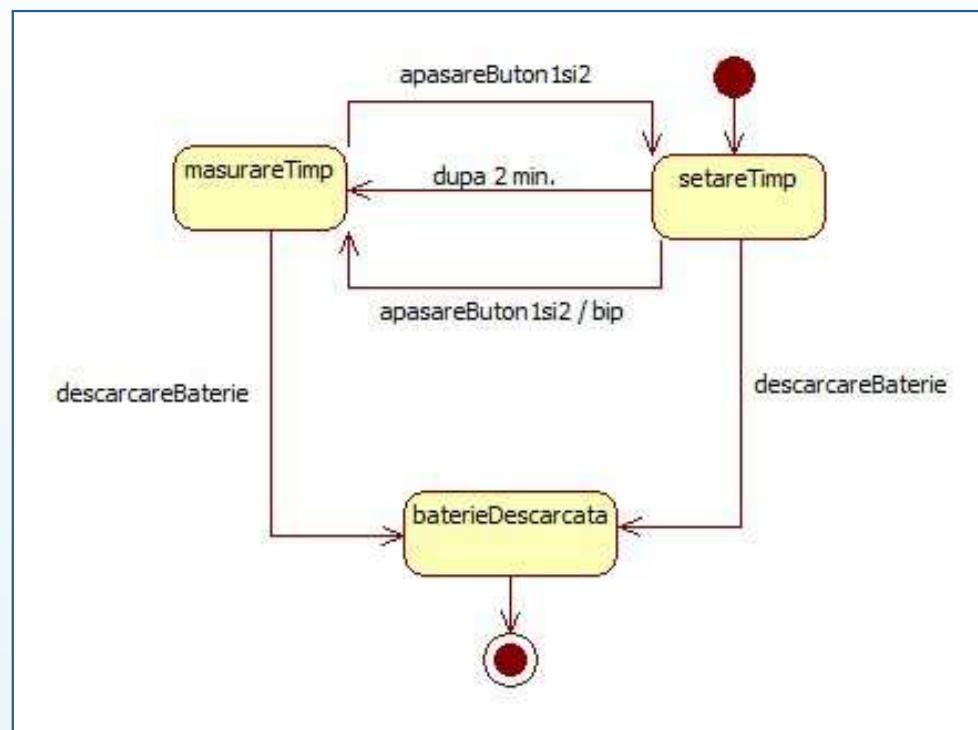
- Ex.: diagramă de tranziție a stărilor UML aferentă clasei *Eveniment*



- Sintactic:
  - O stare se reprezintă ca un dreptunghi cu colțuri rotunjite
  - O tranziție se reprezintă ca o săgeată ce unește două stări
  - Stările sunt etichetate cu numele lor
  - Tranzițiile pot fi etichetate cu numele evenimentelor care le declanșează
  - Un disc denotă (pseudo)starea inițială
  - Un disc încercuit denotă o (pseudo)stare finală

## Stări și tranziții (cont.)

- Ex.: diagramă de tranziție a stărilor UML aferentă clasei Ceas



- Tranzitia de la starea *setareTimp* la starea *măsurareTimp* poate fi declanșată de un eveniment (*apasareButon1si2*) sau de trecerea timpului (2 min.)
- Tranzitia declanșată de eveniment de la starea *setareTimp* la starea *măsurareTimp* are asociată o acțiune (semnal sonor *bip*)

# Acțiuni, tranziții interne și activități

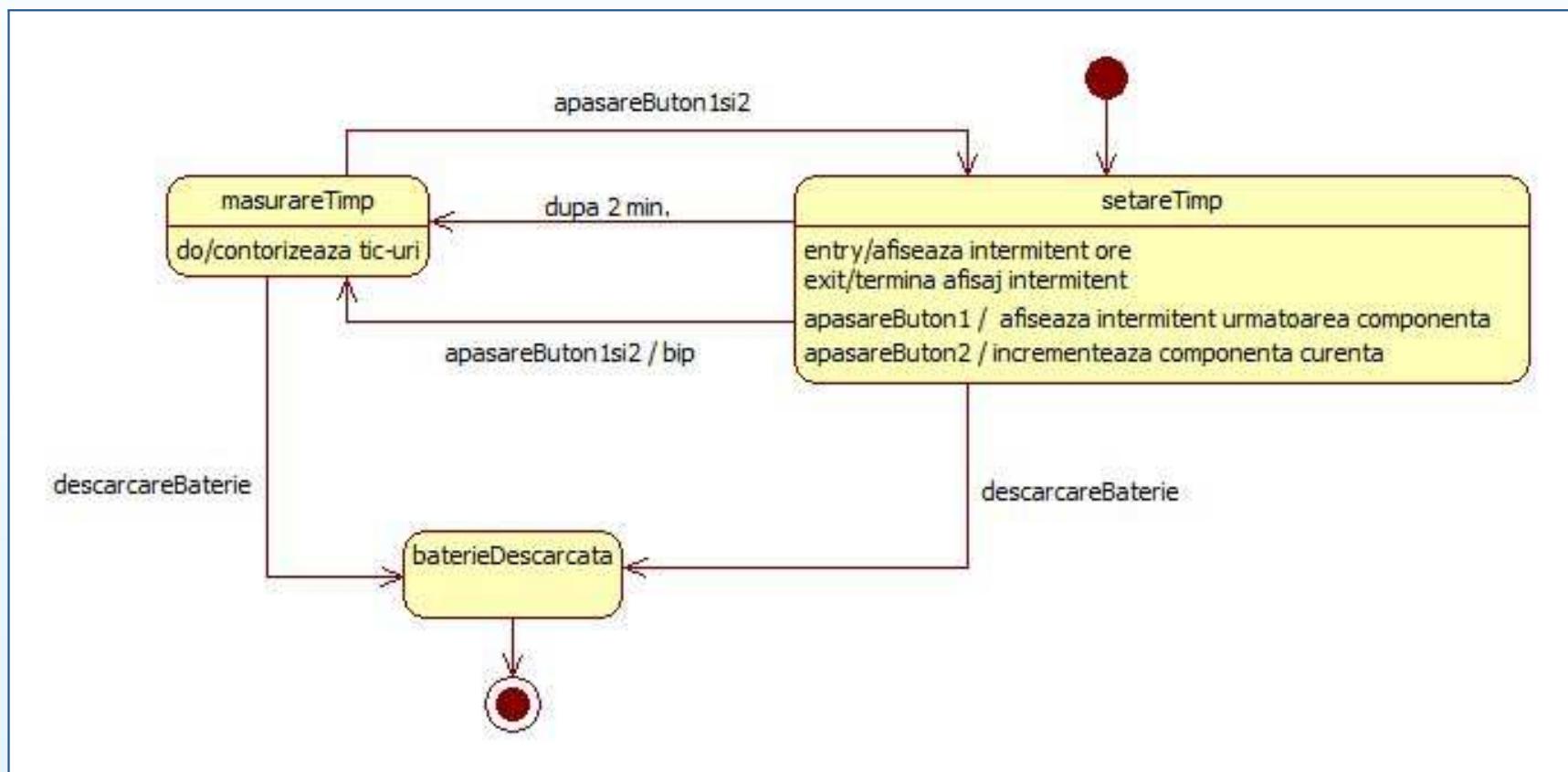
- O acțiune (eng. *action*) reprezintă o unitate fundamentală de procesare, care poate primi input-uri, poate produce output-uri și poate schimba starea sistemului
  - Acțiunile sunt considerate atomice (se execută într-un timp scurt și nu pot fi întrerupte)
  - Ex. de acțiune: apelul unei operații
  - Într-o mașină cu stări, acțiunile pot fi localizate:
    - la nivelul unei tranziții (ex.: acțiunea *bip*, asociată tranziției de la starea *setareTimp* la starea *măsurareTimp*)
    - la intrarea într-o stare, introduse prin eticheta *entry* (ex.: acțiunea *afiseaza intermitent ore*, la intrarea în starea *setareTimp*)
    - la ieșirea dintr-o stare, introduse prin eticheta *exit* (ex. acțiunea *termina afisaj intermitent*, la ieșire adin starea *setareTimp*)
  - În timpul unei tranziții, se execută mai întâi acțiunile de ieșire din starea sursă, apoi acțiunile asociate tranziției, iar la final acțiunile de intrarea ale stării destinație
  - Acțiunile de intrare/ieșire se execută ori de câte ori se intră în / iese din starea respectivă, indiferent de tranziția implicată în schimbarea de stare

## Acțiuni, tranziții interne și activități (cont.)

- O *tranzitie internă* (eng. *internal transition*) este o tranziție ce nu determină părăsirea stării curente
  - Tranzitiile interne sunt declanșate de evenimente și pot avea acțiuni asociate
    - Ex.: tranzitiile interne din starea *setareTimp*, declanșate de evenimentele *apasareButon1* și *apasareButon2*
  - Declanșarea unei tranzitii interne nu determină execuția acțiunilor de intrare/ieșire din stare
- O *activitate* (eng. *activity*) reprezintă o mulțime coordonată de acțiuni
  - O stare poate avea o activitate asociată, care se execută atâta timp cât obiectul rămâne în acea stare
  - Spre deosebire de acțiuni, care sunt atomice, activitățile durează mai mult timp și pot fi întrerupte de ieșirea obiectului din starea curentă
  - Activitățile sunt introduse prin eticheta */do* și plasate în starea în care se execută
    - Ex.: activitatea *contorizează tic-uri* din starea *masurareTimp*

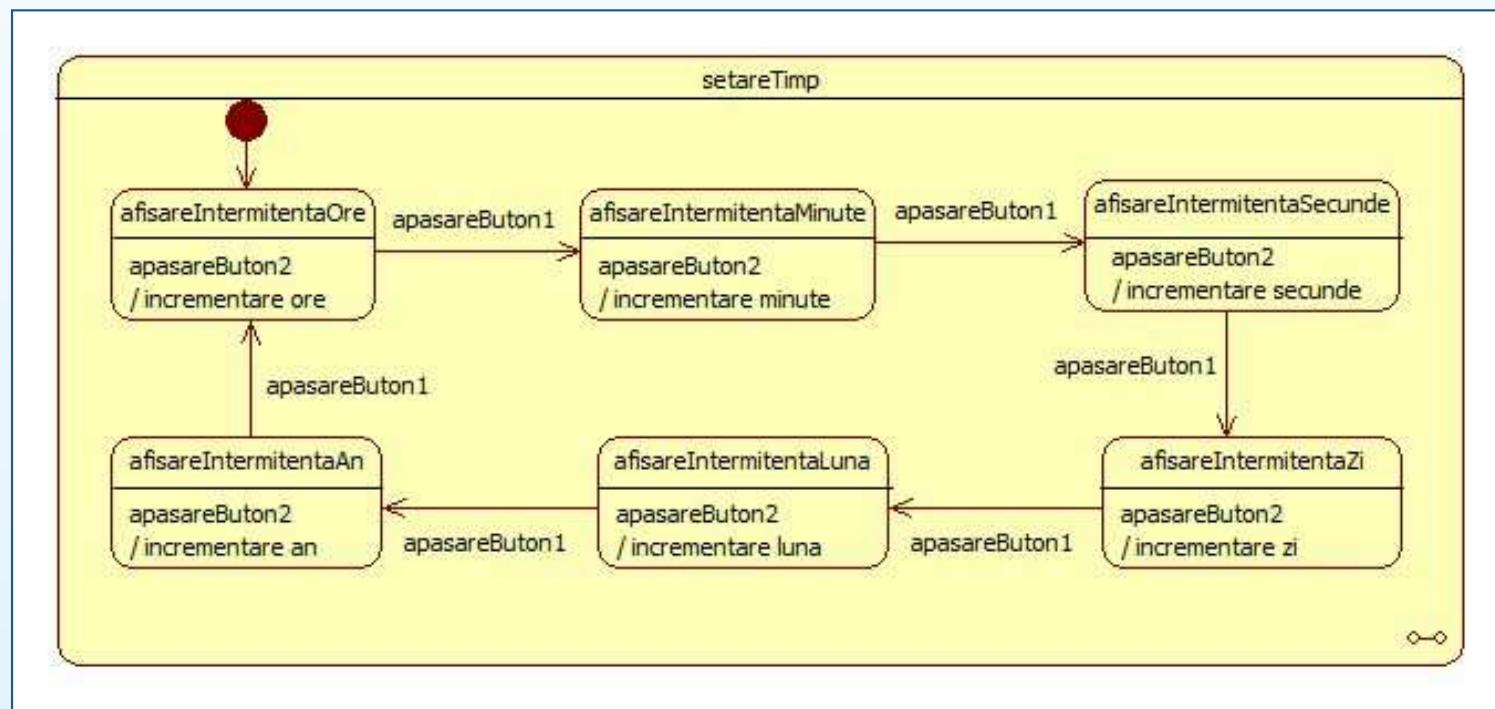
## Acțiuni, tranziții interne și activități (cont.)

- Ex.: Rafinare a diagramei de tranziție a stărilor aferente clasei Ceas, cu ilustrarea unor acțiuni, tranziții interne și activități



# Imbricarea mașinilor cu stări

- Reprezintă o alternativă la folosirea tranzițiilor interne
- Crește inteligibilitatea și permite reducerea complexității diagramelor
- Ex.: Rafinare a stării *setareTimp* prin eliminarea/relocarea tranzițiilor interne și imbricarea unei submașini cu stări



- Fiecare dintre tranzițiile interne ale substărilor ar putea fi reprezentată, într-o rafinare ulterioară, ca o submașină cu stări

## Utilizarea diagramelor de tranziție a stărilor

---

- Diagramele de tranziție a stărilor sunt folosite pentru modelarea comportamentului netrivial al obiectelor sau subsistemelor individuale
- În etapa de analiză, utilizarea lor ajută la identificarea atributelor obiectelor din domeniul problemei și la rafinarea descrierii comportamentului acestora
- În etapa de proiectare, pot fi folosite pentru descrierea obiectelor din domeniul soluției care prezintă comportament complex, dependent de stare (reacționează diferit la același stimул, funcție de starea în care se află)
  - Şablonul de proiectare *State*

# Diagrame de activități

- O *diagramă de activități* (eng. *activity diagram*) descrie modul de realizare a unui anumit comportament în termenii uneia sau a mai multor secvențe de activități și a fluxurilor de obiecte necesare pentru coordonarea acestor activități
  - Diagramele de activități sunt ierarhice: o activitate reprezintă fie o *acțiune*, fie *un graf de subactivități* cu fluxurile de obiecte aferente
- Ex.: diagramă de activități aferentă gestionării unui *Eveniment* din sistemul SGA
  - Activitățile sunt reprezentate prin dreptunghiuri rotunjite
  - Săgețile dintre activități reprezintă fluxul de control
  - Execuția activităților este secvențială: o activitate se poate executa doar după terminarea activităților care o preced



- *SolutioneazaEveniment* - dispecerul primește rapoarte și alocă resurse
- *DocumenteazaEveniment* - ofițerii și dispecerii implicați documentează evenimentul

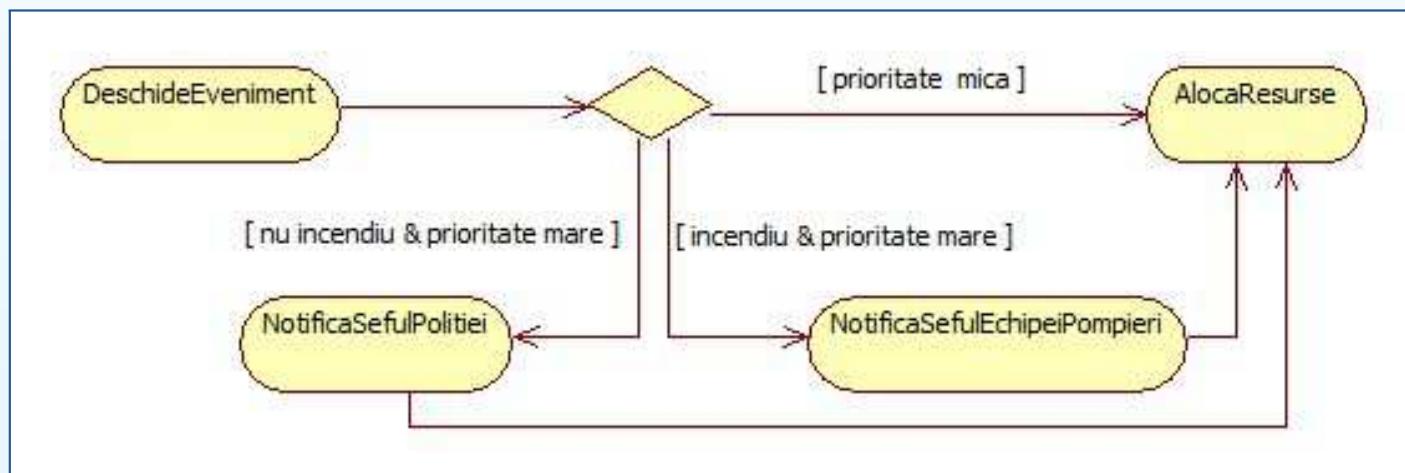
## Elemente de control

---

- *Elementele de control* (eng. *control nodes*) permit coordonarea fluxului de control dintr-o diagramă de activități, oferind mecanisme de reprezentare a deciziilor, a concurenței și sincronizării
- Tipuri principale de elemente de control
  - noduri decizionale
  - noduri fork
  - noduri join

## Noduri decizionale

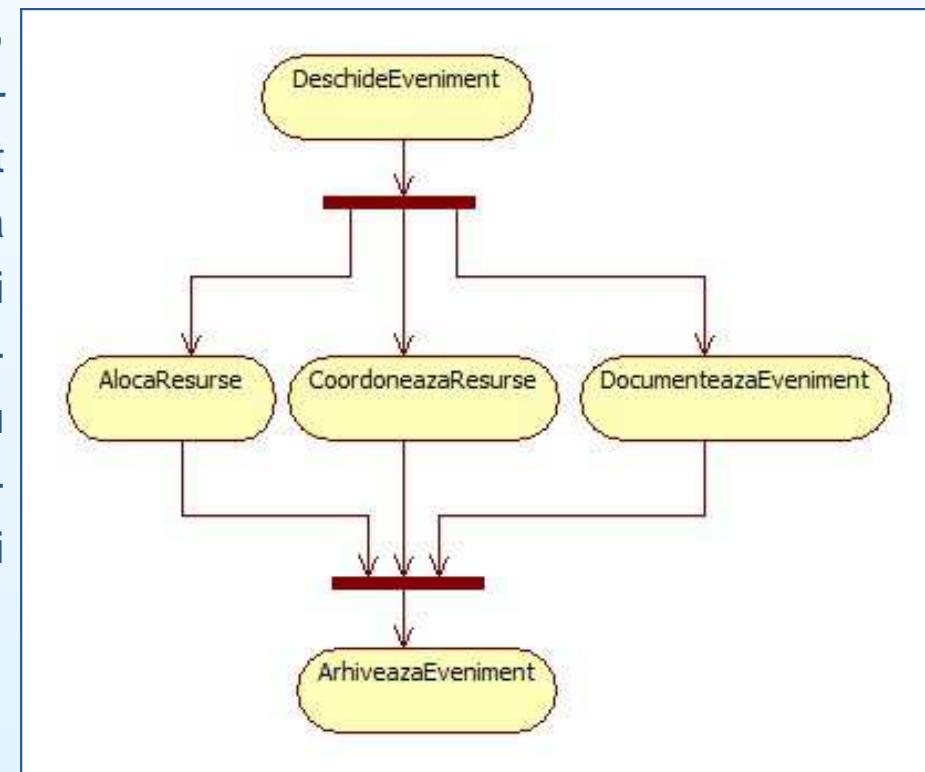
- *Nodul decizional* (eng. *decision node*) reprezintă o ramificare a fluxului de control, ce denotă alternative pe baza unei condiții relativ la starea unui obiect sau grup de obiecte
- Ex.: Nod decizional pentru gestionarea evenimentelor funcție de prioritate și tip



- Nodurile decizionale se reprezintă folosind un romb cu una sau mai multe săgeți de intrare și două sau mai multe săgeți de ieșire
- Ramurile de ieșire sunt etichetate cu condițiile aferente selectării lor în fluxul de control

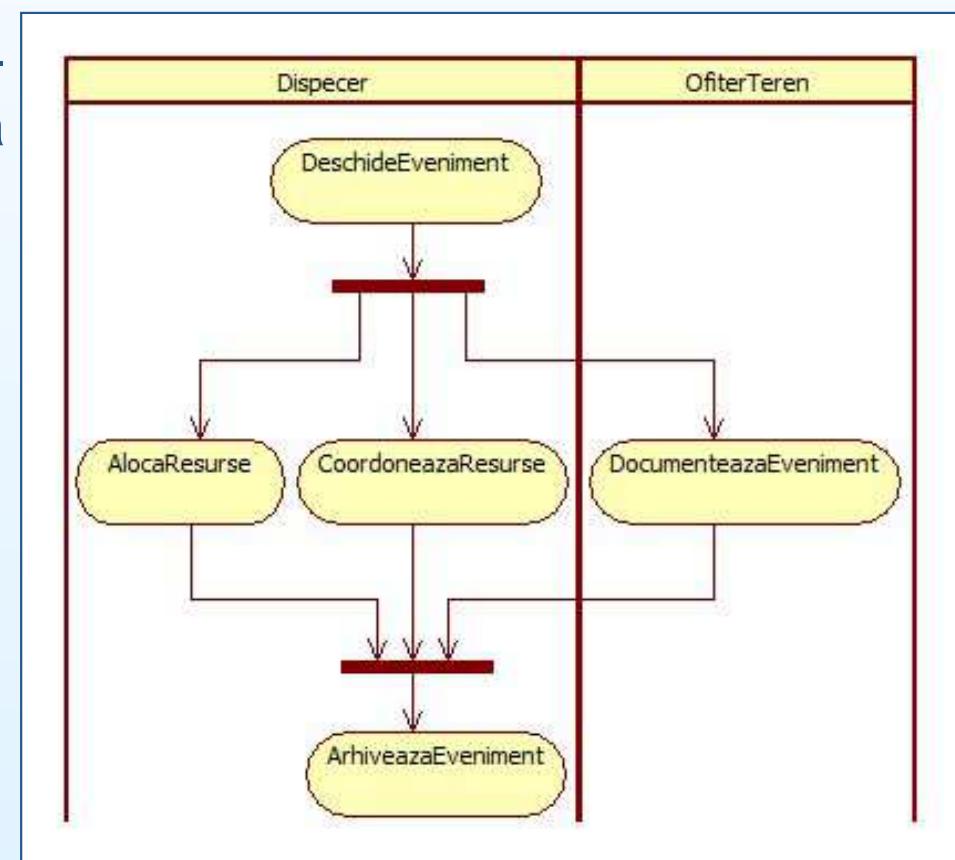
# Noduri fork și join

- Nodurile fork și join permit reprezentarea concurenței și a sincronizării
  - Nodurile fork indică divizarea fluxului de control în thread-uri
  - Nodurile join indică sincronizarea thread-urilor și combinarea fluxurilor de control într-un singur thread
- Ex.: Activitățile *AlocaResurse*, *CoordoneazaResurse*, *DocumenteazaEveniment* pot fi efectuate în paralel, însă doar după terminarea activității *DeschideEveniment*. Activitatea *ArhiveazaEveniment* nu poate fi inițiată decât după terminarea tuturor celor trei activități concurente.



# Partiționarea activităților

- Activitățile pot fi grupate pe *partiții* (eng. *swimlanes*), pentru a indica obiectul sau subsistemul care le va implementa
  - Partițiiile sunt reprezentate ca și dreptunghiuri ce conțin grupuri de activități
  - Tranzițiile pot intersecta partițiiile
- Ex.: Partiționarea activităților legate de gestiunea evenimentelor



## Utilizarea diagramelor de activități

---

- Diagramele de activități oferă o vedere centrată pe sarcini a comportamentului unei mulțimi de obiecte
- Pot fi utilizate pentru
  - Descrierea constrângerilor privind secvențierea cazurilor de utilizare
  - Descrierea activităților secvențiale în cadrul unui grup de obiecte
  - Descrierea sarcinilor unui proiect

## Referinte

---

- [Rumbaugh et al., 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [Booch, 1994] G. Booch, *Object-Oriented Analysis and Design with Applications*, 2nd ed., Benjamin/Cummings, Redwood City, CA, 1994 .
- [Jacobson et al., 1992] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley, Reading, MA, 1992.
- [Harel, 1987] D. Harel, *Statecharts: A visual formalism for complex systems*, Science of Computer Programming, pp. 231-274, 1987.

*Curs 3*  
*Colectarea cerințelor*

*Suport de curs bazat pe B. Bruegge and A.H. Dutoit  
"Object-Oriented Software Engineering using UML, Patterns, and Java"*

## Sumar Curs 3

---

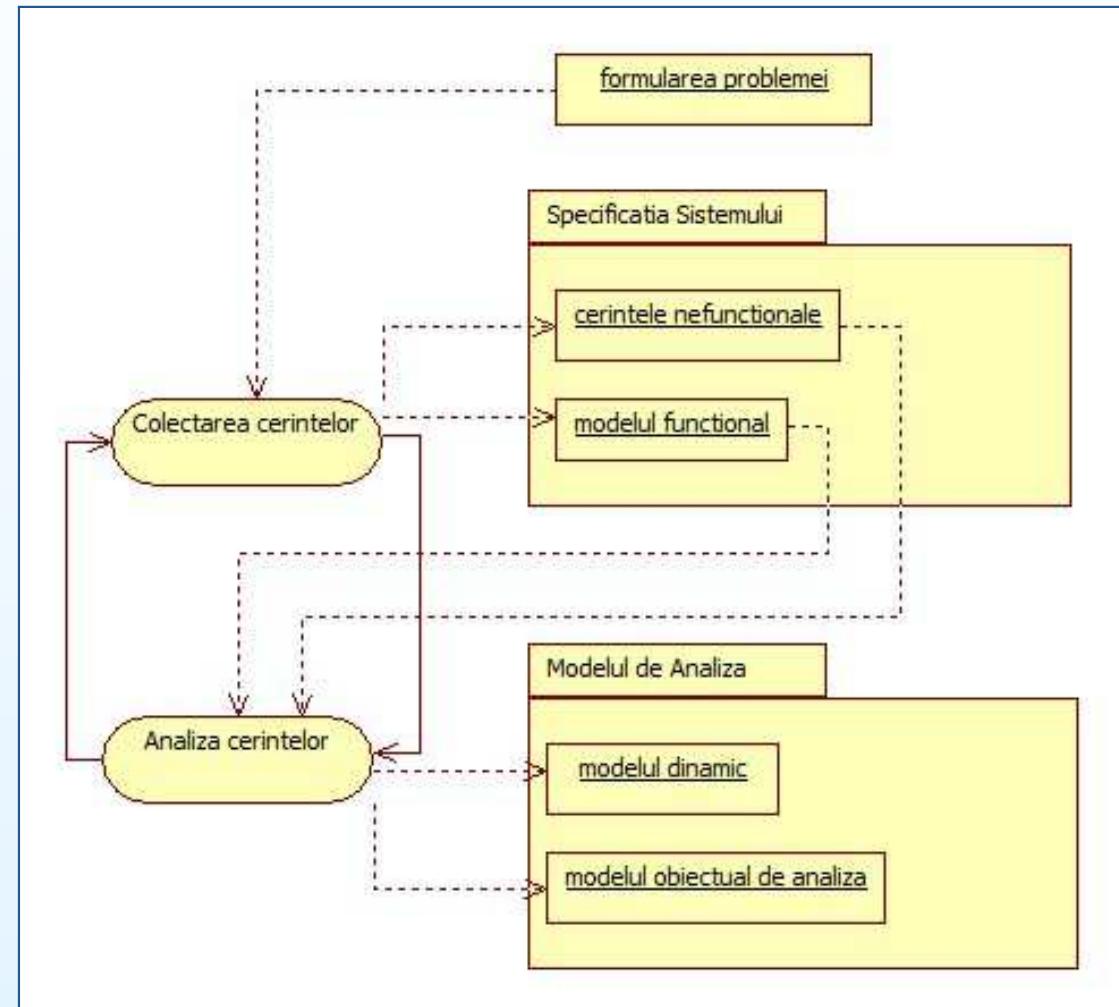
- Cerințe. Ingineria cerințelor
- Colectarea cerințelor - concepte
- Colectarea cerințelor - activități tehnice
- Colectarea cerințelor - management

# Cerințe. Ingineria cerințelor

- O cerință (eng. *requirement*) reprezintă un element de funcționalitate pe care sistemul trebuie să îl ofere sau o constrângere pe care trebuie să o îndeplinească
- *Ingineria cerințelor* (eng. *requirements engineering*) este un subdomeniu al ingineriei softului, având drept scop definirea cerințelor sistemelor soft ce urmează a fi construite
- Activități ale ingineriei cerințelor
  - Colectarea cerințelor (eng. *requirements elicitation*)  $\implies$  *specificația sistemului* (contract între client și dezvoltatori)
  - Analiza cerințelor (eng. *analysis*)  $\implies$  *modelul de analiză*
- Atât specificația sistemului, cât și modelul de analiză reflectă aceeași informație, ele diferă însă prin rolul lor și notația utilizată
  - Specificația sistemului este exprimată în limbaj natural și servește ca și instrument de comunicare cu clienții/utilizatorii
  - Modelul de analiză este exprimat într-o notație semiformală sau formală și servește ca și instrument de comunicare între dezvoltatori

## Cerințe. Ingineria cerințelor (cont.)

- Activități și produse ale ingineriei cerințelor (diagramă UML de activități)
  - Activitățile ingineriei cerințelor se focusează exclusiv pe aspectele externe ale sistemului (perspectiva utilizatorilor externi asupra sistemului)
  - Modelele lor reprezentând aceleași aspecte, cele două activități se desfășoară concurrent și iterativ



# Colectarea cerințelor

- Colectarea cerințelor necesită colaborarea între grupuri de participanți cu background diferit (cliienți/utilizatori - experți în domeniul problemei vs. dezvoltatori - experți în dezvoltare de soft)
  - Corectarea erorilor de comunicare introduse acum (rezultând în specificarea greșită a unor funcționalități, funcționalități lipsă, probleme la nivelul interfețelor utilizator) în etapele ulterioare ale dezvoltării este deosebit de costisitoare în termeni de timp/buget
  - Ca urmare, metodele de colectare a cerințelor au drept obiectiv îmbunătățirea comunicării între aceste grupuri
    - Ex.: *Colectarea cerințelor bazată pe scenarii* (eng. *scenario-based requirements elicitation*)
      - Scenariu = descriere a unui exemplu concret de utilizare a sistemului, în termenii unei secvențe de interacțiuni între utilizator și sistem
      - Caz de utilizare = abstractizare care descrie elementele comune ale unei clase de scenarii

## Colectarea cerințelor bazată pe scenarii

- Dezvoltatorii colectează cerințele prin observarea și chestionarea utilizatorilor în mediul lor
- Se realizează două tipuri de scenarii
  - Inițial - scenarii descriind modul curent de desfășurare a proceselor de lucru în mediul utilizatorilor
  - Ulterior - scenarii prescriptive, ilustrând funcționalitățile care urmează a fi oferite de către noul sistem
- Scenariile realizate sunt validate de către clienți și utilizatori prin inspectare și prin manipularea unor prototipuri ale interfeței grafice cu utilizatorul, oferite de către dezvoltatori
- Pe măsură ce definiția sistemului se stabilizează, dezvoltatorii și clientul convin asupra unei specificații a sistemului constând în descrierea cerințelor funcționale, a celor nefuncționale, a cazurilor de utilizare și a scenariilor aferente

# Activități ale colectării cerințelor bazate pe scenarii

---

- *Identificarea actorilor*
  - Dezvoltatorii identifică diferențele tipuri de utilizatori a căror activitate va fi susținută de către viitorul sistem
- *Identificarea scenariilor*
  - Dezvoltatorii observă utilizatorii în mediul lor și descriu un set de scenarii detaliate aferente funcționalităților tipice oferite de către noul sistem
  - Scenariile sunt folosite pentru comunicarea cu utilizatorii și aprofundarea înțelegерii domeniului problemei de către dezvoltatori
- *Identificarea cazurilor de utilizare*
  - Ulterior stabilizării scenariilor, dezvoltatorii derivă din acestea cazurile de utilizare ce definesc viitorul sistem
- *Rafinarea cazurilor de utilizare*
  - Dezvoltatorii verifică completitudinea specificării cerințelor prin detalierea fiecărui caz de utilizare și descrierea comportamentului sistemului în situații de excepție

## Activități ale colectării bazate pe scenarii (cont.)

---

- *Identificarea relațiilor dintre cazurile de utilizare*
  - Dezvoltatorii factorizează comportamentele comune mai multor cazuri de utilizare și identifică dependențele dintre acestea
  - Această activitate permite verificarea consistenței specificației sistemului
- *Identificarea cerințelor nefuncționale*
  - Dezvoltatorii, clienții și utilizatorii convin asupra constrângerilor legate de performanțele sistemului, resursele utilizate, securitatea și calitatea sa, modalitatea de documentare, etc.

## Colectarea cerințelor - concepte

---

- Cerințe funcționale
- Cerințe nefuncționale
- Completitudine
- Consistență
- Claritate
- Corectitudine
- Realism
- Verificabilitate
- Trasabilitate
- Inginerie Greenfield
- Re-inginerie
- Ingineria interfețelor

## Cerințe funcționale

- *Cerințele funcționale* (eng. *functional requirements*) descriu interacțiunile dintre sistem și mediul acestuia, independent de implementare
  - Mediul = utilizatorii + alte sisteme cu care sistemul în cauză interacționează
- Ex.: Specificarea cerințelor funcționale ale sistemului *SatWatch* - ceas care se actualizează automat (fără intervenții externe)
  - *SatWatch* este un ceas de mână care afișează timpul pe baza locației curente. *SatWatch* utilizează sateliți GPS pentru a determina locația curentă și structuri interne pentru a asocia acelei locații un fus orar.  
*SatWatch* actualizează ora și data la schimbarea fusului orar sau a granițelor politice. Proprietarul ceasului nu trebuie să îl reseteze niciodată, ca urmare ceasul nu are butoane de control.  
*SatWatch* utilizează sateliți GPS pentru determinarea locației curente, ca urmare are aceleași limitări ca și restul dispozitivelor GPS (ex. incapacitatea de a determina locația în anumite contexte, ex. tuneluri/pasaje subterane). În timpul perioadelor de black-out, *SatWatch* consideră că nu se schimbă fusurile orare sau regiunile politice. Imediat după ieșirea din intervalul de black-out, se realizează actualizările aferente.

## Cerințe funcționale (cont.)

*SatWatch* are un afisaj pe două linii, cea de sus indicând timpul (ora, minute, secunde, fus orar), iar cea de jos data (zi din săptămână, număr zi din lună, luna, an).

În cazul modificării granițelor politice, proprietarul poate actualiza softul ceasului prin utilizarea dispozitivului *WebifyWatch* (oferit împreună cu ceasul) și a unui calculator conectat la Internet.

- Cerințele funcționale anterioare surprind doar interacțiunile dintre sistemul *SatWatch* și mediul extern (proprietarul, sateliții GPS, dispozitivul *WebifyWatch*), fără a referi detalii de implementare (procesor, limbaj, tehnologie, etc.)

# Cerințe nefuncționale

- *Cerințele nefuncționale* (eng. *nonfunctional requirements*) descriu diverse tipuri de constrângeri impuse asupra sistemului, ortogonale pe aspectele legate de funcționalitate
- Categorii de cerințe nefuncționale (*de calitate*), conform modelului FURPS+ ([Grady 1992], [IEEE Std. 610.12-1990])
  - *Utilizabilitate* (eng. *usability*)
    - Denotă ușurința cu care un utilizator poate învăța să opereze, să pregătească intrări sau să interpreteze ieșiri ale unui sistem sau ale unei componente
    - Ex. de cerințe privind utilizabilitatea: diferite conveții adoptate la nivelul interfețelor grafice (șabloane de structurare, scheme de culori, logo-uri, fonturi), help online, ghid de utilizare detaliat
  - *Performanță* (eng. *performance*) - referă atribute cuantificabile, precum:
    - *timpul de răspuns* (rapiditatea reacției sistemului la inputul utilizatorului)
    - *puterea de calcul* (volumul de calcule efectuate într-un interval de timp)
    - *acuratețea rezultatelor*
    - *disponibilitatea* (măsura în care un sistem este operațional și accesibil atunci când se dorește utilizarea lui)

## Cerințe nefuncționale (cont.)

- *Fiabilitate* (eng. *reliability*)
  - Reprezintă abilitatea sistemului sau componentei de a îndeplini funcțiile cerute în condițiile stabilită, pentru o anumită perioadă de timp
  - Ex. de cerințe privind fiabilitatea: un timp mediu precizat între eșecuri, abilitatea de a face față unor atacuri de securitate, etc.
  - Referintă și cu termenul de *dependabilitate* și acoperind, pe lângă *corectitudine* (eng. *correctness* - conformanța la specificații) și *robustetea* (eng. *robustness*) - gradul în care un sistem sau o componentă poate funcționa corect în prezența unor intrări invalide sau a unor condiții excepționale și *siguranța* (eng. *safety*) - o măsură a absenței unor consecințe catastrofale în mediu
- *Suportabilitate* (eng. *supportability*)
  - Denotă ușurința modificării sistemului după instalare, incluzând *adaptabilitatea* (eng. *adaptability*) - abilitatea de a modifica sistemul pentru a opera cu noi concepte din domeniul problemei, *mentenabilitatea* (eng. *maintainability*) - abilitatea de a schimba sistemul pentru a opera cu noi tehnologii sau a repara defecte și *internationalizarea* (eng. *internationalization*) - abilitatea de a schimba sistemul pentru a opera cu limbi/unități de măsură/formate numerice străine

## Cerințe nefuncționale (cont.)

---

- Categorii adiționale de cerințe calificate drept nefuncționale în modelul FURPS+ (*pseudo-cerințe sau constrângeri*)
  - *Cerințe privind implementarea* (eng. *implementation requirements*)
    - Sunt constrângeri care vizează utilizarea unei anumite platforme hardware, a unui anumit limbaj de programare sau a anumitor instrumente
  - *Cerințe privind interfața* (eng. *interface requirements*)
    - Sunt constrângeri impuse de către alte sisteme cu care sistemul în cauză interfețează
  - *Cerințe privind modul de operare* (eng. *operation requirements*)
    - Sunt constrângeri privind administrarea și gestiunea sistemului în mediul său de operare
  - *Cerințe privind instalarea* (eng. *packaging requirements*)
    - Sunt constrângri legate de livrarea sistemului, cum ar fi suportul folosit pentru instalare
  - *Cerințe legale* (eng. *legal requirements*)
    - Sunt constrângeri legate de licențe, legi, certificări
    - Ex.: cerințe privind obligativitatea oferirii accesului la un anumit soft persoanelor cu dizabilități

## Cerințe nefuncționale (cont.)

- Ex.: cerințe de calitate pentru *SatWatch*
  - Orice utilizator care știe să citească un ceas digital și cunoaște abrevierile legate de fusurile orare trebuie să poată folosi sistemul fără manual de utilizare (utilizabilitate)
  - *SatWatch* trebuie să afișeze fusul orar corect în maxim 5 minute de la ieșirea dintr-o perioadă de black-out (performanță)
  - Deoarece *SatWatch* nu dispune de butoane, nu trebuie să apară nici o eroare care să necesite resetarea acestuia (fiabilitate)
  - *SatWatch* trebuie să admită actualizări prin intermediul interfeței seriale *WebifyWatch* (suportabilitate)
- Ex.: constrângeri pentru *SatWatch*
  - Softul *SatWatch* va fi scris integral în Java, pentru conformanță cu standardele companiei (implementare)
  - *SatWatch* se conformează interfețelor fizice, electrice și soft definite de *WebifyWatch 2.0* (interfață)

# Completitudine, consistență, claritate și corectitudine

---

- Validarea continuă a cerințelor reprezintă un imperativ în procesul de dezvoltare
- Validarea cerințelor presupune verificarea completitudinii, consistenței, clarității și corectitudinii acestora
  - *Completitudine*
    - Specificarea cerințelor se consideră a fi completă dacă surprinde toate aspectele de interes pentru clienți/utilizatori (au fost descrise toate scenariile posibile, inclusiv cele de excepție)
  - *Consistență*
    - Specificarea cerințelor se consideră a fi consistentă dacă oricare două cerințe sunt non-contradictorii
  - *Claritate/non-ambiguitate*
    - Specificarea cerințelor se consideră a fi neambiguă dacă o aceeași cerință nu admite interpretări distincte
  - *Corectitudine*
    - Specificarea cerințelor se consideră a fi corectă dacă reprezintă fidel interesele clientului și ale dezvoltatorilor, fără a include elemente nedorite
- Corectitudinea și completitudinea sunt greu de apreciat/certificat, mai ales anterior existenței sistemului

# Realism, verificabilitate și trasabilitate

- Specificarea cerințelor se consideră a fi *realistă* dacă sistemul poate fi implementat cu constrângerile impuse
- Specificarea cerințelor se consideră a fi *verificabilă* dacă ulterior dezvoltării sistemului pot fi proiectate teste care să dovedească conformanța acestuia cu specificația
  - Ex.: cerințe greu/imposibil de verificat
    - Sistemul *SatWatch* trebuie să aibă un interval mediu între eșecuri de 100 de ani
    - Produsul trebuie să aibă o interfață utilizator bună
    - Produsul trebuie să fie fără erori
- O specificare a cerințelor are proprietatea de *trasabilitate* dacă fiecare cerință poate fi urmărită, de-a lungul procesului de dezvoltare, până la funcțiile sistem care o implementează și reciproc
  - Trasabilitatea cerințelor e o constrângere critică pentru dezvoltarea testelor și analiza impactului schimbărilor asupra sistemului

# Trasabilitatea cerințelor

- *Trasabilitatea* (eng. *traceability*) reprezintă abilitatea de a urmări evoluția unei cerințe
  - Presupune urmărirea cerinței de la origine (cine a introdus-o? cărei nevoi client corespunde?) până la nivelul acelor componente ale sistemului pe care le afectează (ce componente implementează cerința? ce cazuri de test verifică îndeplinirea sa?)
  - Trasabilitatea permite dezvoltatorilor să motiveze completitudinea sistemului, testerilor să justifice conformanța acestuia cu cerințele, proiectanților să înregistreze argumentele din spatele deciziilor luate și echipelor de întreținere să evaluateze efectul schimbărilor
- Trasabilitatea poate fi urmărită prin întreținerea de referințe între documente, modele și cod
  - Fiecare element individual (cerință, componentă, clasă, operație, caz de test) primește un identificator unic
  - O dependență este apoi documentată prin intermediul unei referințe conținând identificatorii componentelor sursă și destinație

# Inginerie Greenfield, re-inginerie, ingineria interfețelor

---

- Caracteristicile activității de colectare a cerințelor depind de sursa acestora
  - *Ingineria Greenfield*
    - Procesul de dezvoltare începe de la 0, nu există un sistem anterior
    - Cerințele sunt furnizate doar de client și utilizatori
  - *Re-inginerie*
    - Reproiectarea și reimplementarea unui sistem existent, ca urmare a unor modificări la nivelul proceselor de lucru sau a unor modificări tehnologice
    - Funcționalitatea sistemului poate fi eventual extinsă, însă scopul său rămâne același; majoritatea cerințelor sunt extrase din vechiul sistem
  - *Ingineria interfețelor*
    - Reproiectarea și reimplementarea doar a interfeței unui sistem existent
- În cazul ingineriei Greenfield și a reingineriei, dezvoltatorii trebuie să acumuleze cât mai multe cunoștințe relativ la domeniul problemei
  - Surse: descrieri ale proceselor de lucru, documentație distribuită noilor angajați, manuale ale vechiului sistem, note ale utilizatorilor, interviuri cu clienții și utilizatorii

## Colectarea cerințelor - activități tehnice

---

- Identificarea actorilor
- Identificarea scenariilor
- Identificarea cazurilor de utilizare
- Rafinarea cazurilor de utilizare
- Identificarea relațiilor între cazurile de utilizare
- Identificarea cerințelor nefuncționale

## Identificarea actorilor

- Permite identificarea frontierei sistemului și a perspectivelor din care acesta trebuie abordat
- Întrebări utile pentru identificarea actorilor
  - Care sunt grupurile de utilizatori a căror activitate este susținută de sistem?
  - Care sunt grupurile de utilizatori care execută funcțiile principale ale sistemului?
  - Care sunt grupurile de utilizatori care execută funcții secundare, precum întreținere sau administrare?
  - Care sunt echipamentele hardware și sistemele software externe cu care sistemul curent va interacționa?
- Întrebările anterioare conduc la identificarea unei liste de entități ce urmează a fi rafinată, în general, într-un număr mai mic de actori, diferenți din perspectiva utilizării sistemului
  - Se unifică entitățile care utilizează aceleași interfețe
  - Se elimină entitățile care, cel mai probabil, nu vor interacționa în mod direct cu sistemul

## Identificarea scenariilor

- În colectarea cerințelor, dezvoltatorii și utilizatorii concep și rafinează o serie de scenarii, pentru a dobândi un nivel comun de înțelegere relativ la funcționalitatea sistemului
- Întrebări utile pentru identificarea scenariilor
  - Care sunt sarcinile pe care utilizatorul dorește să le execute sistemul?
  - Care sunt informațiile pe care le poate accesa actorul? Cine creează aceste date? Pot fi ele modificate sau șterse? De către cine?
  - Care sunt modificările externe despre care actorul trebuie să informeze sistemul? Cât de des? Când?
  - Care sunt evenimentele despre care sistemul trebuie să informeze actorul? Cu ce periodicitate?
- Pentru a răspunde întrebărilor anterioare, dezvoltatorii consultă diverse documente cu informații privind domeniul problemei
  - manuale ale unor sisteme anterioare, manuale procedurale, standarde ale companiei, note și documente elaborate de utilizatori, interviuri cu clienții și utilizatorii

## Identificarea scenariilor (cont.)

- Ex.: Scenariul *depozitIncendiat* al cazului de utilizare *RaporteazăUrgență* al SGA

<b>Nume</b>	<u>depozitIncendiat</u>
<b>Instanțe</b>	<u>bob, alice : OfițerTeren</u>
<b>actori</b>	<u>john : Dispecer</u>
<b>Flux de evenimente</b>	<ol style="list-style-type: none"><li>1. Trecând prin dreptul unui depozit, Bob simte miros de fum. Partenera sa, Alice, activează funcția <i>Raportează urgență</i> pe terminalul SGA.</li><li>2. Alice introduce adresa clădirii, o scurtă descriere a locației și un nivel de alertă. Zona fiind aglomerată, solicită o echipă de pompieri și mai multe de medici. Confirmă datele și așteaptă confirmarea dispecerului.</li><li>3. John, dispecerul, este alertat de un semnal sonor al stației sale de lucru. Verifică informațiile trimise de Alice și confirmă primirea lor. Alocă o echipă de pompieri și două de medici și ii trimitе lui Alice ora estimată a sosirii acestora.</li><li>5. Alice primește confirmarea și estimarea.</li></ol>

# Identificarea cazurilor de utilizare

- Ex.: Cazul de utilizare *RaporteazăUrgență* al SGA

<b>Nume</b>	<i>RaporteazăUrgență</i>
<b>Participanți</b>	Inițiat de <i>OfițerTeren</i> Comunică cu <i>Dispecerul</i>
<b>Flux de evenimente (scenariu normal)</b>	<ol style="list-style-type: none"><li>1. <i>Ofițerul</i> activează funcția <i>Raportează urgență</i> a terminalului.</li><li>2. Sistemul SGA afișează un formular <i>Ofițerului</i>.</li><li>3. <i>Ofițerul</i> completează formularul, inserând nivelul de alertă, tipul, locația și o scurtă descriere a situației. Poate propune și eventuale soluții la situația de urgență. După completare, <i>Ofițerul</i> trimitе formularul.</li><li>4. Sistemul primește formularul și notifică <i>Dispecerul</i>.</li><li>5. <i>Dispecerul</i> consultă informația primită și apelează cazul de utilizare <i>DeschideCazNou</i>. <i>Dispecerul</i> optează pentru una dintre soluțiile propuse și confirmă primirea formularului.</li><li>6. Sistemul afișează confirmarea și soluția aleasă <i>Ofițerului</i>.</li></ol>
<b>Condiții de intrare</b>	<i>Ofițerul</i> este logat în sistem.
<b>Condiții de ieșire</b>	<i>Ofițerul</i> a primit confirmarea de la <i>Dispecer</i> SAU o explicație privind motivul eșecului tranzacției.
<b>Cerințe de calitate</b>	Confirmarea <i>Dispecerului</i> ajunge în maxim 30 de sec. după trimitere.

## Identificarea cazurilor de utilizare (cont.)

---

- Ghid de descriere a cazurilor de utilizare
  - Numele cazurilor de utilizare trebuie să fie construcții verbale, care să indice scopul utilizatorului (ex. *RaporteazăUrgență*, *AlocăResurse*)
  - Numele actorilor trebuie să fie substantive (ex. *OfițerTeren*, *Dispecer*)
  - Frontiera sistemului trebuie să fie clară, distingându-se între acțiunile actorilor și cele ale sistemului
  - Evenimentele care definesc cazul de utilizare trebuie formulate la modul activ
  - Relația de cauzalitate între două evenimente consecutive trebuie să fie clară
  - Fluxul normal al unui caz de utilizare trebuie să descrie o tranzacție completă (ex.: fluxul normal al cazului *RaporteazăUrgență* descrie toți pașii, de la inițierea raportului și până la primirea confirmării finale)
  - Excepțiile/alternativele se descriu separat
  - Descrierea unui caz de utilizare trebuie să evite referirea directă a componentelor interfeței grafice cu utilizatorul
  - Descrierea unui caz de utilizare nu trebuie să depășească două-trei pagini. În caz contrar, cazul se descompune, folosind relațiile între cazuri de utilizare

# Rafinarea cazurilor de utilizare

---

- Focusul acestei activități îl constituie completitudinea și corectitudinea specificării cerințelor
  - Dezvoltatorii identifică funcționalități neacoperite de scenariile descrise (cazuri rare sau excepții) și le documentează prin rafinarea cazurilor de utilizare existente sau introducerea unor noi cazuri
  - Spre deosebire de identificarea inițială a actorilor și cazurilor de utilizare, focusată pe definirea frontierei sistemului, etapa de rafinare introduce detalii suplimentare privind funcționalitățile oferite de sistem și constrângerile asociate
- Sunt detaliate următoarele aspecte
  - Informația manipulată de sistem
  - Interacțiunile dintre actori și sistem
  - Drepturile de acces (ce cazuri de utilizare poate invoca fiecare actor)
  - Cazurile de excepție (sunt identificate și specificate)
  - Funcționalitatea comună (este factorizată)

## Rafinarea cazurilor de utilizare (cont.)

- Ex.: Rafinarea cazului de utilizare *RaporteazăUrgență*

<b>Nume</b>	<i>RaporteazăUrgență</i>
<b>Participanți</b>	<i>Inițiat de OfițerTeren Comunică cu Dispecerul</i>
<b>Flux de evenimente</b>	<ol style="list-style-type: none"><li>1. Ofițerul activează funcția <i>Raportează urgență</i> a terminalului.</li><li>2. Sistemul SGA afișează un formular <i>Ofițerului</i>. <i>Formularul include componente privind nivelul de alertă, tipul urgenței (general, incendiu, accident auto), locația, descrierea și resursele solicitate.</i></li><li>3. Ofițerul completează formularul, <i>inserând cel puțin nivelul de alertă și descrierea situației</i>. El poate propune și eventuale soluții la situația de urgență și poate să solicite anumite resurse. După completare, <i>Ofițerul</i> trimite formularul.</li><li>4. Sistemul primește formularul și notifică <i>Dispecerul</i>.</li><li>5. <i>Dispecerul</i> verifică informația primită și creează un nou <i>Eveniment</i> în baza de date prin invocarea cazului de utilizare <i>DeschideCazNou</i>. <i>Toate informațiile din formularul primit sunt asociate automat evenimentului creat. Dispecerul alege un răspuns prin alocarea de resurse la eveniment (prin cazul de utilizare AlocăResurse) și confirmă primirea formularului printr-un scurt mesaj către ofițer.</i></li><li>6. Sistemul afișează confirmarea și răspunsul ales <i>Ofițerului</i>.</li></ol>

...

## Identificarea relațiilor

---

- Identificarea relațiilor dintre actori și cazurile de utilizare (comunicare, incluziune, extindere, generalizare) permite reducerea complexității modelului, eliminarea redundanțelor și a potențialelor inconsistențe
- Euristică privind folosirea relațiilor de incluziune și extindere
  - Utilizarea relației de incluziune pentru factorizarea comportamentului comun mai multor cazuri de utilizare
  - Utilizarea relației de extindere pentru modelarea comportamentului opțional, excepțional sau cu frecvență redusă de apariție
  - Utilizarea judicioasă a relațiilor, pentru a evita suprastructurarea modelului funcțional. Un singur caz de utilizare mai complex se poate dovedi a fi mai inteligibil decât un număr prea mare de cazuri mai simple

# Identificarea cerințelor nefuncționale

- Întrebări ce permit identificarea cerințelor nefuncționale aferente modelului FURPS+
  - Utilizabilitate
    - Care este nivelul de expertiză al utilizatorilor?
    - Ce standarde de interfețe sunt familiare utilizatorilor?
    - Ce documentație trebuie pusă la dispoziția utilizatorilor?
  - Fiabilitate (incluzând robustețe, siguranță și securitate)
    - Cât de fiabil/robust trebuie să fie sistemul?
    - Este repornirea sistemului o alternativă viabilă în cazul unui eșec?
    - Sunt admisibile pierderi de date? Ce volum?
    - Cum trebuie să gestioneze sistemul excepțiile?
    - Există cerințe privind siguranță?
    - Există cerințe privind securitatea?
  - Suportabilitate
    - Care sunt extinderile probabile ale sistemului?
    - Cine întreține sistemul?
    - Există probabilitatea de a porta sistemul pe alte platforme hardware/software?

## Identificarea cerințelor nefuncționale (cont.)

---

- Performanță
  - Există sarcini critice din punct de vedere al timpului?
  - Câți utilizatori concurenți trebuie să suporte sistemul?
  - Ce dimensiuni se estimează că va avea depozitul de date?
  - Care este cel mai slab timp de răspuns acceptat de utilizatori?
- Implementare
  - Există constrângeri privind platforma hardware?
  - Există constrângeri impuse echipei de testare?
  - Există constrângeri impuse echipei de întreținere?
- Interfață
  - Va interacționa sistemul cu alte sisteme?
  - Cum sunt exportate/importate datele la nivelul sistemului?
  - Există standarde utilizate de client care se aplică noului sistem?
- Instalare
  - Cine va instala sistemul?
  - Pe câte stații va fi sistemul instalat?
  - Există constrângeri de timp privind instalarea?

## Identificarea cerințelor nefuncționale (cont.)

---

- Operare
  - Cine gestionează sistemul după punerea în exploatare?
- Legal
  - Care este procedura de licențiere?
  - Există taxe rezultând din utilizarea anumitor componente/algoritmi?
  - Există penalizări în cazul căderii sistemului?

## **Colectarea cerințelor - management**

---

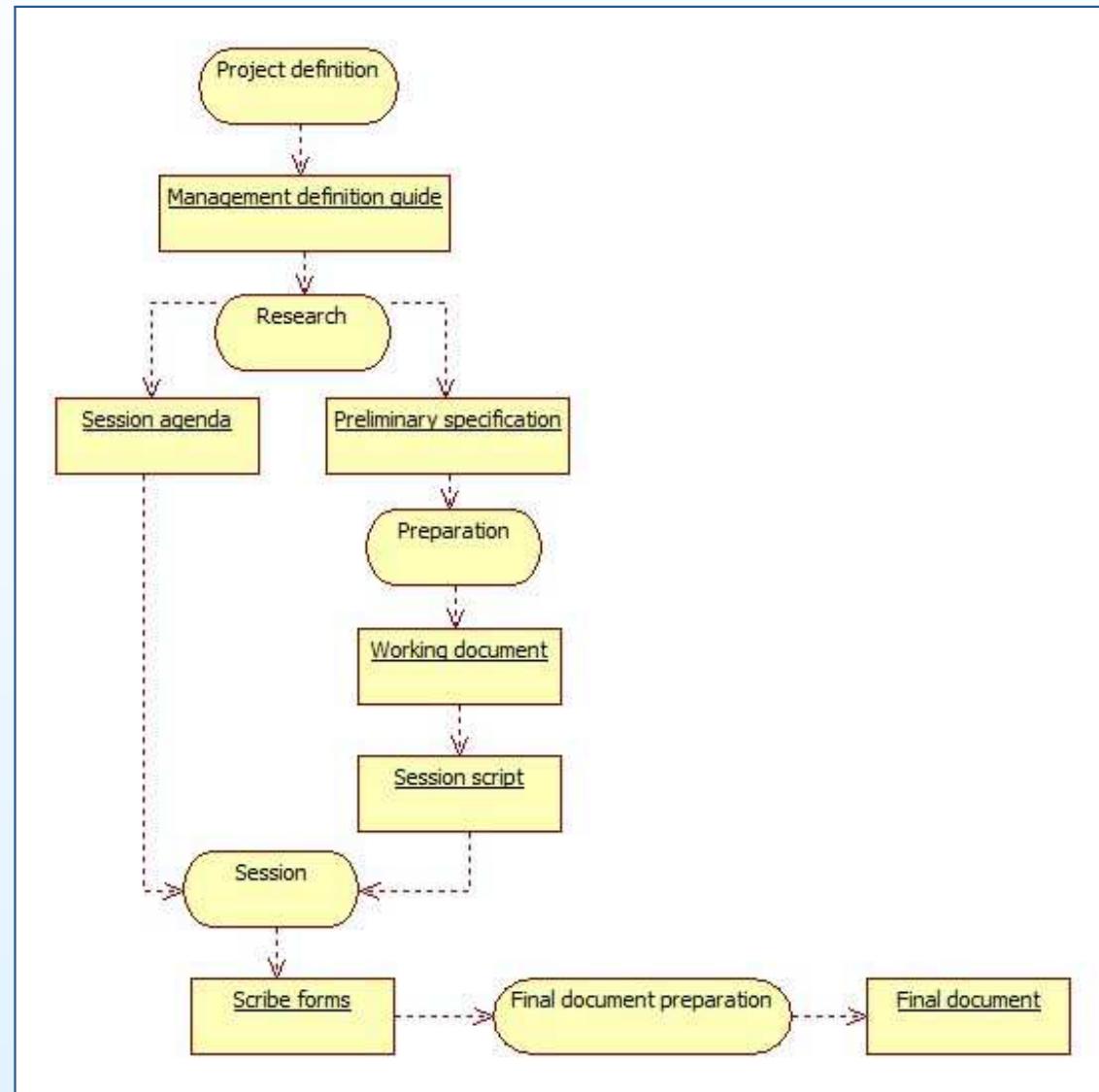
## Metoda JAD

---

- *JAD (Joint Application Design [WoodSilver 1989])* este o metodă de colectare a cerințelor dezvoltată la IBM la sfârșitul anilor '70
- Eficacitatea sa provine din faptul că activitatea de colectare a cerințelor se desfășoară într-o singură sesiune workshop, cu participarea tuturor celor interesați în dezvoltarea sistemului (clienti, utilizatori, dezvoltatori), precum și a unui moderator de sesiune specializat
- Produsul final al workshopului (documentul final JAD) reprezintă specificația completă a sistemului
- Fiind un document redactat de comun acord cu toți participanții, specificația JAD minimizează riscul unor modificări ulterioare ale cerințelor în procesul de dezvoltare
- Succesul unei sesiuni JAD depinde, de cele mai multe ori, de calificarea moderatorului

# Metoda JAD (cont.)

- Activități JAD



# Activități JAD

- *Project definition*
  - Moderatorul JAD interviewează project managerul și clientul, pentru a identifica scopul și obiectivele proiectului, informațiile fiind reținute în *Management Definition Guide*
- *Research*
  - Moderatorul JAD interviewează utilizatorii sistemului, colectează informații relativ la domeniul problemei și elaborează un prim set de cazuri de utilizare. De asemenea, deschide o listă de probleme ce se doresc a fi abordate în cadrul workshop-ului. Rezultatele acestei activități sunt reprezentate de *Session agenda* - agenda de lucru și *Preliminary specification* - specificația preliminară a sistemului
- *Preparation*
  - Moderatorul JAD pregătește sesiunea de lucru, creează o primă versiune a documentului final JAD - *Working document* și compune o echipă constând din project manager, client, utilizatori și dezvoltatori selectați.

# Activități JAD

- *Session*

- Moderatorul JAD coordonează echipa în elaborarea specificației sistemului. O sesiune JAD durează 3-5 zile. Se definesc scenariile, cazurile de utilizare și prototipurile interfeței cu utilizatorul. Toate deciziile sunt documentate (=> *Scribe forms*).

- *Final document preparation*

- Moderatorul pregătește documentul final - *Final document*, revizuind documentul de lucru pentru a include toate deciziile luate în sesiune. Documentul final reprezintă specificația completă sistemului, aprobată în cadrul sesiunii. Acesta este distribuit participanților pentru inspectare. Urmează o sesiune de 1-2 ore în care participanții dezbat modificările și finalizează documentul.

## Referinte

---

- [WoodSilver, 1989] J. Wood, D. Silver, *Joint Application Design*, Wiley, New York, 1989.
- [Grady 1992] R. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, Englewood Cliffs, NJ, 1992.
- [IEEE Std. 610.12-1990] Institute of Electrical and Electronics Engineers, *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, New York, NY, 1990.
- Use Case Analysis demos

Lesson1, <https://www.youtube.com/watch?v=KeMgPqLCkuo&t=19s>

Lesson2, <https://www.youtube.com/watch?v=8qyHeqInnFU>

Lesson3, <https://www.youtube.com/watch?v=iex3QbSo61c>

Lesson4, <https://www.youtube.com/watch?v=3OWNQddbrOs>

Lesson5, <https://www.youtube.com/watch?v=FrdCX9Gcc4g>

**Curs 4**  
*Analiza cerințelor*

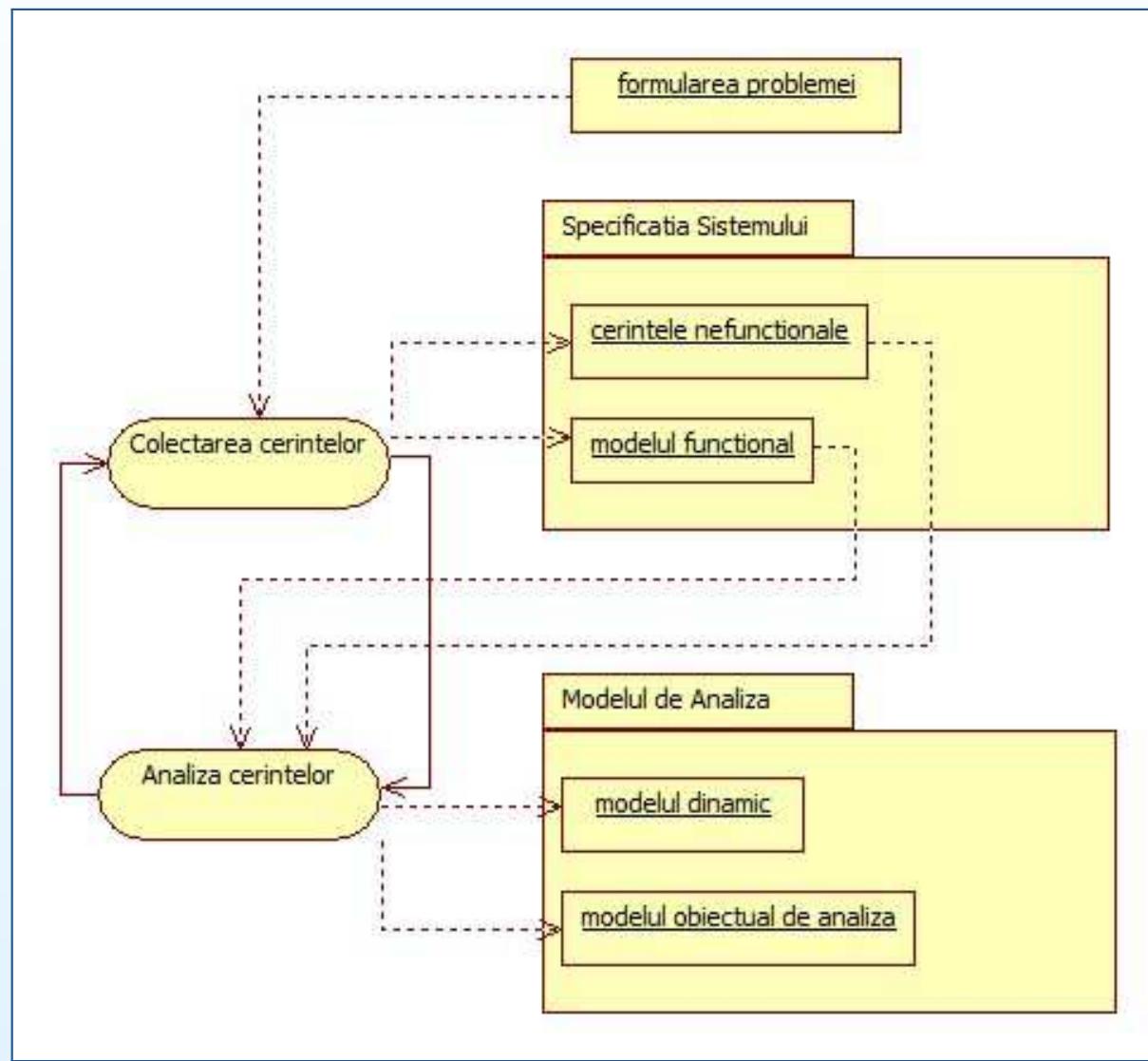
*Suport de curs bazat pe B. Bruegge and A.H. Dutoit  
"Object-Oriented Software Engineering using UML, Patterns, and Java"*

## Sumar Curs 4

---

- Analiza cerințelor - obiective
- Analiza cerințelor - concepte
- Analiza cerințelor - activități tehnice
- Documentul de analiză a cerințelor

# Ingineria cerințelor - activități și modele

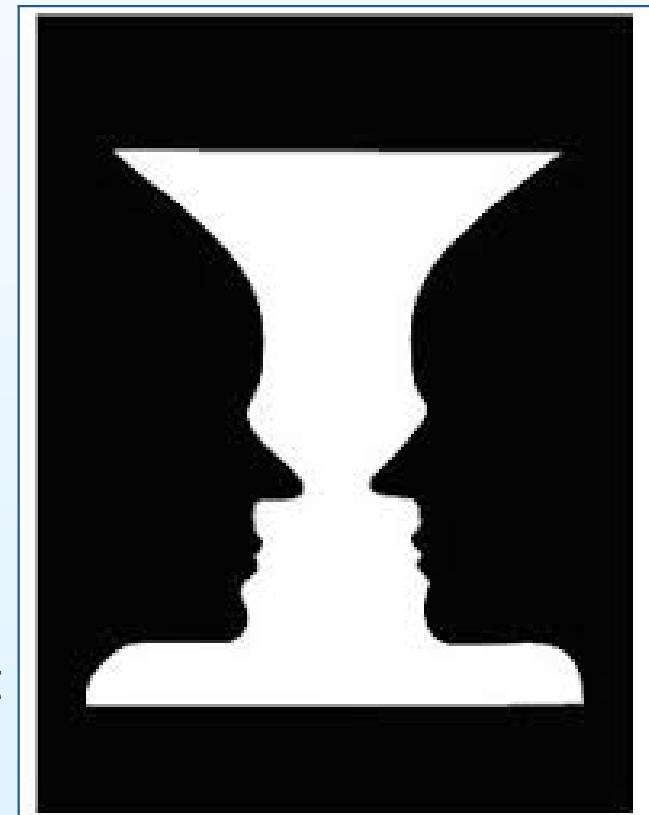


## Analiza cerințelor - obiective

- Scopul analizei cerințelor este realizarea unui model al sistemului (*model de analiză*, eng. *analysis model*) corect, complet, consistent și neambiguu
- Accentul este pus pe structurarea și formalizarea cerințelor colectate în etapa anterioară
- Formalizarea permite identificarea de ambiguități, inconsistențe și incompletitudini în descrierea cerințelor, soluționate prin discuții cu clientul/utilizatorii și rezultând în perfectarea specificației sistemului
- Colectarea și analiza cerințelor sunt activități concurente, care se desfășoară iterativ-incremental

### Imagini multistabile

- Ce reprezintă?
- Ambiguitate - dacă ar fi o specificare, ce model am construi?



## Analiza cerințelor - concepte

---

- *Modele obiectuale și modele dinamice*
- Clase *entity, boundary* și *control*
- *Generalizare / specializare*

# Modele obiectuale și modele dinamice

- *Modelul obiectual de analiză*
  - Surprinde concepțele manipulate de sistem, proprietățile și relațiile acestora
  - Se reprezintă cu ajutorul *diagramei de clase*
  - O clasă din modelul de analiză reprezintă o abstractizare pentru una sau mai multe clase din codul sursă (care, în general, vor conține și un număr mult mai mare de atribută și asocieri)
- *Modelul dinamic*
  - Surprinde comportamentul sistemului
  - Reprezentat cu ajutorul *diagramelor de secvență* și al *diagrameelor de tranziție a stărilor*
    - Diagramale de secvență surprind interacțiunile dintr-o mulțime de obiecte, în cadrul unui caz de utilizare
    - O diagramă de tranziție a stărilor surprinde comportamentul unui singur obiect (sau al unui grup de obiecte strâns cuplate)
  - În analiză, modelul dinamic permite identificarea responsabilităților entităților/ claselor existente, precum și identificarea de clase/ atribută/ asocieri noi
- **Modelele de analiză surprind doar concepte/ atribută/ relații/ comportamente percepute de utilizatori (*domeniul problemei*)**

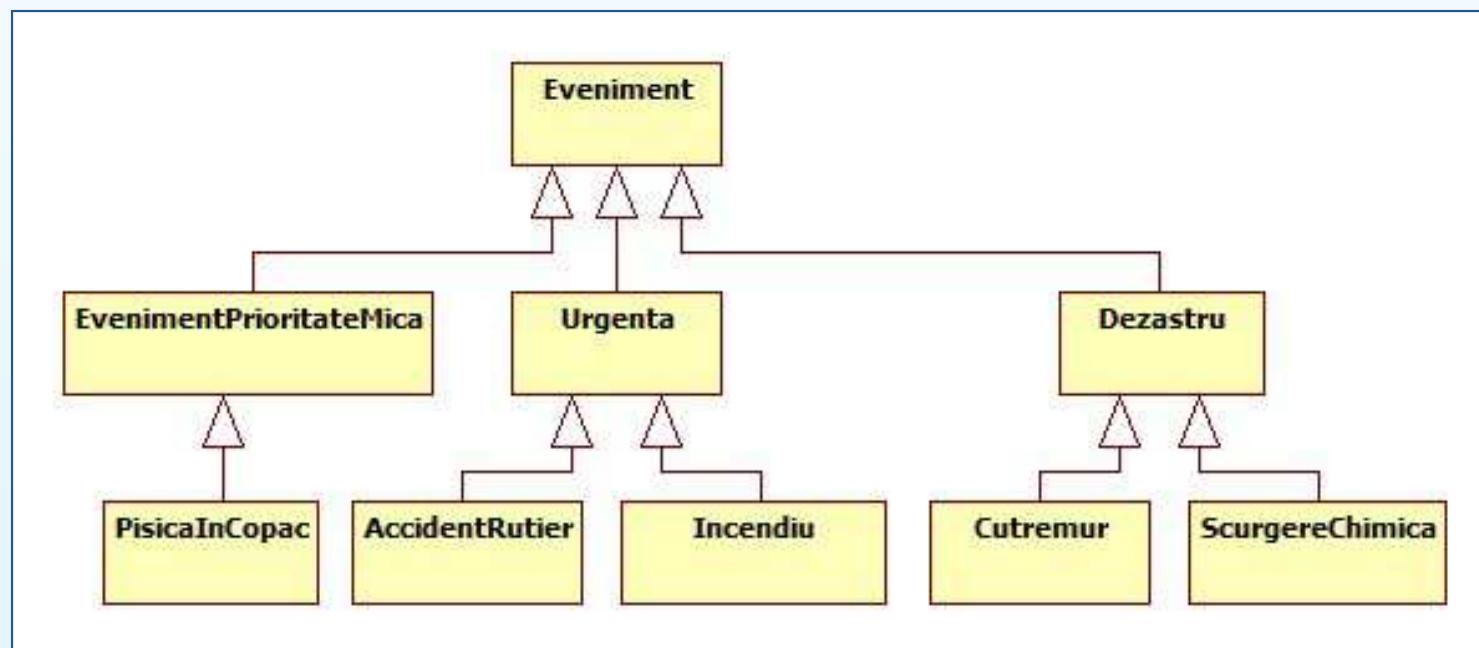
# Clase *entity*, *boundary* și *control*

- Modelul obiectual de analiză constă din clase *entity*, *boundary* și *control*
  - Clase *entity* - responsabile de informația persistentă din sistem
    - Ex.: clasa *RaportUrgență*, clasa *Eveniment*
  - Clase *boundary* - responsabile de interacțiunea actorilor cu sistemul
    - Ex.: clasa *ButonRaportareUrgență*, clasa *FormăRaportareUrgență*
  - Clase *control* - responsabile de realizarea cazurilor de utilizare
    - Ex.: clasa *Control/GestionareUrgență*
- Avantaje ale acestui tip de clasificare
  - Clase / obiecte mai mici, specializate
  - Modele mai ușor de modificat
    - Interfața (obiecte *boundary*) are o probabilitate mai mare de a fi modificată decât funcționalitatea de bază (obiecte *entity* și *control*)
- Notație: stereotipuri UML + convenții de denumire



# Generalizare / specializare

- Permit organizarea conceptelor în ierarhii
  - *Generalizare* = activitate de modelare având drept scop identificarea unor concepe abstracte, pornind de la unele de nivel jos
  - *Specializare* = activitate de modelare având drept scop identificarea unor concepe specializate, pornind de la unele de nivel înalt



## Analiza cerințelor - activități

- Identificarea claselor *entity*
- Identificarea claselor *boundary*
- Identificarea claselor *control*
- Maparea cazurilor de utilizare la obiecte cu diagrame de secvență
- Identificarea asocierilor
- Identificarea agregărilor
- Identificarea atributelor
- Modelarea comportamentului obiectelor cu diagrame de stări
- Identificarea ierarhiilor de clase
- Revizuirea modelului de analiză
  - Aceste activități sunt ghidate de euristică
  - Calitatea produselor lor depinde de experiența dezvoltatorilor în aplicarea euristicilor/metodelor aferente

# Cazul de utilizare (CU) Raportează Urgență a SGA

Nume	Raportează Urgență
Participanti	Inițiat de Ofițer Teren Comunică cu Dispecerul
Flux de evenimente	<ol style="list-style-type: none"><li>Ofițerul activează funcția Raportează urgență a terminalului său.</li><li>Sistemul SGA afișează un formular Ofițerului. Formularul include componente privind tipul urgenței (general, incendiu, accident auto), nivelul de alertă, locația, descrierea și resursele solicitate.</li><li>Ofițerul completează formularul, inserând cel puțin tipul urgenței și descrierea situației. El poate descrie și soluții viabile la situația de urgență și poate solicita resurse specifice. După completare, Ofițerul trimite formularul.</li><li>Sistemul preia informația din formular și notifică Dispecerul.</li><li>Dispecerul verifică informația primită și creează un nou Eveniment în baza de date prin invocarea cazului de utilizare DeschideCazNou. Toate informațiile din formularul primit sunt asociate automat evenimentului creat. Dispecerul alege un răspuns prin alocarea de resurse la eveniment (prin cazul de utilizare AlocareResurse) și confirmă primirea formularului printr-un mesaj către ofițer.</li><li>Ofițerul primește confirmarea și răspunsul ales.</li></ol>

...

## Identificarea claselor *entity*

- Procesul de identificare pornește de la descrierea cazurilor de utilizare
- *Analiza limbajului natural* (eng. *natural language analysis* [Abbot 1983]) oferă un set util de euristici pentru identificarea claselor/obiectelor, atributelor, operațiilor, relațiilor și constrângerilor
  - Părților de vorbire le corespund tipuri de elemente din model

Parte de vorbire	Componentă din model	Exemple
Substantiv propriu	Obiect	Alice
Substantiv comun	Clasă sau atribut	Ofițer din teren
		Descrierea evenimentului
Verb "a face"	Operație sau asociere	Creează, trimite
Verb "a fi"	Moștenire	Este un/o
Verb "a avea"	Agregare	Are, constă din
Verb modal	Constrângere	Trebuie să fie

## Identificarea claselor *entity* (cont.)

- Avantaje
  - Metodă focusată pe terminologia utilizatorilor
  - Rezultate bune atunci când se dorește identificarea claselor candidat pe baza unor descrieri scurte (fluxul unui scenariu sau al unui caz de utilizare)
- Dezavantaje
  - Calitatea modelului obiectual e dependentă de stilul de specificare al analistului (claritate, consecvență în utilizarea termenilor)
  - Numărul substantivelor este, de regulă, mai mare decât cel al claselor (incluzând sinonime sau attribute ale altor clase)
- Euristică suplimentară pentru identificarea claselor *entity*
  - Termeni pe care dezvoltatorii și utilizatorii trebuie să îi clarifice pentru a înțelege cazul de utilizare
  - Substantive care se repetă în descrierea cazului de utilizare (ex: *Eveniment*)
  - Entități din lumea reală pe care sistemul trebuie să le gestioneze (ex.: *Dispecer, Resursă*)
  - Activități din lumea reală pe care sistemul trebuie să le gestioneze (ex.: *PlanDeOperațiiUrgență*)

## CU Raportează Urgență - clase entity

Nume clasă	Descriere
<i>OfițerTeren</i>	Ofițer de poliție sau pompieri la datorie. Un <i>OfițerTeren</i> este identificat prin număr de ecuson și nu poate fi alocat la două <i>Evenimente</i> simultan.
<i>RaportUrgență</i>	Raport initial legat de un <i>Eveniment</i> , trimis de un <i>OfițerTeren</i> unui <i>Dispecer</i> . Primirea unui <i>RaportUrgență</i> determină, de obicei, crearea unui <i>Eveniment</i> de către <i>Dispecer</i> . Un <i>RaportUrgență</i> conține un nivel de urgență, un tip (incendiu, accident rutier, etc.), o locație și o descriere.
<i>Dispecer</i>	Ofițer de poliție care gestionează <i>Evenimentele</i> . Un <i>Dispecer</i> deschide, documentează și închide un <i>Eveniment</i> , ca și răspuns al unui <i>RaportUrgență</i> sau al altui tip de comunicare cu un <i>OfițerTeren</i> . Un <i>Dispecer</i> este identificat prin număr de ecuson.
<i>Eveniment</i>	Situație care necesită intervenția unui <i>OfițerTeren</i> . Un <i>Eveniment</i> poate fi raportat în sistem de către un <i>OfițerTeren</i> , sau de către orice altă entitate externă. Un <i>Eveniment</i> constă dintr-o descriere, un răspuns, o stare (deschis, închis, documentat), o locație și un număr de <i>Ofițeri Teren</i> implicați.

# Identificarea claselor *boundary*

---

- Clasele *boundary* reprezintă interfața sistemului cu actorii
  - În cadrul fiecărui caz de utilizare, fiecare actor interacționează cu cel puțin un obiect de tip *boundary*
  - Obiectele *boundary* colectează inputul actorilor și îl transformă într-o formă utilizabilă de către obiectele *entity* și *control*
- Euristică de determinare a claselor *boundary*
  - Identificarea controalelor de care utilizatorii au nevoie pentru a iniția un caz de utilizare (ex.: *ButonRaportareUrgență*)
  - Identificarea formelor de care utilizatorii au nevoie pentru a introduce date în sistem (ex.: *FormaRaportUrgență*)
  - Identificarea notificărilor și mesajelor folosite de către sistem, pentru a răspunde actorilor (ex.: *NotificareConfirmare*)
  - În situația în care un caz de utilizare implică mai mulți actori, identificarea terminalelor acestora, pentru a referi interfața utilizator aferentă (ex.: *StatiaDispecer*)
  - Clasele *boundary* reprezintă elemente de interfață de granularitate mare
  - Elementele de interfață trebuie descrise folosind exclusiv vocabularul utilizatorilor

## CU Raportează Urgență - clase boundary

Nume clasă	Descriere
<i>ButonRaportareUrgență</i>	Buton utilizat de <i>OfițerTeren</i> pentru a iniția cazul de utilizare aferent.
<i>FormăRaportUrgență</i>	Formă deschisă unui <i>OfițerTeren</i> pe o <i>StațieOfițerTeren</i> în momentul selectării funcției <i>RaporteazăUrgență</i> . Forma conține câmpuri aferente tuturor atributelor unui raport de urgență și un buton pentru a fi trimisă.
<i>FormăEveniment</i>	Formă utilizată pentru crearea unui <i>Eveniment</i> . Este prezentată unui <i>Dispecer</i> , pe o <i>StațieDispecer</i> , în momentul primirii unui <i>RaportUrgență</i> . <i>Dispecerul</i> o utilizează și pentru alocarea resurselor și trimiterea confirmării către <i>OfițerTeren</i> .
<i>NotificareConfirmare</i>	Notificare utilizată pentru a afișa confirmarea <i>Dispecerului</i> către <i>Ofițerul Teren</i>
<i>StațieDispecer</i>	Calculator utilizat de către <i>Dispecer</i>
<i>StațieOfițerTeren</i>	Calculator portabil utilizat de către <i>OfițerTeren</i>

## Identificarea claselor *control*

---

- Obiectele *control* sunt responsabile de coordonarea obiectelor *boundary* și *entity*
  - Obiectele *control* nu au, de obicei, un corespondent în lumea reală
  - Un obiect *control* poate fi asociat, de regulă, unui caz de utilizare: este creat la începutul cazului de utilizare și distrus la finalizarea acestuia.
  - Ex.: Obiectele *control* gestionează comportamentul legat de succesiunea formelor, cozi istoric/ undo, transmiterea informației într-un mediu distribuit, etc.
- Euristică pentru determinarea obiectelor *control*
  - Identificarea unui control per caz de utilizare
  - Identificarea unui control per actor într-un caz de utilizare
  - Durata de viață a obiectului *control* corespunde durei de execuție a cazului de utilizare sau durei unei sesiuni utilizator

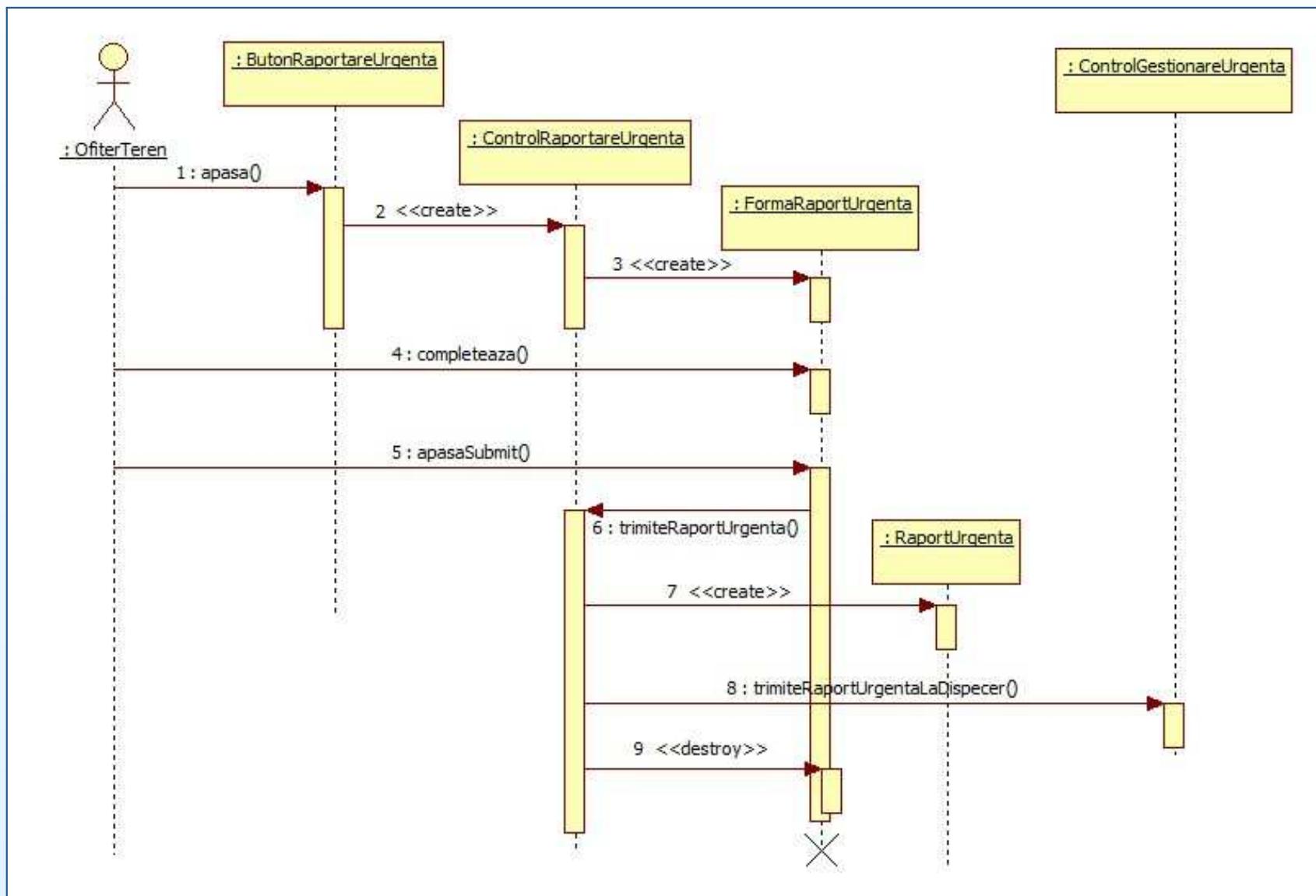
## CU Raportează Urgență - clase control

Nume clasă	Descriere
<i>ControlRaportareUrgență</i>	Gestioneză raportarea unei urgențe de pe o <i>StațieOfițerTeren</i> . Un obiect de acest tip este creat în momentul în care un <i>OfițerTeren</i> apasă butonul "Raportează Urgență". Ulterior, acesta creează un obiect de tip <i>FormăRaportUrgență</i> , pe care o afișează ofițerului. La trimitera formularului, acest obiect colectează informația completată, creează un obiect de tip <i>RaportUrgență</i> și îl trimită <i>Dispecerului</i> . Obiectul control așteaptă apoi o confirmare de la stația dispecerului. În momentul primirii confirmării, creează un obiect de tip <i>NotificareConfirmare</i> și o afișează ofițerului.
<i>ControlGestionareUrgență</i>	Gestioneză raportarea unei urgențe pe o <i>StațieDispecer</i> . Un obiect de acest tip este creat în momentul primirii unui <i>RaportUrgență</i> . Ulterior, acesta creează o <i>FormăEveniment</i> și o afișează <i>Dispecerului</i> . După ce dispecerul creează un <i>Eveniment</i> , alocă resurse și trimită o confirmare, obiectul control transmite confirmarea <i>Ofițerului Teren</i> .

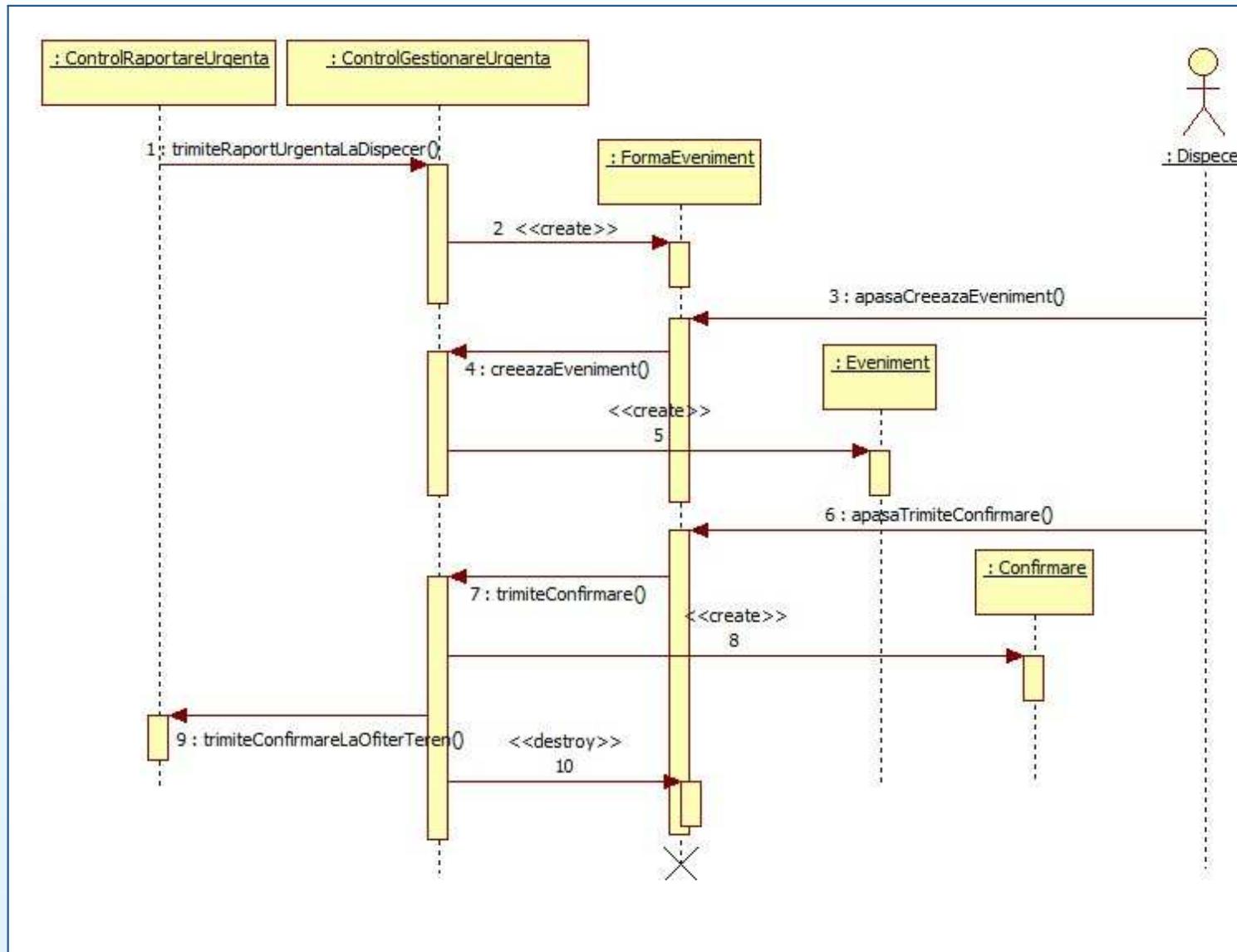
# Realizarea diagramelor de secvență

- Diagramele de secvență realizează legătura dintre clase/ obiecte și cazurile de utilizare/ scenariile descrise
  - Prezintă modul în care comportamentul descris de un scenariu/ caz de utilizare este distribuit între obiectele participante
  - În etapa de analiză, permit identificarea unor descrieri de comportament ambigu sau a unor clase/ obiecte participante omise
  - Datorită notației utilizate, nu sunt, în general, o metodă de comunicare cu clienții/ utilizatorii la fel de eficientă ca și scenariile/ cazurile de utilizare
- Euristică de realizare a diagramelor de secvență de analiză
  - Prima coloană trebuie să corespundă actorului care a inițiat cazul de utilizare
  - A doua coloană trebuie să corespundă obiectului *boundary* folosit de actor pentru a iniția cazul de utilizare
  - A treia coloană trebuie să corespundă obiectului *control* care gestionează cazul de utilizare
  - Obiectele *control* sunt create de către obiectele *boundary* care inițiază cazurile de utilizare
  - Ulterior, obiectele *boundary* sunt create de către obiecte *control*
  - Obiectele *entity* sunt accesate de către obiectele *boundary* și *control*, dar nu accesează niciodată obiecte *boundary* sau *control*

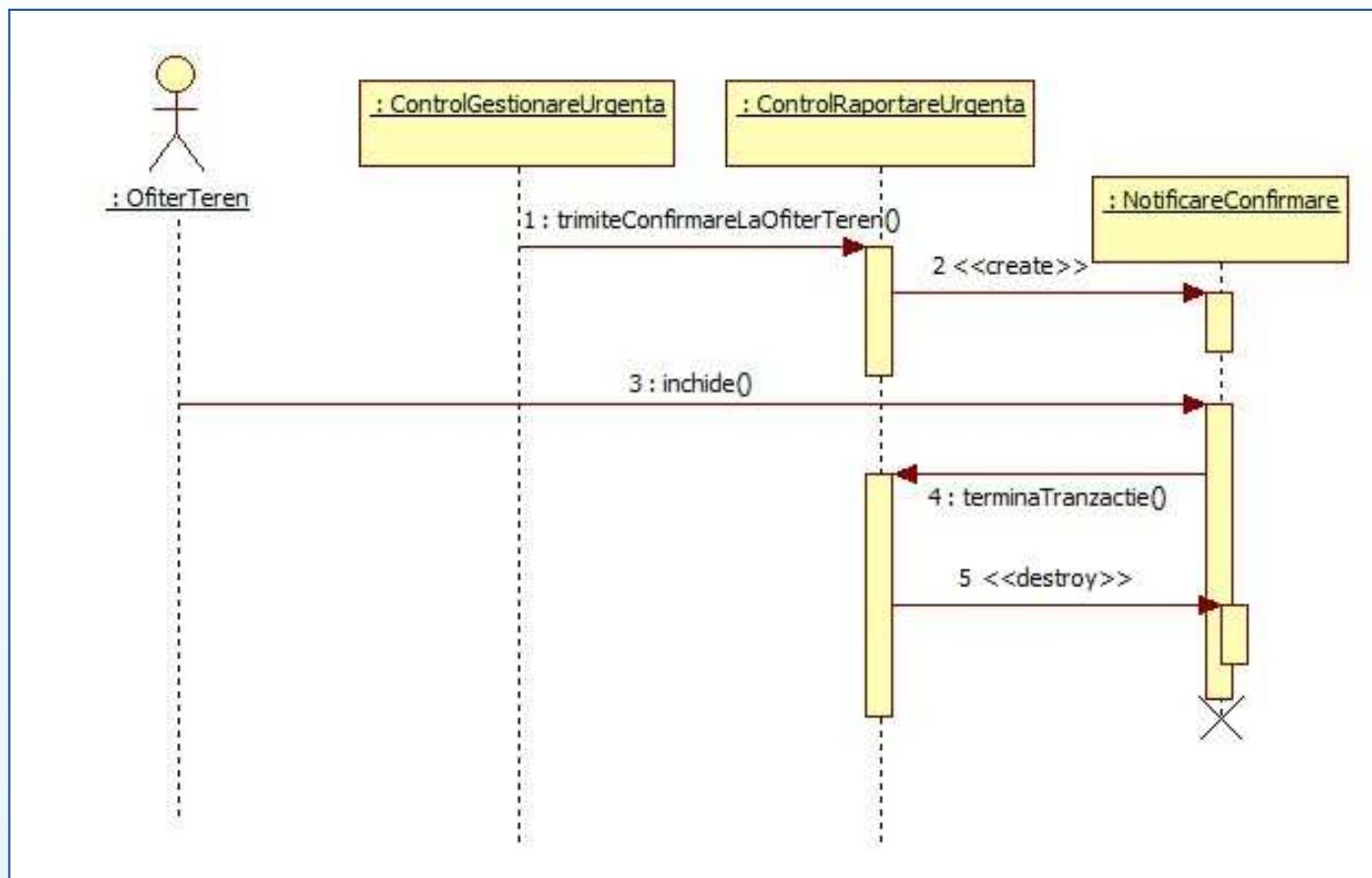
# CU Raportează Urgență - diagramă de secvență



## CU Raportează Urgență - diagramă de secvență (cont.)



## CU Raportează Urgență - diagramă de secvență (cont.)



## CU RaporteazăUrgență - rafinare

- Clasă *entity* nouă, identificată ca urmare a formalizării descrierii cazului de utilizare folosind diagrama de secvență

Nume clasă	Descriere
<i>Confirmare</i>	Răspuns al unui <i>Dispecer</i> la un <i>RaportUrgență</i> trimis de către un <i>OfițerTeren</i> . Trimînd o <i>Confirmare</i> , <i>Dispecerul</i> îi comunică ofițerului că a primit raportul, a creat un <i>Eveniment</i> și i-a asignat resurse. <i>Confirmarea</i> include specificarea resurselor alocate și timpul estimat al sosirii lor.

- Rafinarea descrierii cazului de utilizare

Nume	<i>RapoteazăUrgență</i>
...	...
Flux de evenimente	<p>5. ... <i>Confirmarea</i> îi indică ofițerului că raportul a fost primit, evenimentul a fost creat și resursele au fost alocate. <i>Confirmarea</i> include specificarea resurselor (ex. camion pompieri) și momentul estimativ al sosirii lor.</p> <p>...</p>
...	...

# Identificarea asocierilor

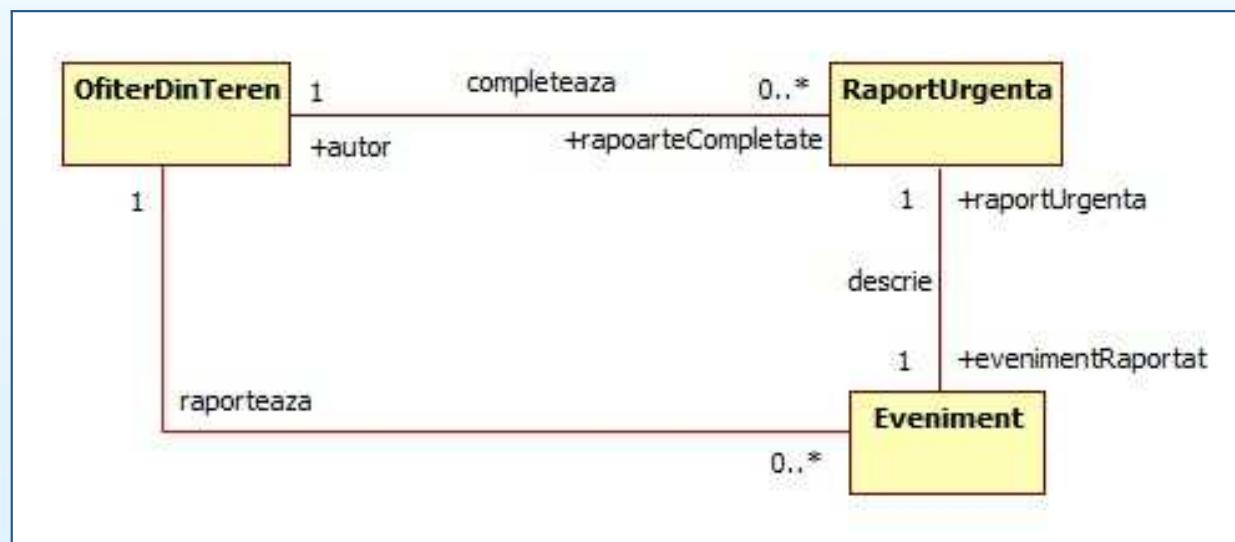
- O *asociere* indică o relație între două sau mai multe clase
  - Ex.: Un ofițer din teren scrie rapoarte de urgență.



- Proprietăți ale unei asocieri
  - nume
  - roluri
  - multiplicitate
- Avantaje ale modelării asocierilor în analiză
  - Clarificarea modelului de analiză prin explicitarea modului de relaționare a obiectelor
    - Raportul este scris de către un ofițer din teren și nu de către un dispecer
  - Investigarea cazurilor limită
    - Există rapoarte cu mai mulți autori? Dar anonime?

## Identificarea asocierilor (cont.)

- Euristică pentru reprezentarea asocierilor
  - Examinarea structurilor verbale
  - Numirea explicită a asocierilor și rolurilor
  - Utilizarea calificatorilor pentru a identifica spații de nume și atribute cheie
  - Eliminarea tuturor asocierilor redundante (derivabile din alte asocieri)
  - Reprezentarea multiplicităților doar după stabilizarea mulțimii de asocieri
  - Un număr exagerat de asocieri generează redundanță și afectează negativ inteligențialitatea modelului
- Model cu asocieri redundante

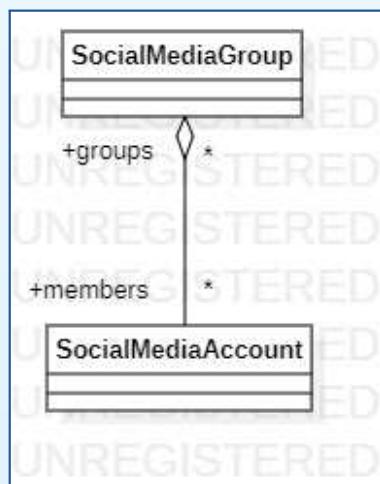


# Identificarea agregărilor

- Agregarea (varianta "simplă"/ partajată) reprezintă un tip particular de asociere, care denotă o relație de tip parte-întreg
- Compunerea este o agregare mai "puternică", în care partea este, la un moment dat, conținută într-un singur întreg, iar existența părților e condiționată de întreg

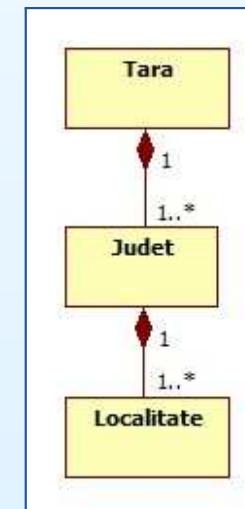
## Ex.: Agregare

- Pe rețelele de socializare, o persoană este, în general, înscrișă în mai multe grupuri. După eventuala ștergere a unui astfel de grup, ea ramâne parte a celorlalte.



## Ex.: Compunere

- O localitate e parte a unui singur județ, cel din urmă fiind parte a unei singure țări la un moment dat (dar există posibilitatea schimbării granițelor politice)



# Identificarea atributelor

- Atributele reprezintă proprietăți ale obiectelor individuale



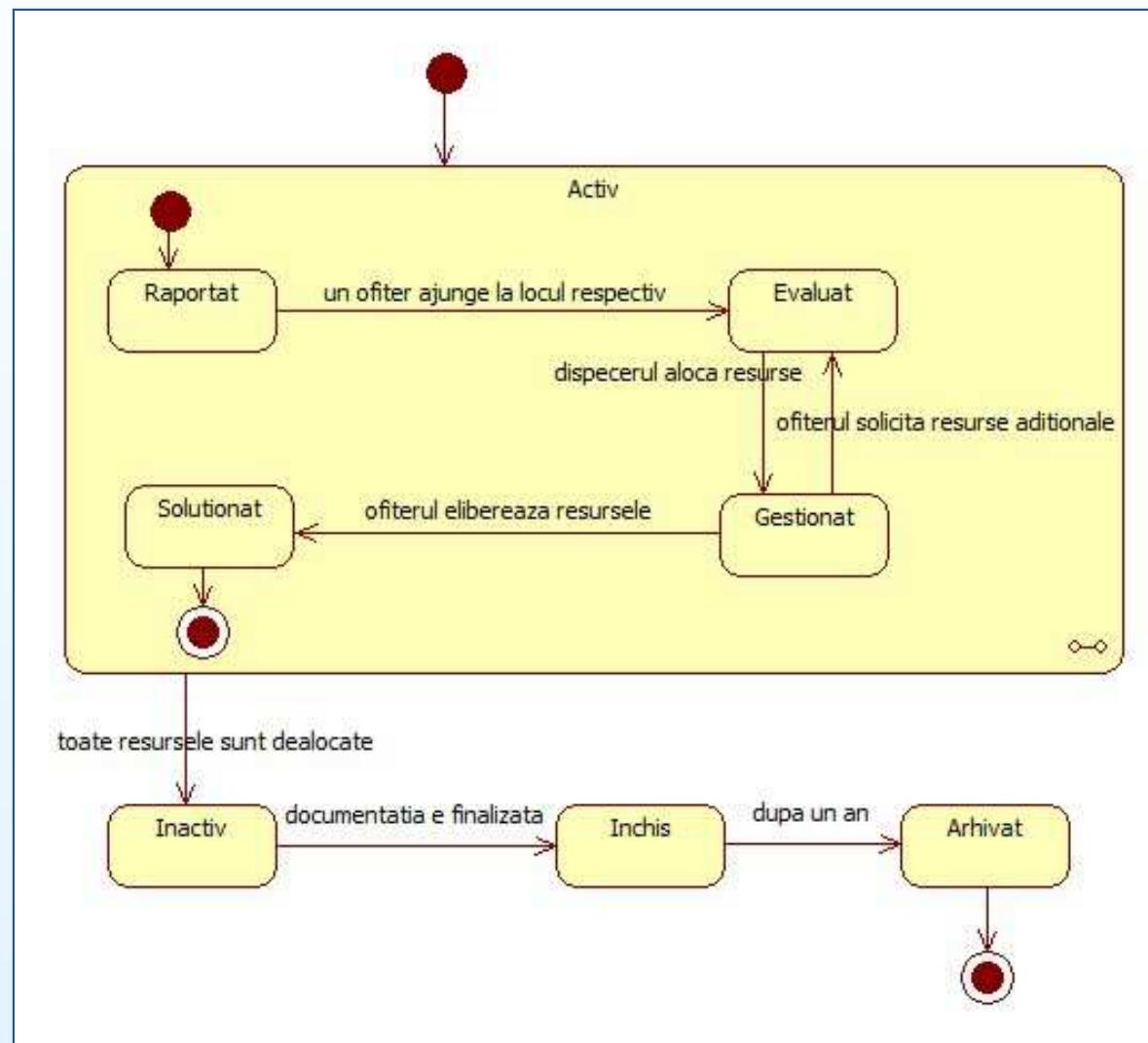
- Euristică pentru identificarea atributelor
  - Examinarea construcțiilor substantivale posesive (ex.: descrierea urgenței)
  - Reprezentarea proprietăților persistente ca și atrbute
  - Identificarea doar a celor atrbute relevante pentru sistemul în cauză
  - Amânarea identificării atrbutorilor mai puțin relevante din punct de vedere al funcționalității până după stabilizarea modelului obiectual
  - Descrierea sumară a fiecărui atrbut
  - Proprietățile de tip obiect/ referință nu se reprezintă ca și atrbute, în acest caz se folosesc asocieri!

# Modelarea comportamentului cu diagrame de stări

---

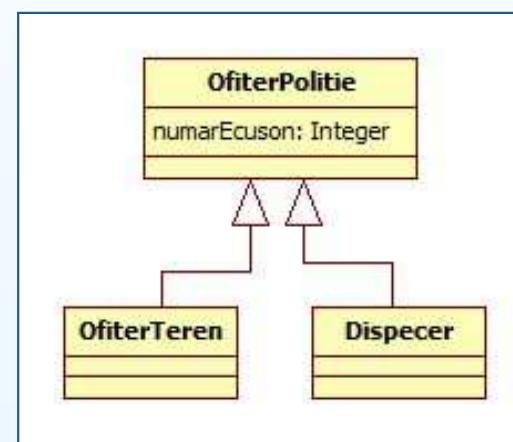
- Modelarea comportamentului
  - Diagrame de secvență - prezintă comportamentul din perspectiva unui singur caz de utilizare
  - Diagrame de tranziție a stărilor - prezintă comportamentul din perspectiva unui singur obiect
- Rolul diagramelor de tranziție a stărilor în etapa de analiză constă în identificarea unor cazuri de utilizare omise sau a unor omisiuni în descrierea cazurilor de utilizare existente
- Nu este necesară construirea unei diagrame de tranziție a stărilor pentru fiecare clasă a modelului obiectual, doar pentru cele ale căror obiecte au durată de viață mare și comportament complex, dependent de stare
  - Cel mai frecvent pentru obiecte *control*, mai puțin frecvent pentru obiecte *entity* și aproape niciodată pentru obiecte *boundary*

## CU Raportează Urgență - diagrama de stări pentru *Eveniment*

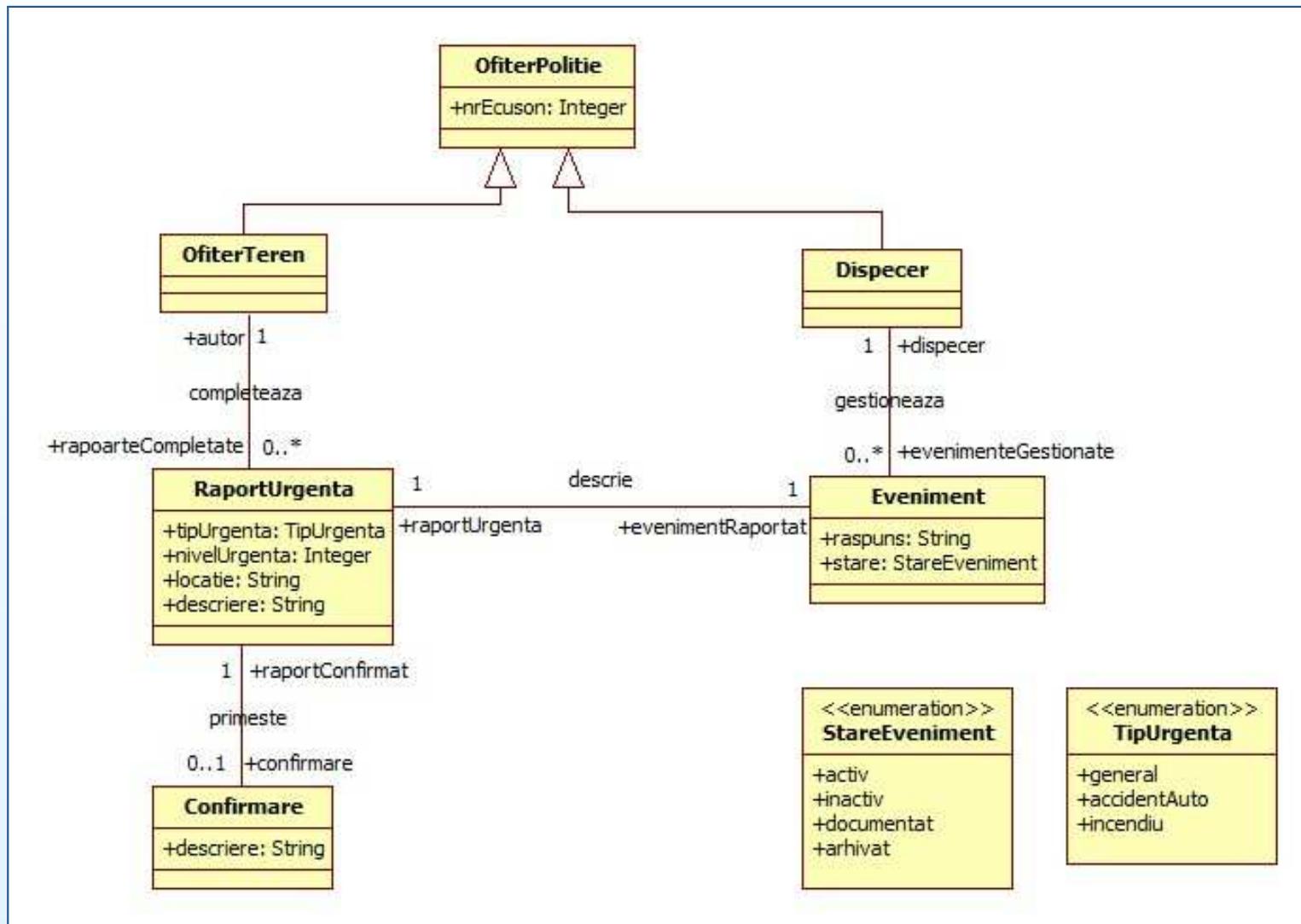


## Identificarea ierarhiilor de clase

- Rolul generalizărilor îl constituie eliminarea redundanțelor din modelul obiectual
- Ex.:



## CU Raportează Urgență - model al claselor entity (conceptual)



# Revizuirea modelului de analiză

---

- Modelul de analiză se dezvoltă iterativ-incremental
- Odata ce devine stabil, urmează revizuirea acestuia
  - Revizuire internă (dezvoltatori) + revizuire dezvoltatori și client
  - E specificarea cerințelor corectă, completă, consistentă și neambiguă? Sunt cerințele realiste și verificabile?
  - Revizuirea e facilitată de existența unor liste de întrebări posibile
- Asigurarea *corectitudinii*
  - Este dicționarul claselor *entity* înțeles de către client?
  - Corespund clasele abstracte unor concepte utilizator?
  - Sunt toate descrierile conforme cu definițiile date de utilizatori?
  - Au toate obiectele *entity* și *boundary* ca și nume construcții substantivale sugestive?
  - Au toate cazurile de utilizare și obiectele *control* ca și nume construții verbale sugestive?
  - Sunt toate cazurile de eroare descrise și gestionate?

# Revizuirea modelului de analiză (cont.)

---

- Asigurarea *completitudinii*

- Pentru fiecare obiect: Este folosit de vreun caz de utilizare? În cadrul cărui caz de utilizare este creat? modificat? distrus? Poate fi accesat pornind de la un obiect *boundary*?
- Pentru fiecare atribut: Unde ii este atribuită o valoare? Care este tipul său? Poate reprezenta un calificator?
- Pentru fiecare asociere: Când este traversată? De ce a fost aleasă respectiva multiplicitate?
- Pot fi calificate asocierile *one-to-many* și *many-to-many*?
- Pentru fiecare obiect *control*: Are asociările necesare accesării tuturor obiectelor din respectivul caz de utilizare?

- Asigurarea *consistenței*

- Există mai multe clase sau cazuri de utilizare cu același nume?
- Denotă entitățile cu nume similare concepte înrudite?
- Există obiecte cu atrbute și asocieri similare și care nu fac parte din aceeași ierarhie de moștenire?

## Revizuirea modelului de analiză (cont.)

---

- Asigurarea *realismului*
  - Pot fi îndeplinite cerințele legate de performanță și fiabilitate?
  - Au fost aceste cerințe verificate folosind prototipuri pe platforma hardware aleasă?

# Şablonul documentului de analiză a cerințelor

---

- 1. Introducere
  - 1.1 Scopul sistemului
  - 1.2 Obiectivele și criteriile de succes ale proiectului
  - 1.3 Definiții, acronime și abrevieri
  - 1.4 Referințe
  - 1.5 Sumar
- 2. Sistemul curent
- 3. Sistemul propus
  - 3.1 Sumar
  - 3.2 Cerințe funcționale
  - 3.3 Cerințe nefuncționale
    - 3.3.1 Utilizabilitate
    - 3.3.2 Fiabilitate
    - 3.3.3 Performanță
    - 3.3.4 Suportabilitate
    - 3.3.5 Implementare
    - 3.3.6 Interfață

# Şablonul documentului de analiză a cerințelor (cont.)

---

- 3.3.7 Instalare
- 3.3.8 Cerințe legale
- 3.4 Modele
  - 3.4.1 Scenarii
  - 3.4.2 Modelul cazurilor de utilizare
  - 3.4.3 Modelul obiectual
  - 3.4.4 Modelul dinamic
  - 3.4.5 Prototipul interfeței utilizator
- 4. Glosar

## Referinte

---

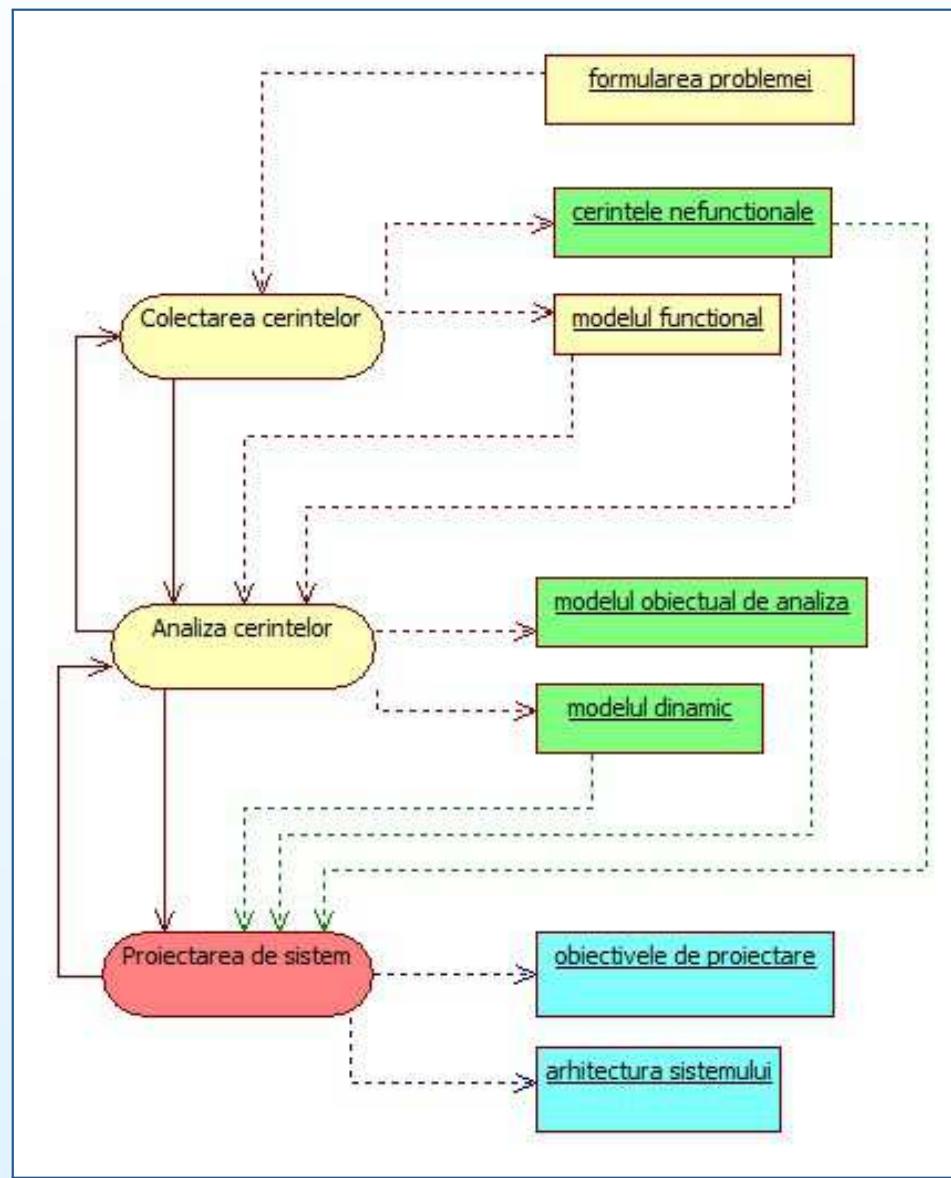
- [Abbott, 1983] R. Abbott, *Program design by informal English descriptions*, Communications of the ACM, Vol. 26, No. 11, 1983.

## Curs 5

### Proiectarea de sistem (I)

*Suport de curs bazat pe B. Bruegge and A.H. Dutoit  
"Object-Oriented Software Engineering using UML, Patterns, and Java"*

# Proiectarea de sistem



## Proiectarea de sistem (cont.)

- Procesul de transformare a modelului rezultat din ingineria cerințelor într-un model arhitectural al sistemului
- Produse ale proiectării de sistem
  - *Obiectivele de proiectare* (eng. *design goals*)
    - Calități ale sistemului pe care dezvoltatorii trebuie să le optimizeze
    - Derivate din cerințele nefuncționale
  - *Arhitectura sistemului*
    - Subsistemele componente (de dimensiuni mai mici, asignabile unei subechipe de dezvoltare)
    - Responsabilitățile subsistemelor și dependențele între ele
    - Maparea subsistemelor la hardware
    - Strategii de dezvoltare: fluxul global de control, strategia de gestionare a datelor cu caracter persistent, politica de control a accesului

## Proiectarea de sistem (cont.)

- Activități ale proiectării de sistem
  - *Identificarea obiectivelor de proiectare*
    - Identificarea și prioritizarea acelor calități ale sistemului pe care dezvoltatorii trebuie să le optimizeze
  - *Descompunerea inițială a sistemului*
    - Pe baza modelului funcțional și a modelelor de analiză
    - Bazată pe utilizarea unor stiluri arhitecturale standard
  - *Rafinarea descompunerii inițiale în vederea atingerii obiectivelor de proiectare*
    - Rafinarea arhitecturii de la pasul anterior, până la îndeplinirea tuturor obiectivelor de proiectare
- Analogie cu proiectarea arhitecturală a unei clădiri
  - Componente: camere vs. subsisteme
  - Interfețe: pereți/uși vs. servicii
  - Reproiectare: mutarea pereților vs. schimbarea subsistemelor/interfețelor

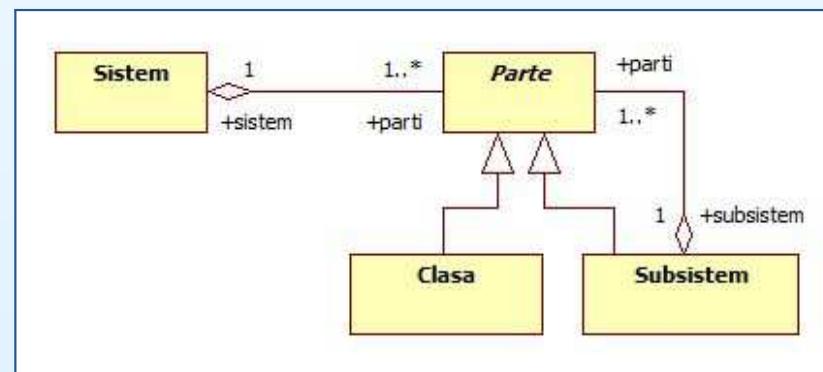
# Concepte în proiectarea de sistem

---

- Subsisteme și clase
- Servicii, interfețe și API-uri
- Coeziune și cuplare
- Stratificare și partiționare
- Stiluri arhitecturale și arhitecturi software

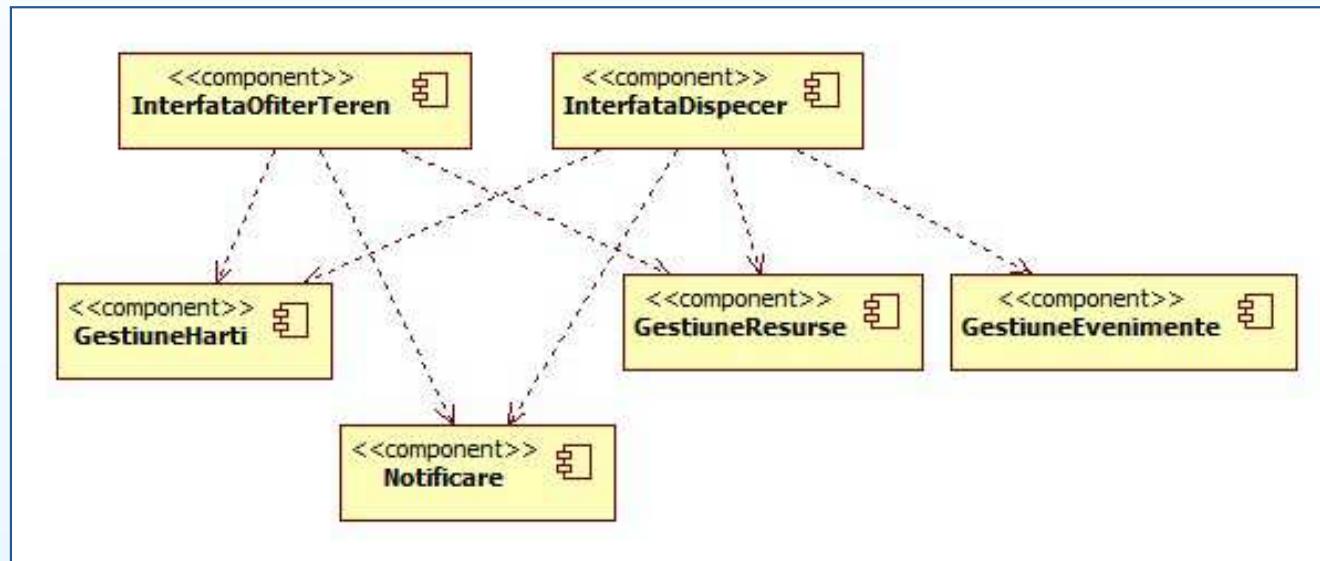
# Subsisteme și clase

- *Subsistem* = parte înlocuibilă a unui sistem (constând într-un număr de clase din domeniul soluției), caracterizată prin interfețe bine definite, care încapsulează starea și comportamentul claselor componente
  - Descompunerea în subsisteme permite gestionarea complexității ("divide et impera")
  - Un subsistem se dezvoltă, de regulă, de către un programator sau o echipă de dezvoltare
  - Prin descompunerea sistemului în subsisteme (relativ) independente, se permite dezvoltarea (relativ) concurentă a acestora
  - Sistemelor complexe le corespund mai multe nivale de descompunere (*Composite pattern*)



## Subsisteme și clase (cont.)

- Ex.: descompunere în subsisteme a sistemului SGA (diagramă UML de componente)



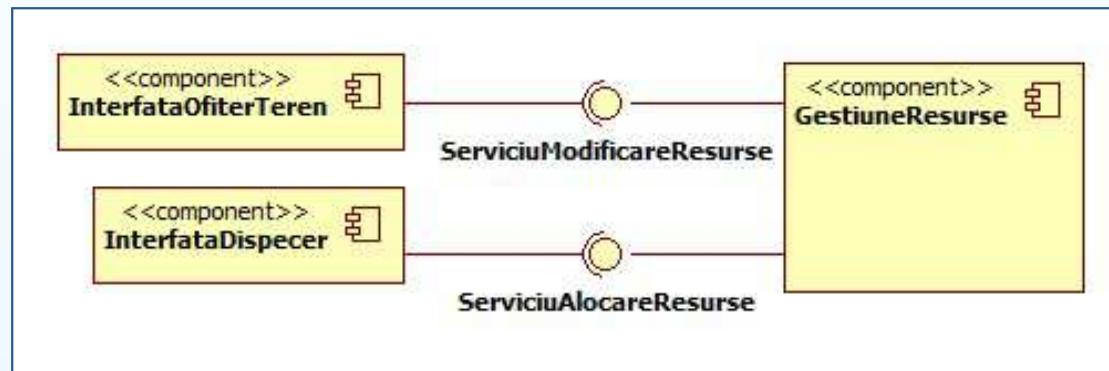
- Subsistemele sunt reprezentate ca și componente UML, cu relații de dependență între ele
- O componentă UML poate reprezenta
  - O componentă logică = un subsistem ce nu are un echivalent runtime
  - O componentă fizică = un subsistem ce are un echivalent runtime

# Servicii, interfețe și API-uri

- *Serviciu* = mulțime de operații înrudite (definite cu același scop)
  - Subsistemele sunt caracterizate de serviciile oferite altor subsisteme
  - Ex.: un subsistem care oferă un serviciu de notificare va defini operații de tipul *LookupChannel()*, *SubscribeToChannel()*, *UnsubscribeFromChannel()*, *SendNotification()*
  - Serviciile se identifică în timpul proiectării de sistem
- *Interfață* (a unui subsistem) = mulțime de operații UML înrudite, complet specificate (nume, tipuri parametri, tip returnat)
  - Rafinare a unui serviciu, specifică interacțiunile și fluxul de informații dinspre și înspre frontieră subsistemului (nu și în interiorul acestuia)
  - Interfețele se definesc în timpul proiectării obiectuale
- *API (Application Programming Interface)* = specificare a unei interfețe subsistem într-un limbaj de programare
  - API-urile se definesc în etapa de implementare

## Servicii, interfețe și API-uri (cont.)

- Ex.: Interfețe/servicii oferite (eng. *provided*) și solicitate/utilizate (eng. *required*)



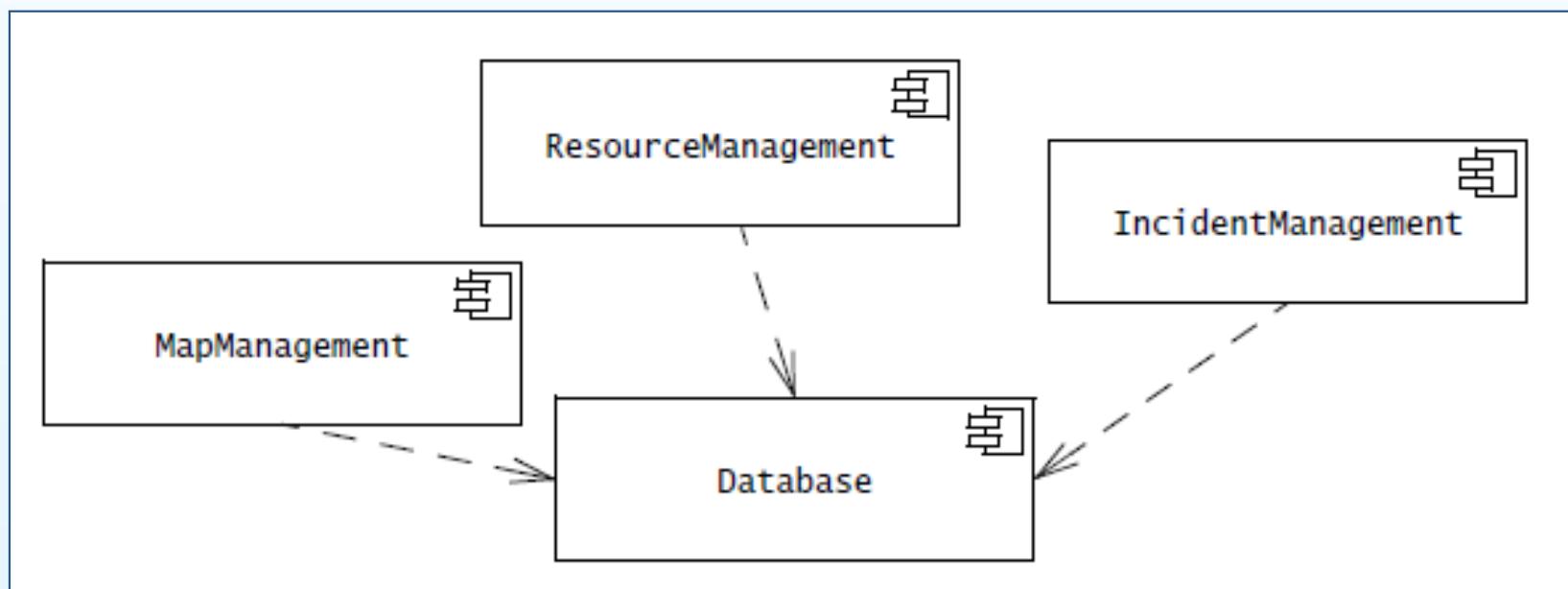
- Notația UML: conectori *ball-and-socket*
  - Ball (lollipop) = interfață oferită, socket = interfață solicitată
  - Dependențele dintre subsisteme se reprezintă prin cuplarea conectorilor ball cu cei socket
  - Reprezentare utilizată în momentul în care descompunerea în subsisteme e relativ stabilă și focusul se schimbă de pe identificarea subsistemelor pe identificarea serviciilor (anterior se folosesc relații UML de dependență)

# Coeziune și cuplare

- *Cuplare* (eng. *coupling*) = măsură a dependenței dintre două subsisteme
  - *Cuplare slabă* (eng. *low coupling*) - număr mic de dependențe (subsisteme relativ independente)
  - *Cuplare strânsă* (eng. *strong coupling*) - număr mare de dependențe (schimbările efectuate într-un sistem îl vor afecta și pe celălalt)
  - Dezirabilă într-o descompunere: cuplarea slabă
  - Reducerea cuplării conduce, în general, la creșterea complexității prin introducerea de subsisteme suplimentare
- *Coeziune* (eng. *cohesion*) = măsură a dependențelor din interiorul unui subsistem
  - *Coeziune înaltă* (eng. *high cohesion*) - subsistemul conține un număr mare de clase puternic relaționate și care efectuează sarcini similare
  - *Coeziune slabă* (eng. *low cohesion*) - subsistemul conține un număr de clase nerelaționate
  - Dezirabilă într-o descompunere: coeziunea înaltă
  - Creșterea coeziunii (prin descompunere în subsisteme cu coeziune înaltă) conduce și la creșterea cuplării, prin dependențele nou introduse

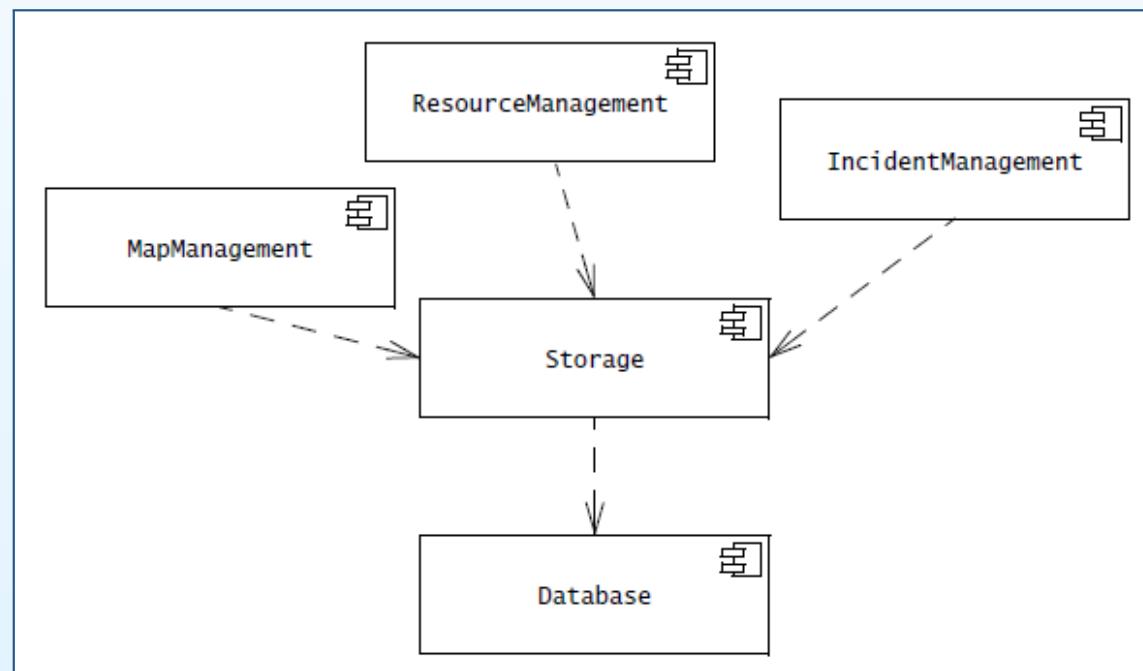
# Cuplare

- Ex.: Subsisteme stâns cuplate
  - Subsistemele de gestiune trimit SGBD-ului comenzi SQL
  - Trecerea la un alt SGBD sau schimbarea strategiei de persistență (fișiere text) determină modificări la nivelul tuturor celor trei subsisteme de gestiune



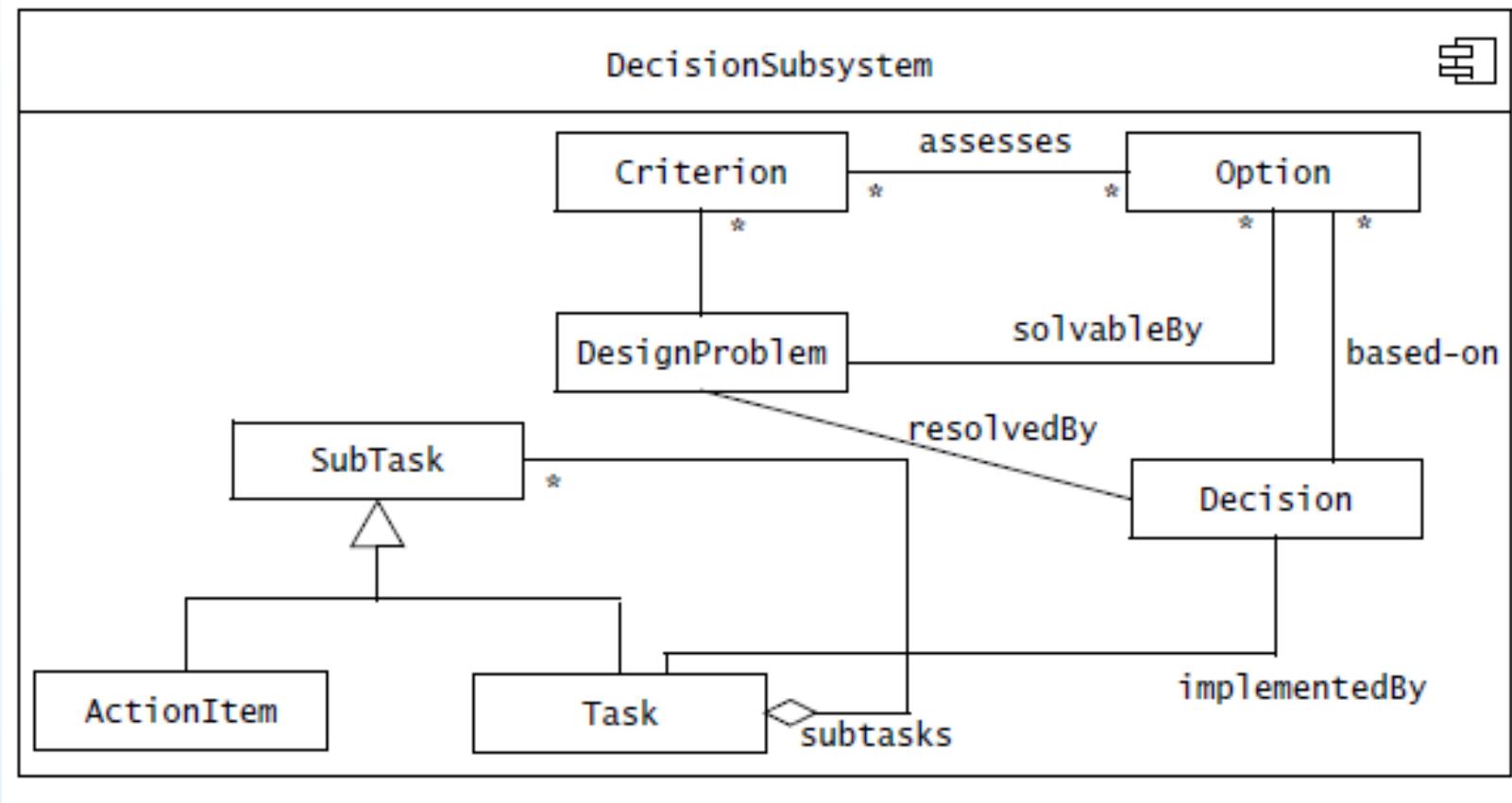
## Cuplare (cont.)

- Ex.: Reducerea cuplării prin inserarea unui subsistem suplimentar
  - Noul subsistem izolează subsistemele de gestiune de SGBD
  - Subsistemele de gestiune utilizează doar serviciile oferite de noul subsistem, care va fi responsabil cu trimiterea de comenzi SQL spre SGBD
  - Trecerea la un alt SGBD sau schimbarea strategiei de persistență va determina doar modificări la nivelul subsistemului introdus



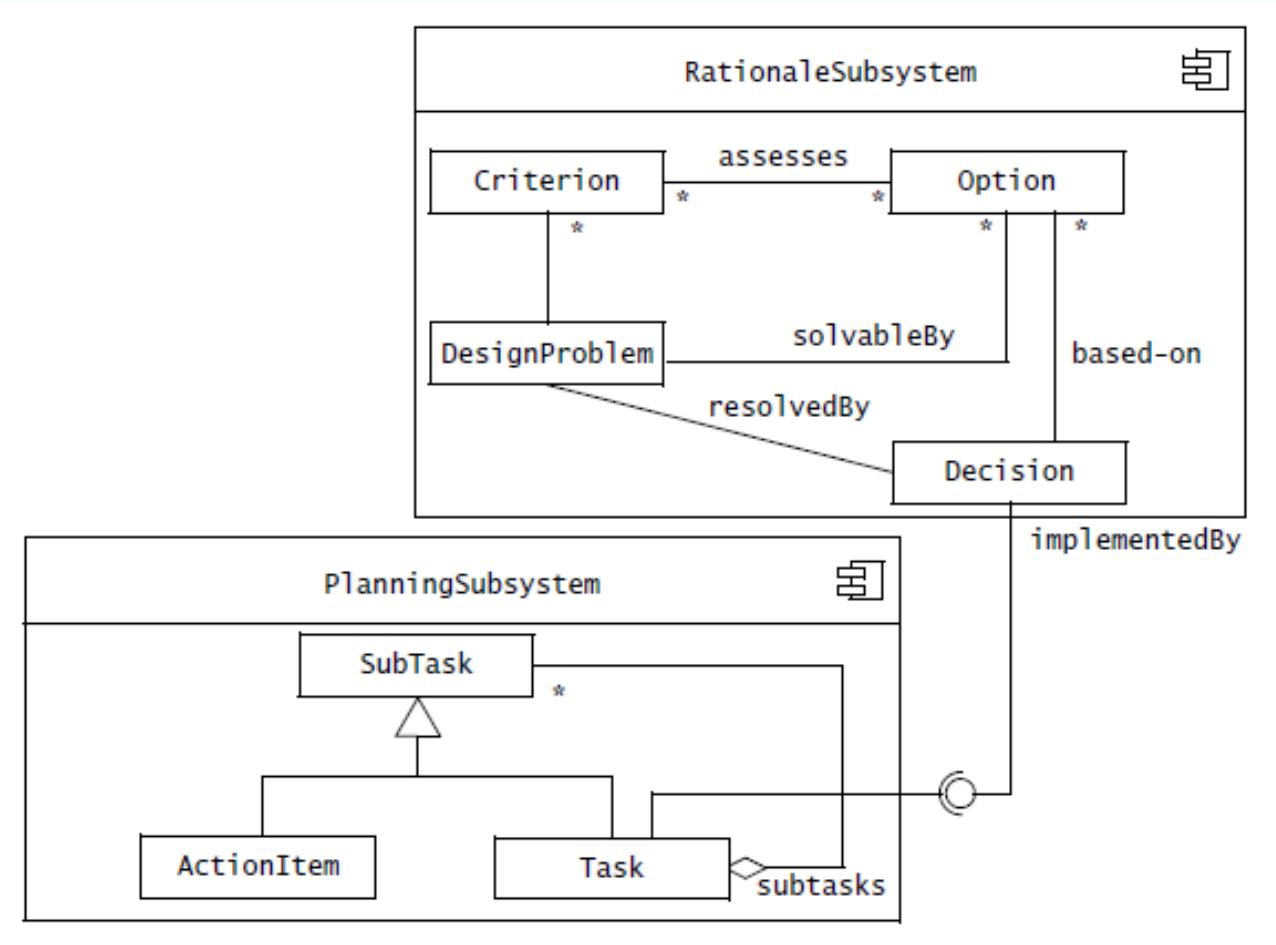
# Coeziune

- Subsistem cu coeziune slabă
  - Clasele componente pot fi partaționate în două submulțimi slab cuplate



## Coeziune (cont.)

- Creșterea coeziunii prin descompunerea subsistemului inițial



# Stratificare și partaționare

- *Descompunere ierarhică a unui sistem (stratificare)*
  - Generează o mulțime ordonată de straturi (eng. *layers*)
  - Un strat reprezintă un grup de subsisteme ce oferă servicii înrudite, eventual utilizând servicii dintr-un alt strat
  - Straturile sunt ordonate: un strat poate accesa doar servicii ale straturilor inferioare
    - *Arhitectură închisă* - fiecare strat poate accesa doar servicii din stratul imediat inferior (scop = modificabilitate/flexibilitate)
    - *Arhitectură deschisă* - un strat poate accesa servicii din oricare dintre straturile inferioare (scop = eficiență )
- *Partitionare*
  - Generează un grup de subsisteme la același nivel (eng. *peers*), fiecare dintre ele fiind responsabil de o categorie diferită de servicii
- În general, o descompunere a unui sistem complex este rezultatul atât al stratificării, cât și al partaționării

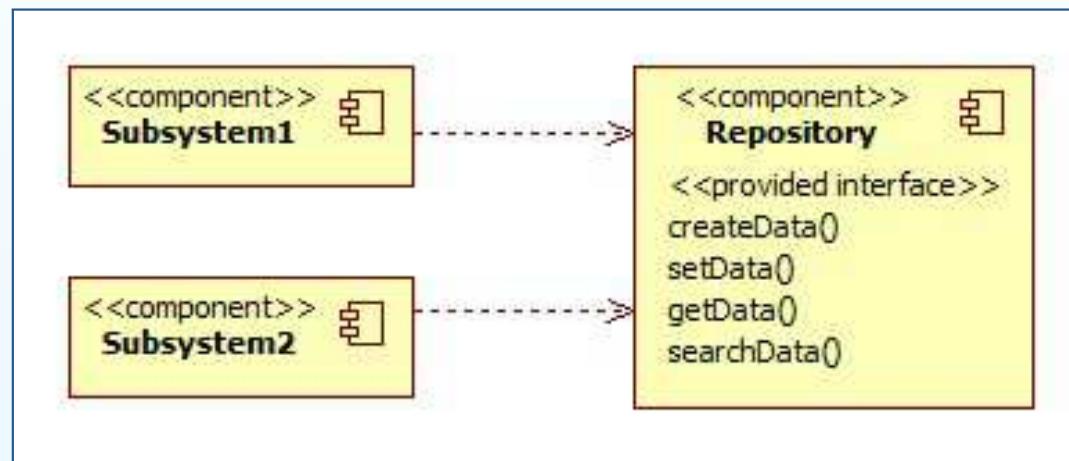
# Stiluri arhitecturale și arhitecturi software

---

- *Descompunerea sistemului* (eng. *system decomposition*) = identificarea subsistemelor, a serviciilor și a relațiilor între acestea
- *Stil arhitectural* (eng. *architectural style*) = şablon de descompunere a sistemelor
- *Arhitectură software* (eng. *software architecture*) = instanță a unui stil arhitectural
- Exemple de stiluri arhitecturale
  - Repository
  - Model-View-Controller
  - Client-Server
  - Peer-to-Peer
  - Three-tier architecture
  - Four-tier arhitecture
  - Pipes and filters

# Repository

- Subsistemele accesează și modifică o singură structură de date centralizată, denumită *repository*

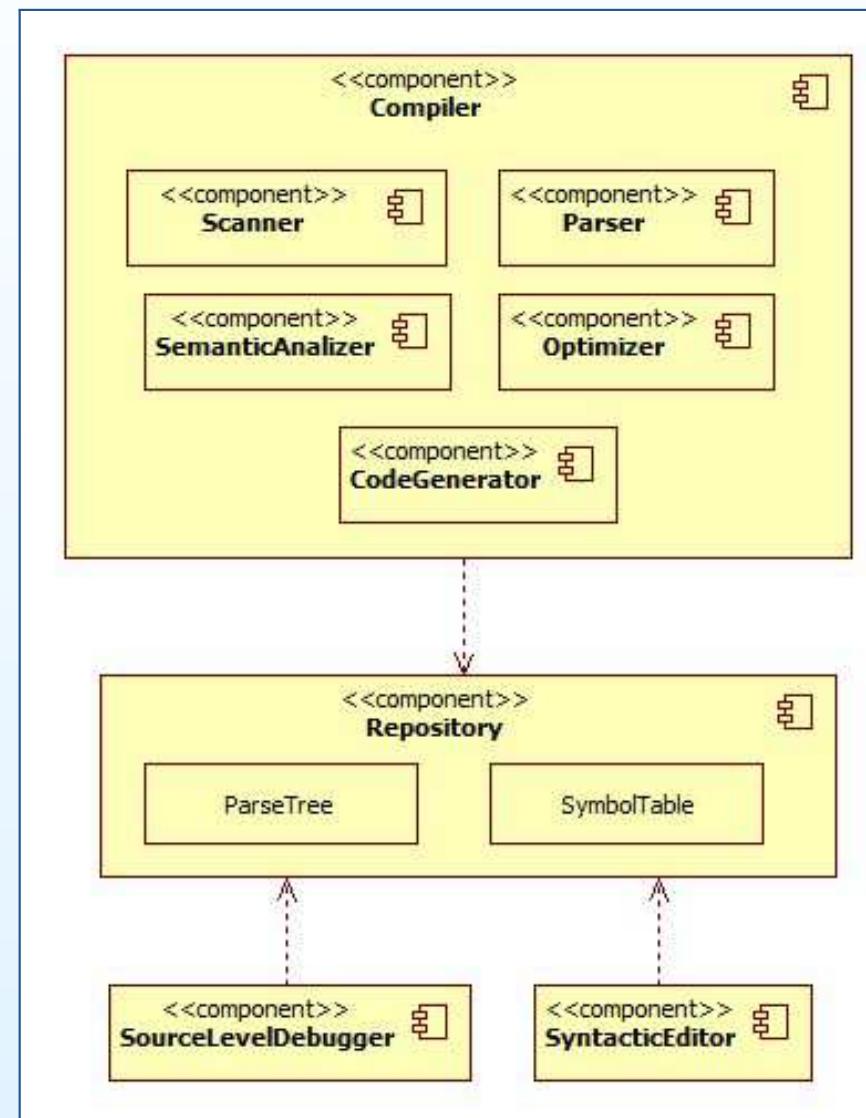


- Subsistemele sunt relativ independente și interacționează doar prin intermediul repository-ului (cuplare slabă)
- Fluxul de control poate fi dictat de repository (prin triggere) sau de către subsisteme (prin blocaje și sincronizări)

# Repository (cont.)

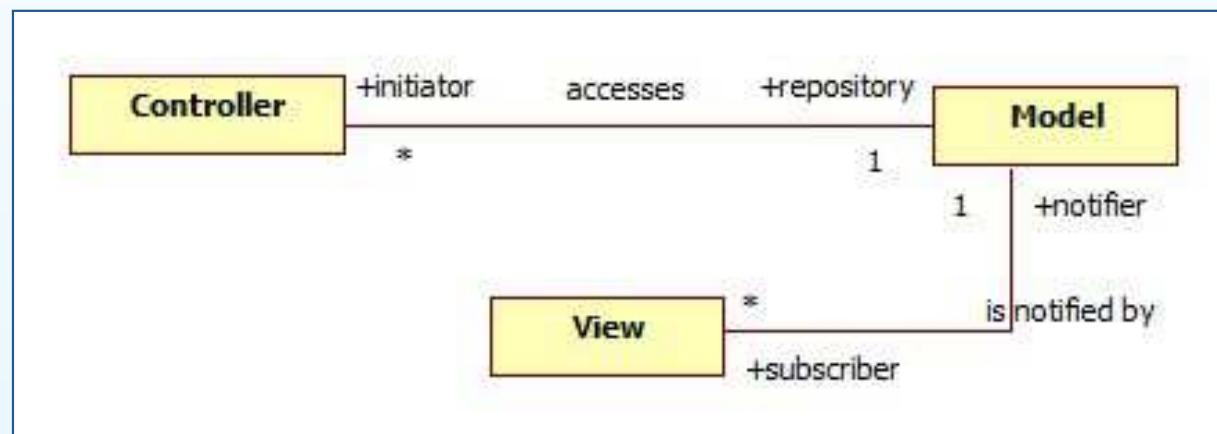
- Avantaje
  - Arhitectură utilă în cazul sistemelor cu necesități de procesare complexe, în continuă schimbare
  - Odată definit repository-ul, pot fi oferite servicii noi prin definirea de subsisteme adiționale
- Dezavantaje
  - Subsistemele și repository-ul sunt strâns cuplate, facând dificilă modificarea repository-ului fără a afecta subsistemele
  - Probleme de performanță
- Utilitate
  - Sisteme de gestiune a bazelor de date
  - Compilatoare și medii integrate de dezvoltare (eng. *Integrated Development Environments, IDEs*)

# Exemplu de arhitectură Repository pentru un IDE



# Model-View-Controller (MVC)

- Subsistemele sunt încadrate în una din trei categorii
  - *Model* - reprezintă informații/cunoștințe din domeniul problemei
  - *View* - afisează aceste informații utilizatorului
  - *Controller* - translatează interacțiunile cu *view*-ul în acțiuni asupra modelului



- Subsistemele model nu depind de nici un subsistem view sau controller
  - Modificările produse la nivelul acestora sunt propagate în subsistemele view prin intermediul unui protocol de înscriere/ notificare
  - Funcționalitatea de înscriere/notificare este realizată, de obicei, cu ajutorul şablonului de proiectare *Observer*

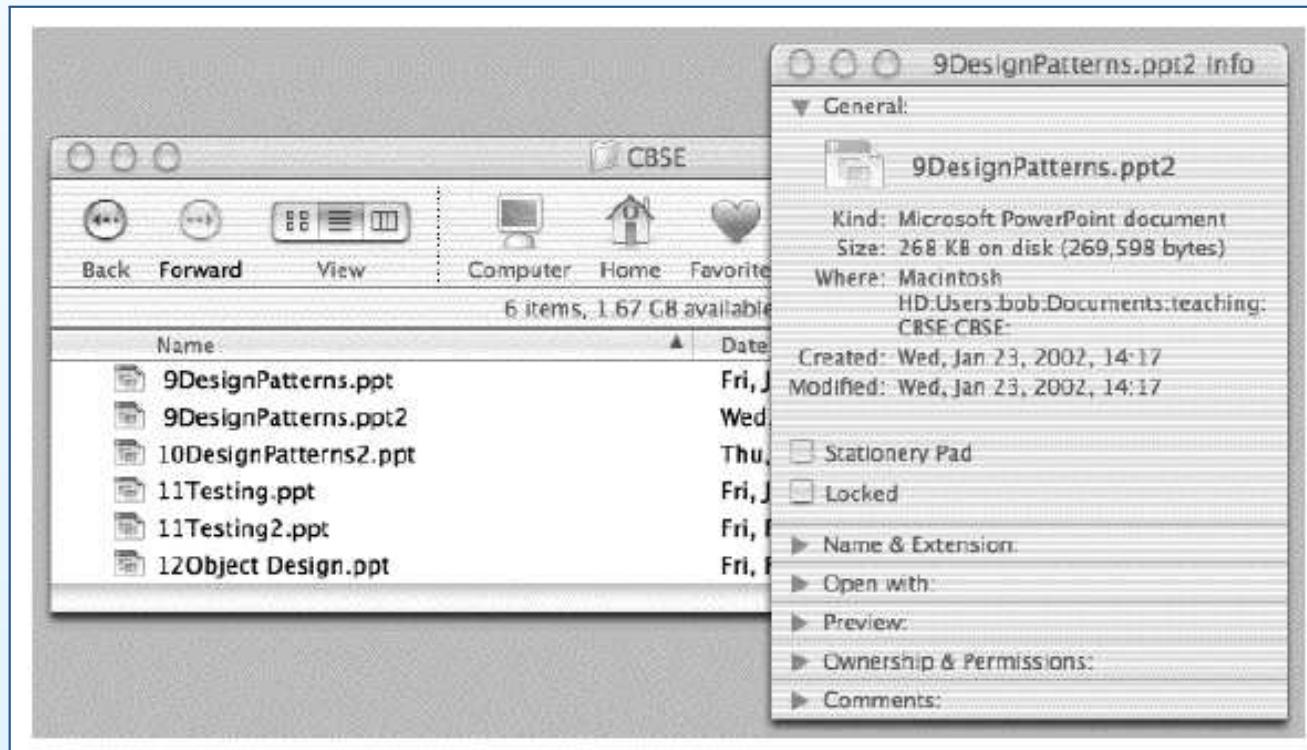
# Model-View-Controller (cont.)

---

- Justificare
  - Interfața utilizator (view-urile și controller-ele) sunt mult mai predispuse spre schimbare decât informațiile din model
  - Pot fi adăugate vederi noi, fără a modifica în alt fel sistemul
- Utilitate
  - Sisteme interactive, mai ales cele care necesită diferite vederi ale aceluiași model
- MVC este un tip particular de Repository, în care modelul corespunde structurii repository, iar fluxul de control este dictat de către obiectele controller

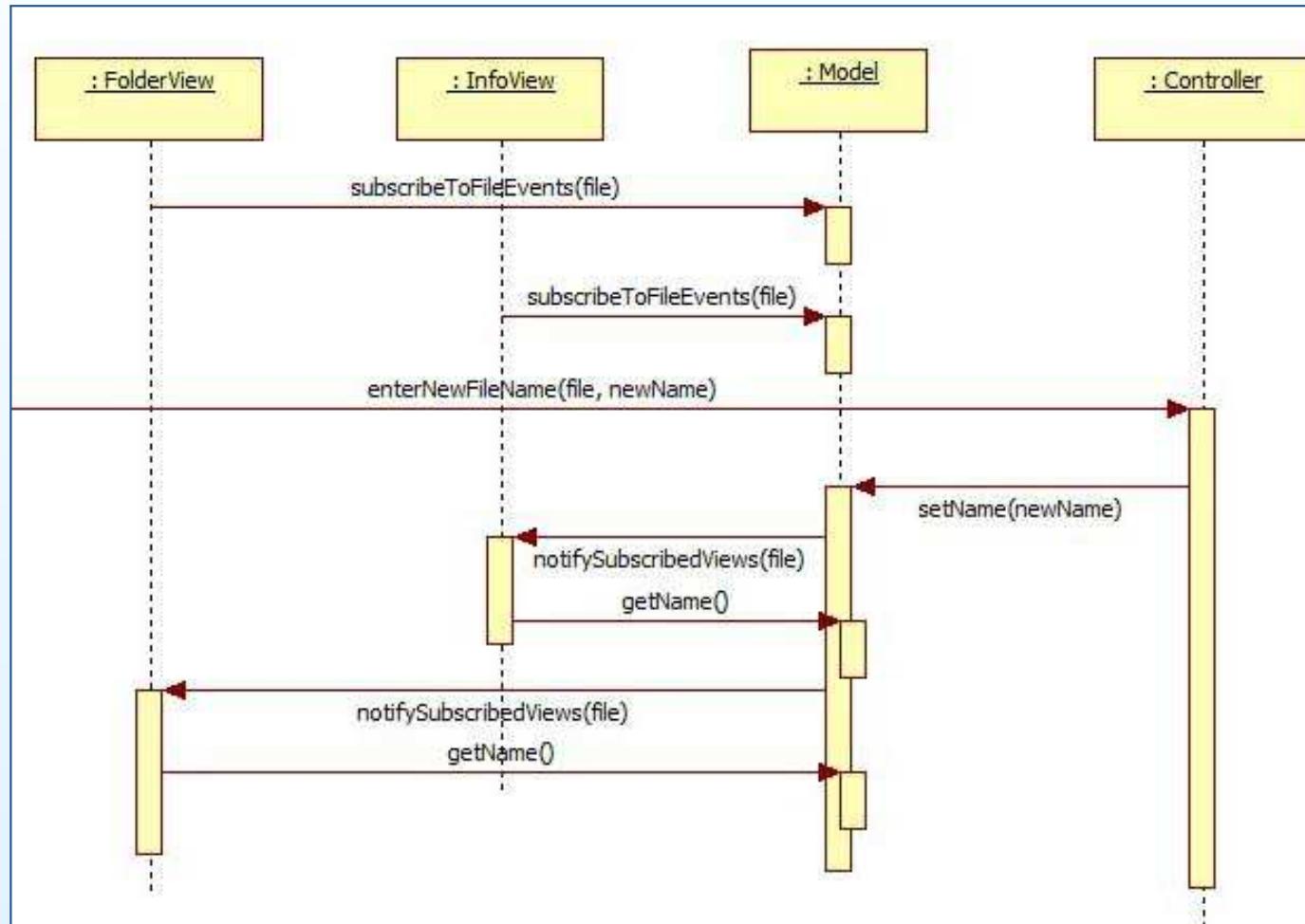
# Exemplu de arhitectură MVC

- Modelul este reprezentat de fișierul *9DesignPatterns.ppt2*
- Una dintre vederi este fereastra CBSE, care listează conținutul directorului cu același nume, cealalta este fereastra Info, care afișează informații relativ la fișierul *9DesignPatterns.ppt2*
- Schimbarea numelui fișierului determină actualizarea ambelor vederi



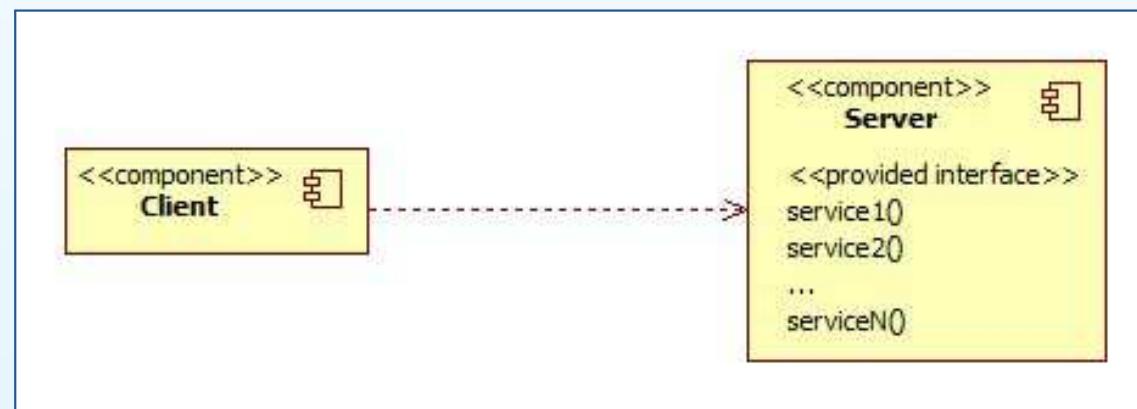
## Exemplu de arhitectură MVC (cont.)

- Secvența de interacțiuni aferentă schimbării numelui fișierului  
*9DesignPatterns.ppt2*



# Client-Server

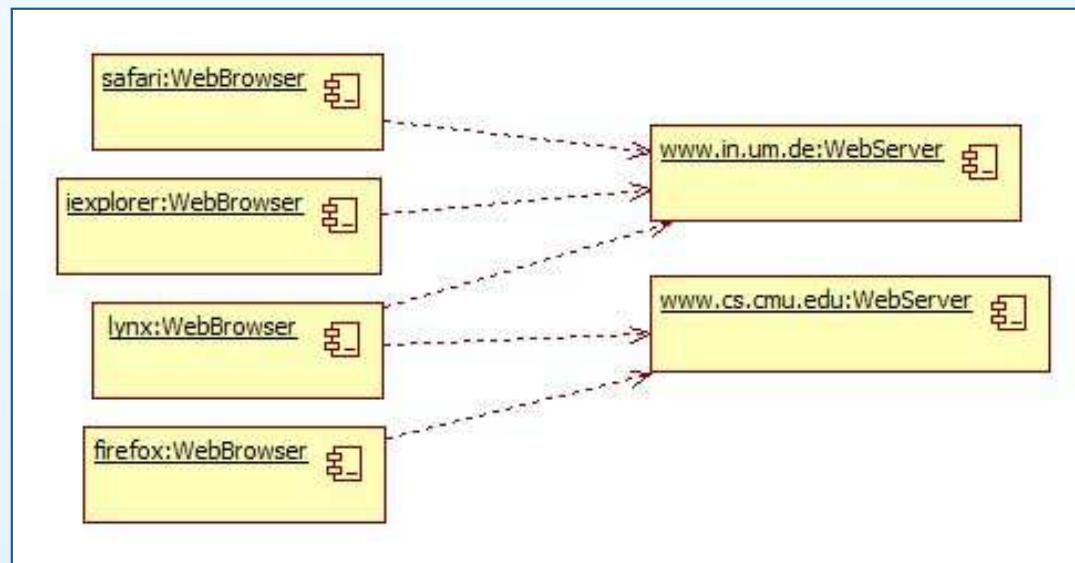
- Un subsistem, numit *server*, oferă servicii instanțelor unor alte subsisteme, numite *clienți*, care sunt responsabile de interacțiunea cu utilizatorii
  - Solicitarea unui serviciu se face, de obicei, printr-un mecanism de apel la distanță (Java RMI, CORBA, HTTP)
  - Fluxurile de control din server și clienți sunt independente, cu excepția sincronizărilor pentru gestionarea cererilor și primirea răspunsurilor



- Utilitate
  - Sisteme distribuite complexe, care gestionează un volum mare de date

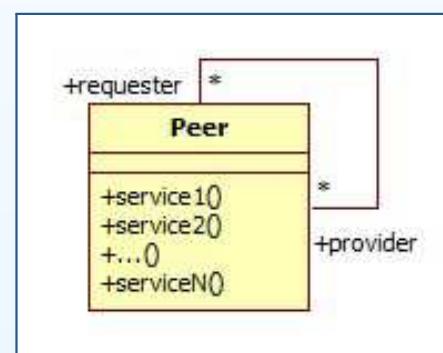
# Exemple de arhitecturi client-server

- Un sistem informațional cu o bază de date centralizată
  - Clienții sunt responsabili de colectarea inputurilor utilizator, validarea acestora și inițierea tranzacțiilor cu baza de date
  - Serverul este responsabil de executarea tranzacțiilor și garantarea integrității datelor
  - În acest caz, stilul client/server este o particularizare a stilului repository, în care structura de date centralizată este gestionată de un proces
- WWW - un client accesează date oferite de diverse servere



# Peer-to-peer

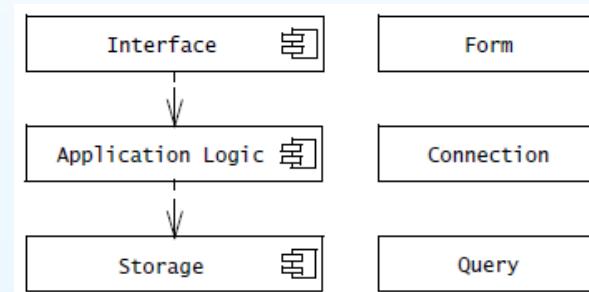
- Generalizare a stilului arhitectural client-server: un subsistem poate juca atât rol de client, cât și de server ( fiecare subsistem poate solicita și oferi servicii)
  - Fluxurile de control sunt independente, cu excepția sincronizărilor pe cereri



- Exemple
  - O bază de date care acceptă cereri de la o aplicație, dar o și notifică atunci când se produc schimbări asupra datelor

# Three-tier architecture

- Subsistemele sunt organizate pe trei straturi/nivele
  - *interfață utilizator* - include toate obiectele *boundary* care mediază interacțiunea cu utilizatorii (ferestre, forme, pagini Web, etc.)
  - *logica aplicației* - include toate obiectele *entity* și *control* care realizează verificările, procesările și notificările cerute de aplicație
  - *accesul la date* - gestionează și oferă acces la datele cu caracter persistent



- Avantaje
  - Nivelul de acces la date joacă rolul repository-ului din stilul arhitectural omonim și poate fi partajat de către aplicațiile care operează asupra respectivelor date
  - Separarea dintre logica aplicației și interfață permite modificări ale interfeței fără a afecta logica aplicației

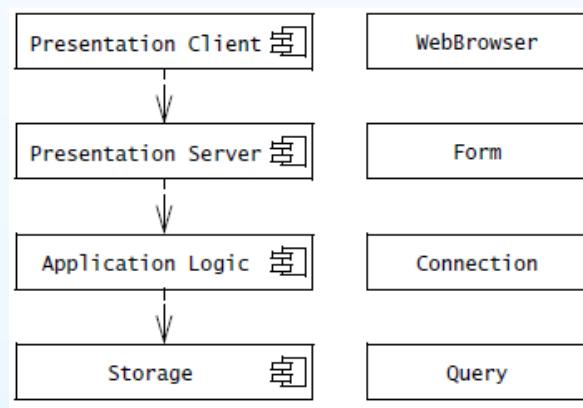
# MVC vs. Three-tier architecture

---

- Stilul arhitectural MVC este nonierarhic (triangular)
  - Subsistemul *view* trimite cereri către subsistemul *controller*
  - Subsistemul *controller* actualizează subsistemul *model*
  - Subsistemul *view* este notificat de către subsistemul *model*
- Stilul arhitectural 3-tier este ierarhic (liniar)
  - Nivelul de prezentare nu comunică niciodată direct cu cel de date (arhitectură închisă)
- MVC nu acoperă problema persistenței datelor

# Four-tier architecture

- O variație a stilului arhitectural three-tier, în care nivelul *interfață* se descompune în
  - *prezentare client* - localizat pe mașinile client
  - *prezentare server* - localizat pe unul sau mai multe servere



- Avantaje
  - Pe nivelul prezentare client pot exista diferiți clienți, o parte a obiectelor *boundary* fiind reutilizate
  - Ex.: un sistem bancar include pe nivelul de prezentare client o interfață Web pentru utilizatori, un ATM și o interfață desktop pentru angajații băncii. Formele partajate de toți clienții sunt definite și procesate la nivelul prezentare server, eliminând redundanța între clienți

# Pipes and filters

- *Pipeline* - lanț de unități de procesare (procese, thread-uri, ...) aranjate astfel încât output-ul uneia reprezintă input-ul următoarei
- *Pipes and filters* - stil arhitectural constând din două tipuri de subsisteme, denumite *pipes* (canale) și *filters* (filtre)
  - *Filter* - subsistem care efectuează un pas de procesare
  - *Pipe* - conexiune dintre doi pași de procesare
- Fiecare subsistem filtru are un canal de intrare și unul de ieșire
  - Datele preluate din canalul de intrare sunt procesate de către filtru și trimise canalului de ieșire
- Ex. Unix shell: `ls -a | cat`

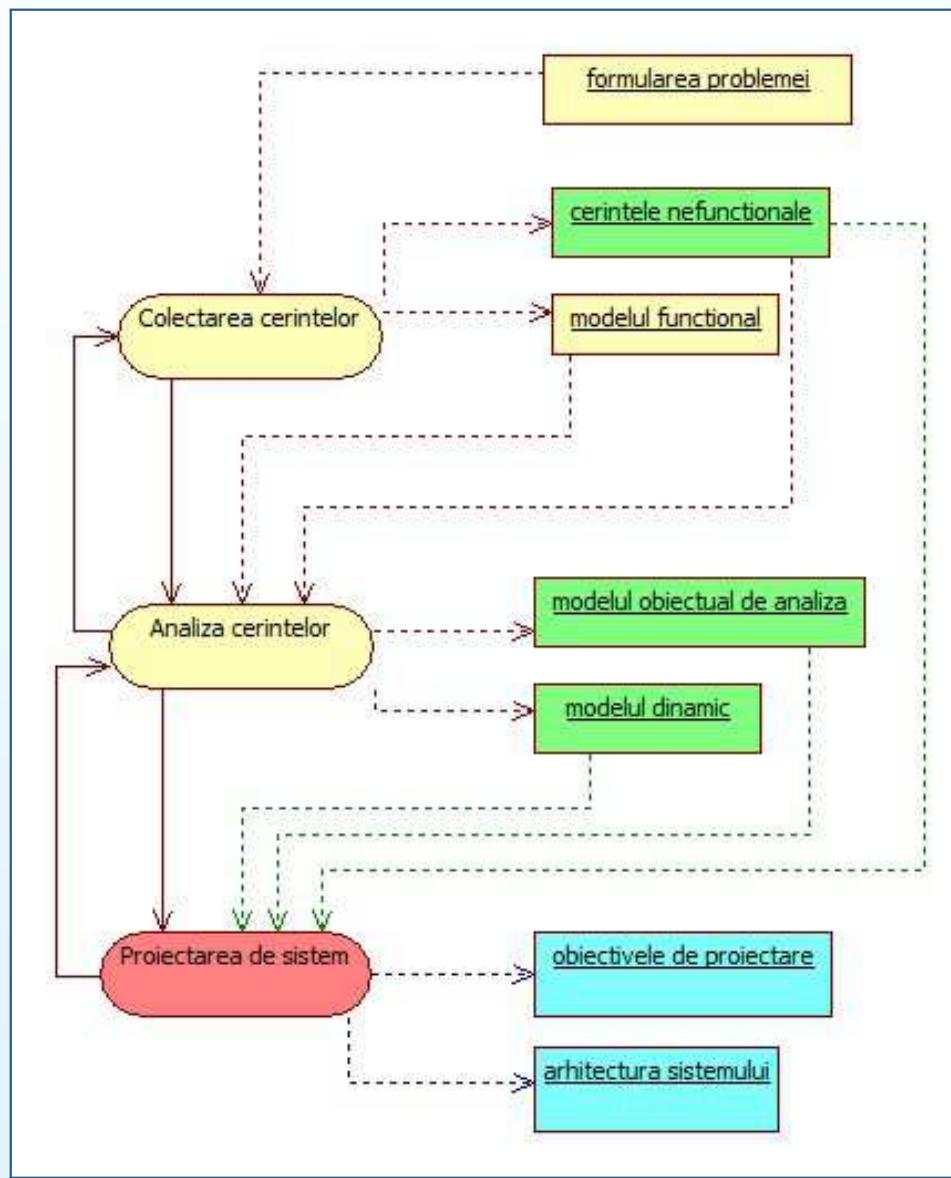
## Curs 6

### *Proiectarea de sistem (II)*

*Suport de curs bazat pe B. Bruegge and A.H. Dutoit*

*"Object-Oriented Software Engineering using UML, Patterns, and Java"*

# Proiectarea de sistem



## Proiectarea de sistem (cont.)

- Procesul de transformare a modelului rezultat din ingineria cerințelor într-un model arhitectural al sistemului
- Produse ale proiectării de sistem
  - *Obiectivele de proiectare* (eng. *design goals*)
    - Calități ale sistemului pe care dezvoltatorii trebuie să le optimizeze
    - Derivate din cerințele nefuncționale
  - *Arhitectura sistemului*
    - Subsistemele componente (de dimensiuni mai mici, asignabile unei subechipe de dezvoltare)
    - Responsabilitățile subsistemelor și dependențele între ele
    - Maparea subsistemelor la hardware
    - Strategii de dezvoltare: strategia de gestionare a datelor cu caracter persistent, politicile de control a accesului, fluxul global de control

## Proiectarea de sistem (cont.)

- Activități ale proiectării de sistem
  - *Identificarea obiectivelor de proiectare*
    - Identificarea și stabilirea priorităților acelor calități ale sistemului pe care dezvoltatorii trebuie să le optimizeze
  - *Descompunerea inițială a sistemului*
    - Pe baza modelului funcțional și a modelelor de analiză
    - Bazată pe utilizarea unor stiluri arhitecturale standard
  - *Rafinarea descompunerii inițiale pentru a răspunde obiectivelor de proiectare*
    - Rafinarea arhitecturii de la pasul anterior până la îndeplinirea tuturor obiectivelor de proiectare
- Analogie cu proiectarea arhitecturală a unei clădiri
  - Componete: camere vs. subsisteme
  - Interfețe: pereți/uși vs. servicii
  - Reproiectare: mutarea pereților vs. schimbarea subsistemelor/interfețelor

## Proiectarea de sistem (cont.)

- Subactivități în rafinarea descompunerii inițiale
  - *Maparea hardware-software*
    - Aspecte urmărite: Care este configurația hardware a sistemului? Cum sunt distribuite funcționalitățile pe noduri? Cum se realizează comunicarea dintre noduri? Ce servicii sunt realizate folosind componente existente? Cum sunt aceste componente încapsulate?
    - Această subactivitate conduce adesea la definirea unor subsisteme adiționale, dedicate transferului de date de la un nod la altul sau gestionării problemelor legate de concurență și fiabilitate
  - *Gestiunea datelor cu caracter persistent*
    - Aspecte urmărite: Care sunt datele cu caracter persistent? Unde ar trebui stocate aceste date? Cum vor fi ele accesate?
    - Această subactivitate conduce adesea la selectarea unui sistem de gestiune a bazelor de date și la identificarea unor subsisteme adiționale dedicate gestiunii datelor cu caracter persistent

## Proiectarea de sistem (cont.)

---

- *Definirea politicilor privind controlul accesului*
  - Aspecte urmărite: Cine are acces la date? La care dintre ele? Se pot schimba dinamic drepturile de acces? Cum este specificat și realizat controlul accesului?
- *Stabilirea fluxului global de control*
  - Aspecte urmărite: Cum sunt secvențiate operațiile în sistem? Este sistemul unul dirijat de evenimente? Poate el gestiona mai mulți utilizatori simultan?
- *Descrierea cazurilor limită (eng. boundary conditions)*
  - Aspecte urmărite: Cum este inițializat și oprit sistemul? Cum sunt gestionate cazurile excepționale?

## Exemplu: sistemul *MyTrip*

---

- *MyTrip* este un sistem care permite planificarea călătoriilor (traseelor aferente) de către șoferi
- Folosind *MyTrip*, un utilizator își poate planifica traseul de efectuat prin intermediul unui calculator personal, prin conectarea la un serviciu Web de planificare (cazul de utilizare *Planificare Traseu*). Traseul este salvat pe server, în vederea consultării ulterioare. Serviciul de planificare trebuie să suporte diferiți utilizatori
- Ulterior, utilizatorul/șoferul efectuează călătoria cu mașina personală, în timp ce calculatorul de bord îi oferă instrucțiuni, pe baza informațiilor legate de traseu salvate pe server și a poziției curente indicate de sistemul GPS încorporat (cazul de utilizare *Efectuare Traseu*)

# *MyTrip: cazul de utilizare Planificare Traseu*

Nume	<i>Planificare Traseu</i>
<b>Flux de evenimente</b>	<ol style="list-style-type: none"><li>Şoferul îşi deschide calculatorul personal şi se loghează în serviciul Web de planificare.</li><li>Şoferul introduce o listă de destinaţii la care doreşte să ajungă.</li><li>Folosind o bază de date cu hărţi, serviciul de planificare calculează cea mai scurtă variantă de a vizita destinaţiile, în ordinea specificată. Rezultatul este dat sub forma unei serii de segmente de drum, ce unesc o serie de intersecţii şi a unei liste de instrucţiuni.</li><li>Şoferul poate revizui planul, prin inserarea sau ştergerea unor destinaţii.</li><li>Şoferul salvează traseul sub un nume ales în baza de date a serviciului de planificare, în vederea utilizării ulterioare.</li></ol>

# *MyTrip: cazul de utilizare Efectuare Traseu*

Nume	<i>Efectuare Traseu</i>
<b>Flux de evenimente</b>	<ol style="list-style-type: none"><li>1. Șoferul își pornește mașina personală și se loghează în sistemul de asistență rutieră al calculatorului de bord.</li><li>2. După logare, șoferul specifică serviciul Web de planificare și numele traseului pe care dorește să îl execute.</li><li>3. Sistemul de asistență obține lista destinațiilor, instrucțiunilor, segmentelor de drum și intersecțiilor aferente de la serviciul de planificare.</li><li>4. Pe baza poziției GPS curente, sistemul de asistență oferă șoferului următorul set de instrucții.</li><li>5. Șoferul ajunge la destinație și oprește sistemul de asistență.</li></ol>

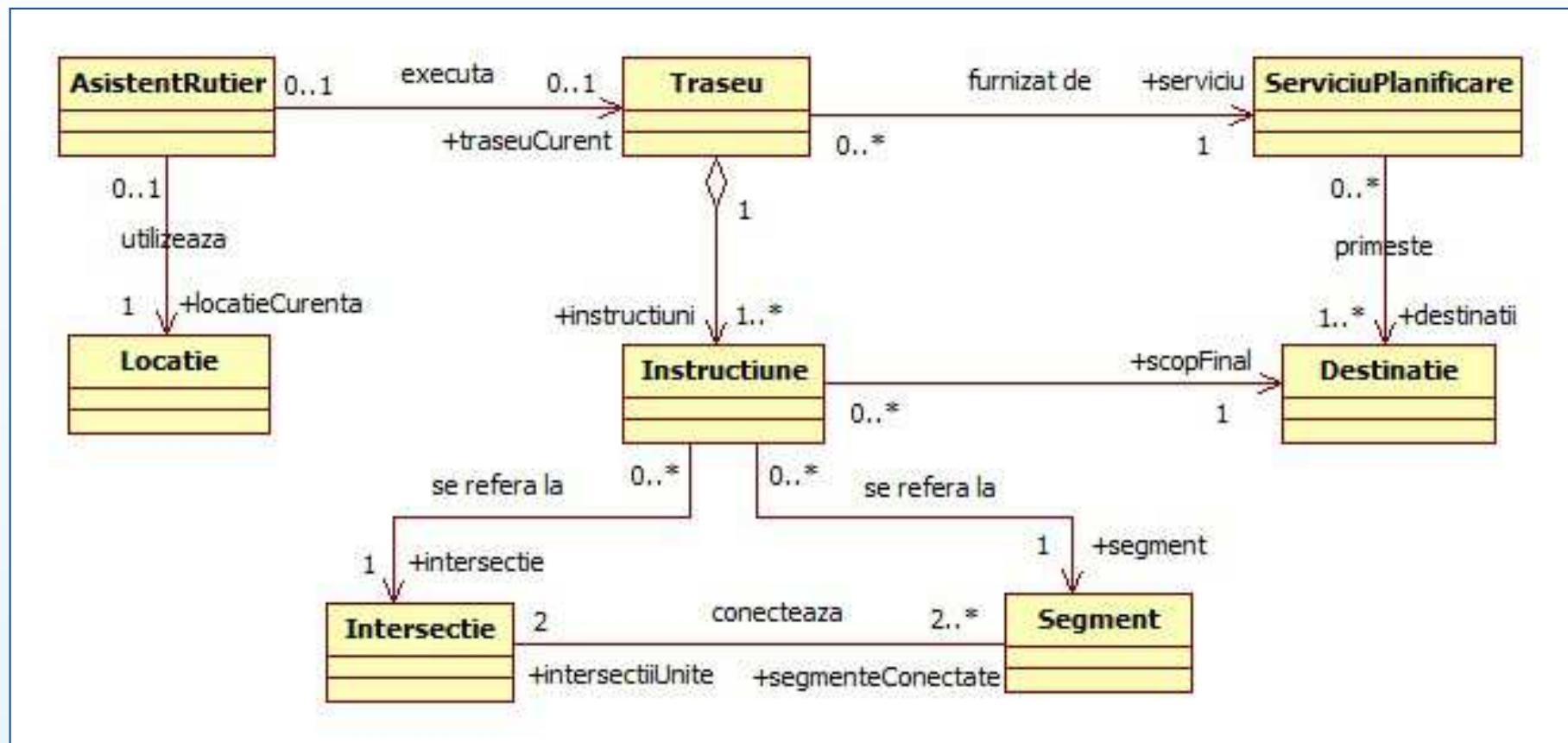
## *MyTrip*: cerințe nefuncționale

1. Conexiunea cu serviciul de planificare este realizată prin intermediul unui modem wireless. Se presupune că acesta funcționează corect la locația inițială.
2. Odată începută călătoria, sistemul *MyTrip* trebuie să furnizeze instrucțiuni corecte, chiar și în condițiile pierderii conexiunii cu serviciul web de planificare.
3. Sistemul *MyTrip* trebuie să minimizeze timpul de conectare, pentru a reduce costurile de operare.
4. Replanificarea este posibilă doar în condițiile în care conectarea la serviciul de planificare este posibilă.
5. Serviciul de planificare trebuie să suporte cel puțin 50 de șoferi diferiți și cel puțin 1000 de trasee diferite.

# MyTrip: concepte din domeniul problemei

<i>Destinație</i>	O <i>Destinație</i> reprezintă un loc în care șoferul dorește să ajungă
<i>Intersecție</i>	O <i>Intersecție</i> reprezintă un punct geografic în care se întâlnesc mai multe <i>Segmente</i>
<i>Segment</i>	Un <i>Segment</i> reprezintă calea dintre două <i>Intersecții</i>
<i>Instructiune</i>	Dată fiind o <i>Intersecție</i> și un <i>Segment</i> adjacent, o <i>Instructiune</i> descrie, în limbaj natural, modalitatea de a dirija mașina pe respectivul <i>Segment</i>
<i>Traseu</i>	Un <i>Traseu</i> constă dintr-o succesiune de <i>Instructiuni</i> între două <i>Destinații</i>
<i>ServiciuPlanificare</i>	Un <i>ServiciuPlanificare</i> este un serviciu Web care poate construi un <i>Traseu</i> pe baza unui șir de destinații, sub forma unei secvențe de <i>Intersecții</i> și <i>Segmente</i> asociate
<i>AsistentRutier</i>	Un <i>AsistentRutier</i> oferă <i>Instructiuni</i> șoferului, pe baza <i>Locației</i> curente și a următoarei <i>Intersecții</i>
<i>Locație</i>	O <i>Locație</i> reprezintă o poziție a mașinii indicată de sistemul GPS încorporat

# MyTrip: modelul conceptual



# Identificarea obiectivelor de proiectare

- *Obiective de proiectare* = criterii de calitate pe care sistemul trebuie să se focuseze
- Trebuie specificate explicit, pentru ca fiecare decizie luată să să se conformeze aceluiași set uniform de criterii
- Categorii de obiective de proiectare
  1. *performanță*: timp de răspuns, puterea de calcul, memorie utilizată
  2. *dependabilitate*: robustețe, fiabilitate, disponibilitate, toleranță la erori, securitate, siguranță
  3. *cost*: costuri de dezvoltare, instalare, întreținere, administrare
  4. *întreținere*: extensibilitate, modificabilitate, adaptabilitate, portabilitate, lizibilitate, trasabilitatea cerințelor
  5. *criterii utilizator*: utilitate, utilizabilitate
- Categoriile 1,2,5 pot fi deduse din cerințele nefuncționale și domeniul problemei, celelalte sunt dictate de client și furnizor
- Priorități și compromisuri: viteză vs. memorie, deadline vs. funcționalitate/calitate/personal

## *MyTrip: Obiective de proiectare*

---

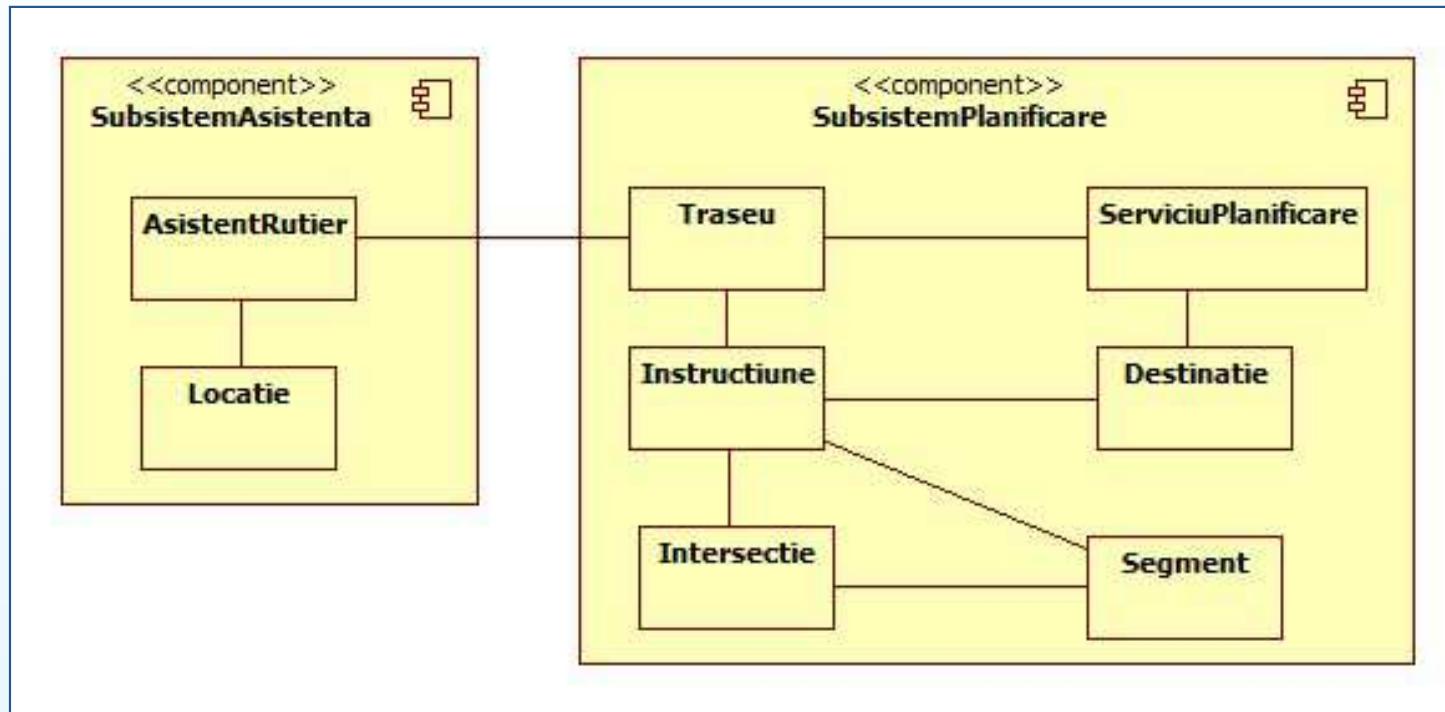
- *Fiabilitate*: Sistemul *MyTrip* trebuie să fie fiabil [generalizare a cerinței nefuncționale 2]
- *Toleranță la erori*: Sistemul *MyTrip* trebuie să funcționeze și în condițiile pierderii conexiunii între sistemul de asistență și serviciul web [reformulare a cerinței funcționale 2]
- *Securitate*: Sistemul *MyTrip* nu trebuie să permită accesul neautorizat la traseele salvate de un șofer [dedusă din domeniul problemei]
- *Modificabilitate*: Sistemul *MyTrip* trebuie să poată fi modificat pentru a utiliza un alt serviciu de planificare [anticipare a schimbării de către dezvoltatori]

# Descompunerea inițială a sistemului

---

- Se realizează pe baza cerințelor funcționale, folosind stiluri arhitecturale predefinite
- Sistemul *MyTrip* - două grupuri de obiecte
  - cele implicate în cazul de utilizare *PlanificareTraseu*
  - cele implicate în cazul de utilizare *ExecutareTraseu*
  - Clasele *Traseu*, *Destinație*, *Instructiune*, *Segment*, *Intersecție* (strâns cuplate, folosite pentru a reprezenta un traseu) sunt partajate între cele două cazuri de utilizare
- Descompunere *MyTrip* - stil arhitectural *Repository*, subsistemul de planificare este responsabil de crearea structurii de date centralizate
  - Subsistemu *SubsistemPlanificare* = *ServiciuPlanificare* + clasele folosite pentru reprezentarea traseului
  - Subsistemu *SubsistemAsistenta* = restul claselor
  - O singură dependență între cele două subsisteme

# MyTrip: Subsisteme și responsabilități inițiale



## SubsistemPlanificare

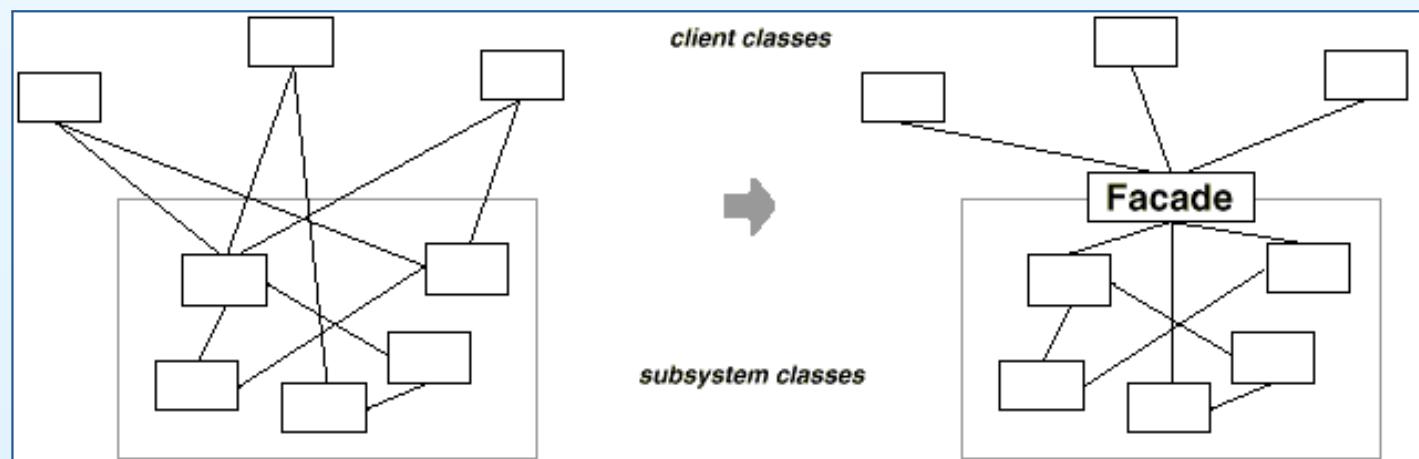
Este responsabil de construirea unui *Traseu* ce unește o serie de *Destinații*. Răspunde la cereri de replanificare din partea *SubsistemuluiAsistenta*.

## SubsistemAsistenta

Este responsabil de descărcarea unui *Traseu* din *ServiciulPlanificare* și de executarea acestuia, prin furnizarea de *Instrucțiuni* șoferului, pe baza *Locației* curente.

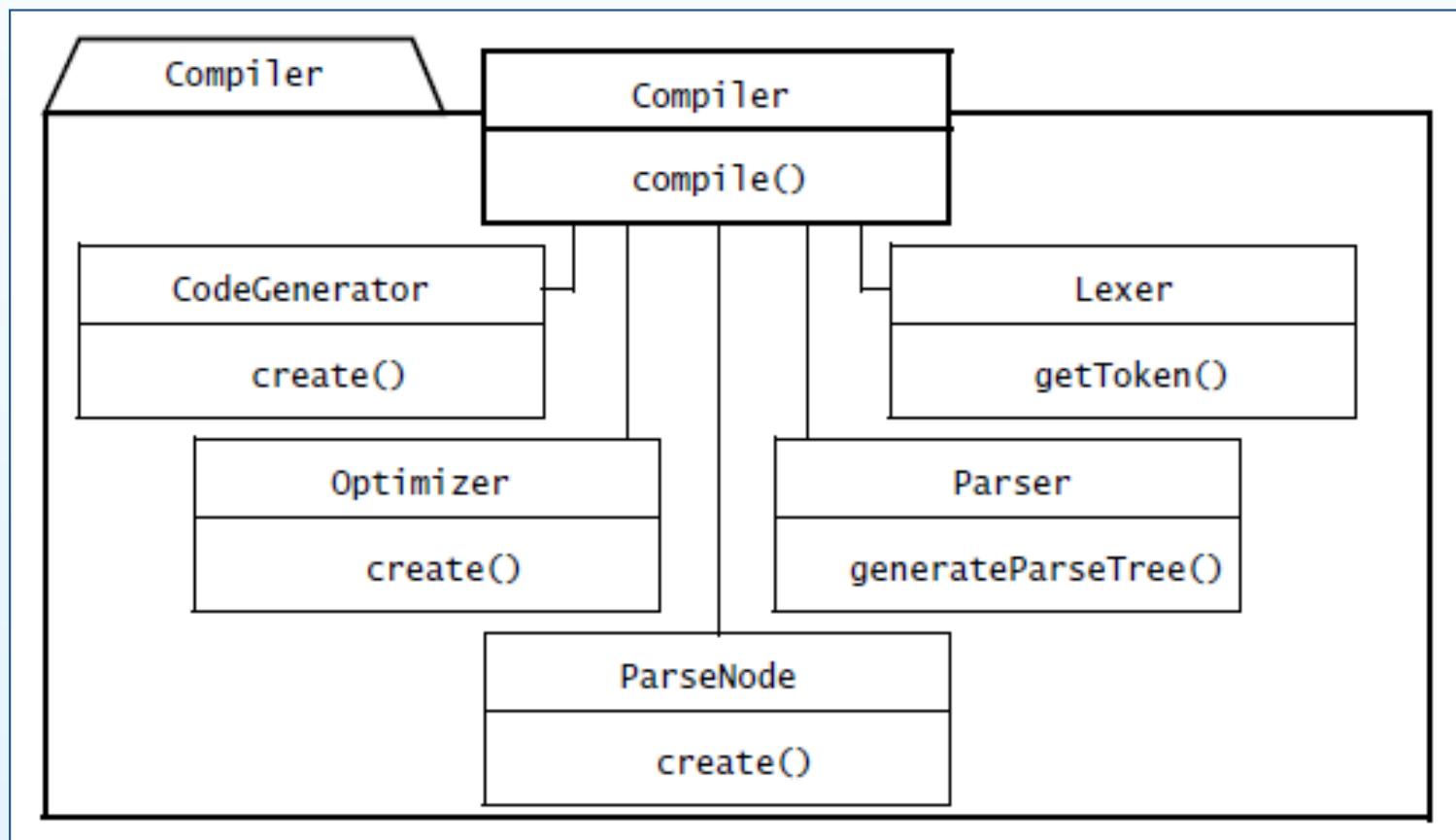
## Descompunerea inițială a sistemului (cont.)

- Euristică pentru descompunerea inițială a sistemului în subsisteme
  - Asignează clasele identificate la nivelul unui caz de utilizare aceluiași subsistem
  - Creează un subsistem dedicat pentru clasele utilizate pentru transportul datelor între subsisteme, sau asignează acele clase subsistemelor responsabile de crearea lor
  - Toate clasele dintr-un subsistem trebuie să fie înrudite funcțional
  - Minimizează numărul de asocieri ce depășesc granițele subsistemelor
- Şablonul de proiectare *Facade* (structural) [Gamma et al., 1994]
  - Permite reducerea dependențelor (cuplării) dintre un subsistem și clienții săi



## Exemplu de aplicare a şablonului *Facade*

- Clasa *Compiler* reprezintă o fațadă ce ascunde clasele *Lexer*, *Parser*, *ParseNode*, *Optimizer*, *CodeGenerator*



# Şablonul *Facade*

---

- *Definiție*
  - Oferă o interfață unificată, de nivel înalt, pentru un grup de interfețe ale unui subsistem, care facilitează utilizarea acestuia
- *Aplicabilitate*
  - Oferirea unei interfețe simple către un subsistem complex
  - Diminuarea numărului de dependențe între clienti și clasele de implementare ale unei abstractizări
  - Stratificarea unui subsistem
- *Participanți*
  - *Facade (Compiler)*
    - Știe ce clase ale subsistemului sunt responsabile de o cerere
    - Delegă cererile clientului către obiectele corespunzătoare din subsistem
  - Clasele subsistemului (*Lexer, Parser, ...*)
    - Implementează funcționalitatea subsistemului
    - Rezolvă sarcinile atribuite de obiectul fațadă
    - Nu știu de existența fațadei (nu păstrează referințe spre ea)

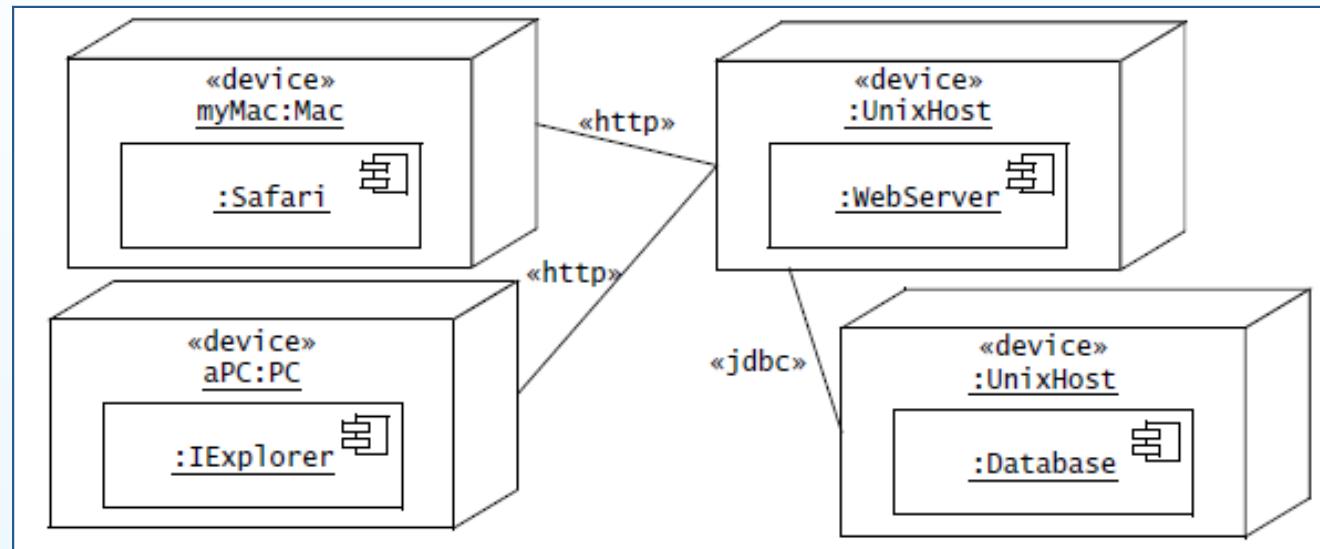
## Şablonul *Facade* (cont.)

- *Colaborări*
  - Clienții comunică cu subsistemul trimițând cereri către obiectul fațadă, care le transmite către obiectele corespunzătoare din subsistem. Deși obiectele din subsistem efectuează munca reală, fațada ar putea fi obligată să efectueze sarcini proprii, pentru a translata interfața sa în interfețele subsistemului
  - Clienții care utilizează fațada nu trebuie să acceseze direct obiectele din subsistemul acesteia
- Şablonul nu interzice clienților să utilizeze direct clasele subsistemului, dacă acest lucru este necesar

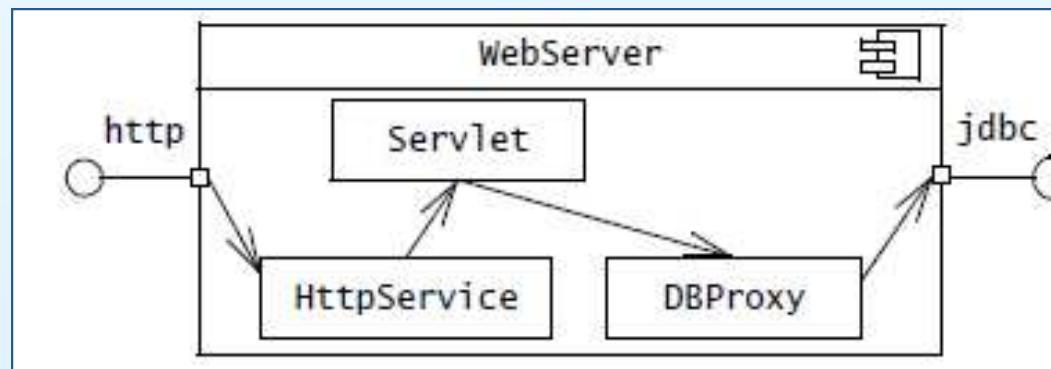
# Diagrame de repartiție a resurselor

- La runtime, un sistem este reprezentat de o mulțime de componente care interacționează, distribuite la nivelul unor noduri
- O *diagramă de repartiție UML* (eng. *UML deployment diagram*) ilustrează relația dintre componentele runtime și noduri
  - Componentă = unitate autoconținută, care oferă servicii altor componente sau actorilor
    - Un server web este o componentă care oferă servicii browserelor web
    - Un browser web este o componentă care oferă servicii utilizatorilor
  - Nod = dispozitiv fizic (calculator) sau mediu de execuție pentru componente
    - Un nod poate conține un alt nod (un dispozitiv conține un mediu de execuție)
- Reprezentare
  - Nodurile sunt reprezentate ca și paralelipipede dreptunghice, ce conțin componente
  - Nodurile pot fi stereotipizate pentru a indica un dispozitiv fizic sau un mediu de execuție
  - Comunicarea dintre noduri se reprezintă cu o linie continuă, ce poate fi stereotipizată cu un nume de protocol

# Exemplu de diagramă UML de repartiție

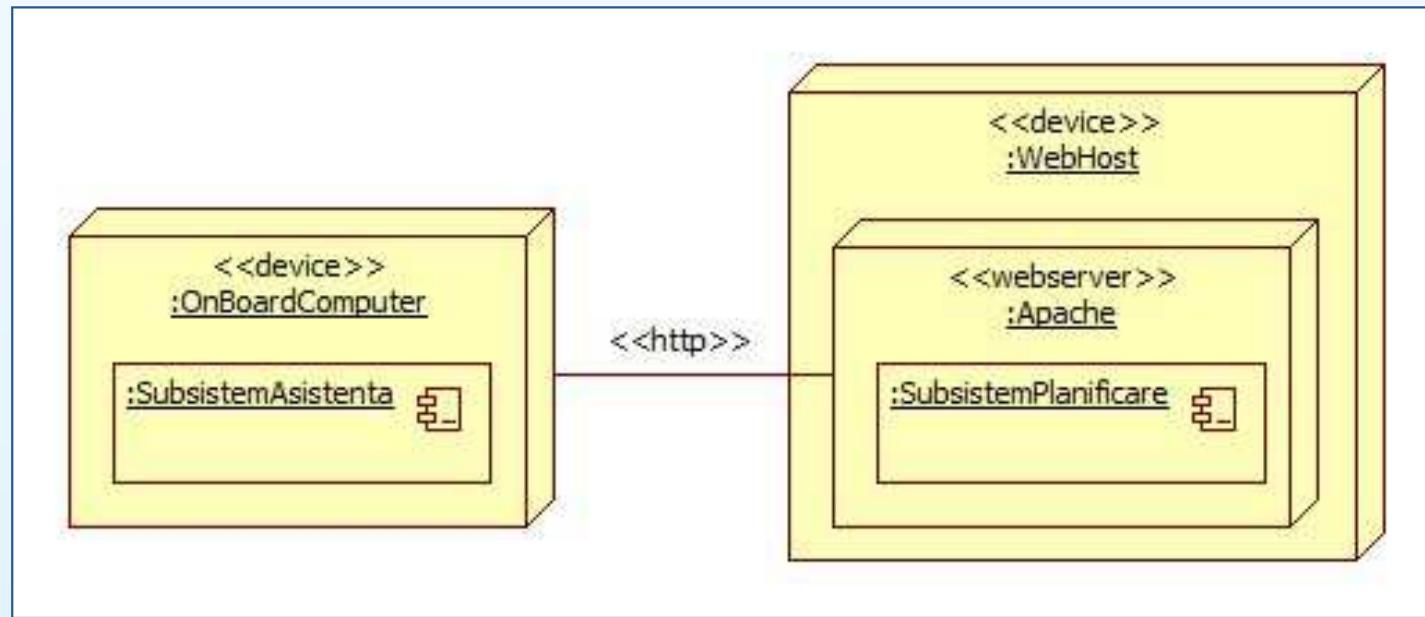


- Componentele pot fi rafinate, pentru a include informații relativ la clasele pe care le conțin și la interfețele pe care le oferă/solicită



# Maparea subsistemelor la hardware

- Strategia de alocare a subsistemelor pe echipamentele hardware și proiectarea infrastructurii de comunicare între aceste subsisteme influențează semnificativ performanțele sistemului și complexitatea acestuia
  - Selectarea configurației hardware include și selectarea mașinii virtuale pe care va fi construit sistemul
- Alocarea subsistemelor *MyTrip*

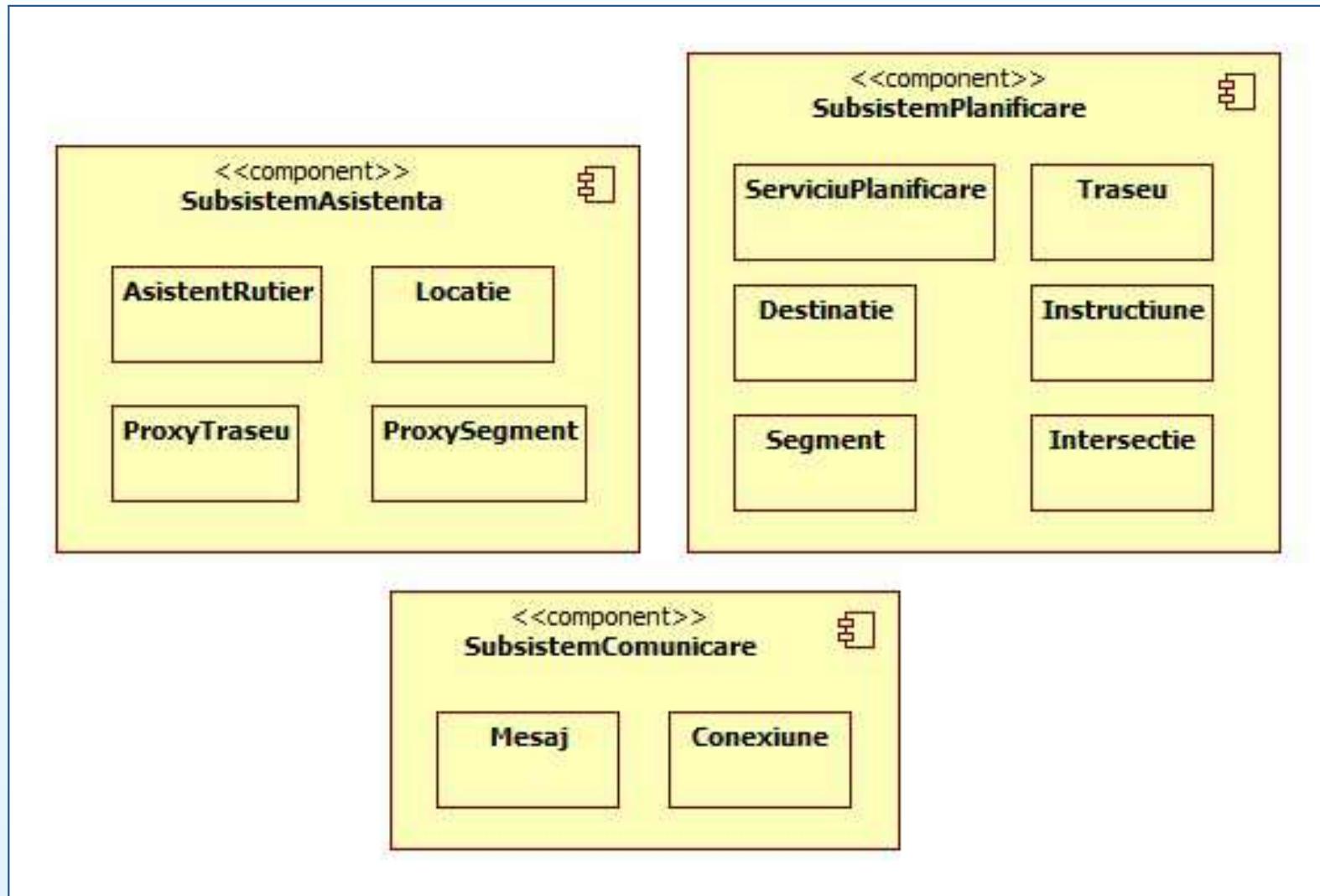


## Maparea subsistemelor la hardware (cont.)

---

- Alocarea subsistemelor *MyTrip*
  - Subsistemul *SubsistemPlanificare* rulează pe un server web (*WebHost*), în mediul de execuție Apache. Mașina virtuală e o mașină Unix
  - Subsistemul *SubsistemAsistență* rulează pe computerul de bord al mașinii (*OnBoardComputer*), mașina virtuală fiind un browser web
- Alocarea subsistemelor pe noduri, implică, de obicei, identificarea unor noi clase/subsisteme utilizate pentru transportul datelor între aceste noduri
  - În cazul *MyTrip*, obiectele aferente unui traseu (*Traseu*, *Instructiune*, *Intersecție*, *Segment*, *Destinație*) trebuie transportate de la subsistemul de planificare spre cel de asistență
  - Pentru a gestiona comunicarea între aceste două subsisteme, se introduce un subsistem nou *SubsistemComunicare*, ce va fi localizat pe ambele noduri

# MyTrip: prima rafinare a descompunerii inițiale



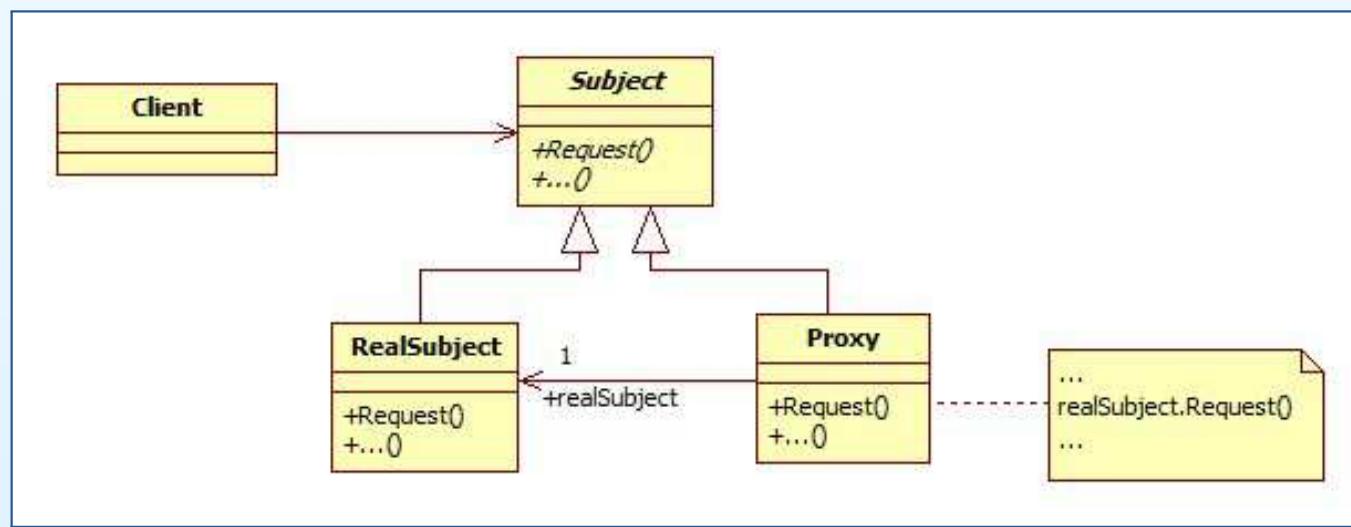
## MyTrip: prima rafinare a descompunerii inițiale (cont)

---

<i>SubsistemComunicare</i>	Responsabil de transportarea obiectelor de la <i>SubsistemuPlanificare</i> la <i>SubsistemuAsistenta</i>
<i>Conexiune</i>	O <i>Conexiune</i> reprezintă o legătură activă între subsistemul de planificare și cel de asistență. Un obiect <i>Conexiune</i> tratează cazurile excepționale cauzate de pierderea conexiunii în rețea
<i>Mesaj</i>	Un <i>Mesaj</i> reprezintă un <i>Traseu</i> și elementele asociate ( <i>Instructiune</i> , <i>Destinatie</i> , <i>Segment</i> , <i>intersecție</i> ), codificate pentru transport

# Şablonul *Proxy* (structural) [Gamma et al., 1994]

- *Definiție*
  - Şablonul asigură, pentru un obiect, un surogat sau un înlocuitor, în scopul controlării accesului la acesta
- *Aplicabilitate*
  - *Proxy la distanță* - oferă un reprezentant local pentru un obiect dintr-un spațiu de adresă diferit
  - *Proxy virtual* - creează la cerere obiecte costisitoare
  - ...
- *Structură*



# Şablonul *Proxy* (cont.)

- *Participanți*

- *Proxy*

- Păstrează o referință care îi permite să acceseze subiectul real
    - Asigură o interfață identică celei a clasei *Subject*, astfel încât un obiect proxy să poată înlocui un obiect real
    - Controlează accesul la subiectul real și poate răspunde de crearea și stergerea acestuia
    - Obiectele *proxy la distanță* răspund de codificarea unei cereri și a argumentelor acesteia și de transmiterea cererii codificate către subiectul real, aflat într-un spațiu de adresă diferit
    - Obiectele *proxy virtuale* pot depozita informație suplimentară despre subiectul real, astfel încât să poată amâna accesarea acestuia (ex.: un obiect *ImageProxy* poate stoca dimensiunile unui obiect *RealImage*)

- *Subject*

- Definește interfața comună pentru obiectele *RealSubject* și *Proxy*, astfel încât un obiect *Proxy* să poată fi utilizat în orice loc în care este așteptat un obiect *RealSubject*

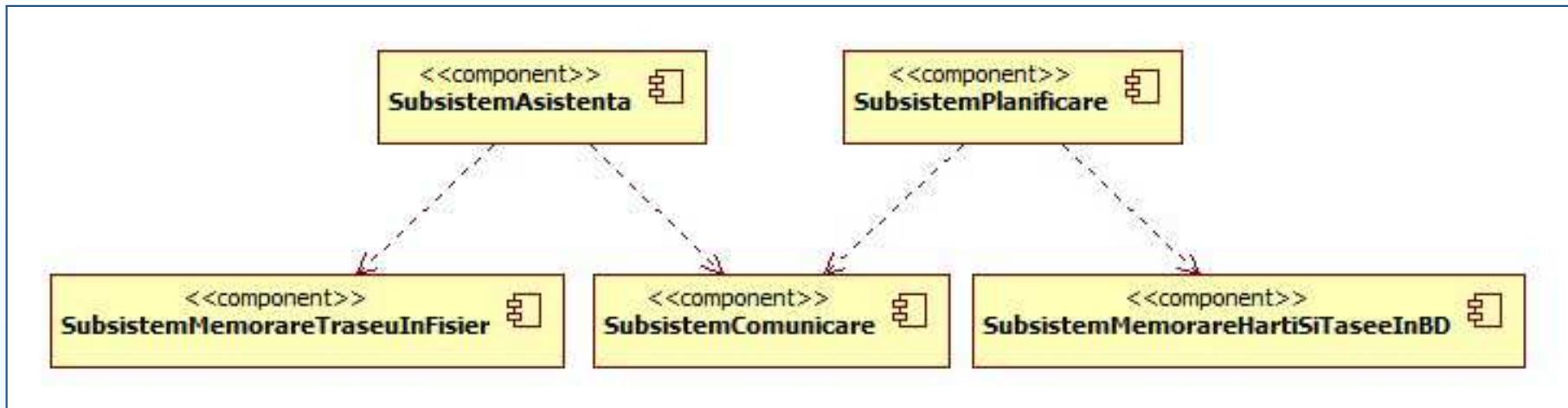
## Şablonul *Proxy* (cont.)

- *RealSubject*
  - Definește obiectul real reprezentat de obiectul proxy
- *Colaborări*
  - Când se impune, obiectul proxy transmite cererile către subiectul real, funcție de tipul de proxy
- *Consecințe*
  - Şablonul *Proxy* introduce un grad de "ocolire" la accesul unui obiect. Semnificația acestei ocoliri depinde de tipul de proxy: un proxy la distanță ascunde faptul că un obiect se află într-un alt spațiu de adresă; un proxy virtual poate efectua optimizări, cum ar fi crearea unui obiect la cerere (un proxy pentru o imagine instanțiază obiectul imagine concret doar atunci când este necesară afișarea imaginii)

# Gestiunea datelor persistente

- Unele dintre obiectele ce compun sistemul trebuie să fie *persistente*
  - Valorile atributelor acestora au o durată de viață ce o depășește pe cea a unei singure execuții a sistemului
- Persistența obiectelor se poate realiza folosind
  - Fișiere - în cazul în care există mai mulți cititori, dar un singur scriitor
  - Baze de date - în cazul în care datele sunt accesate de cititori și scriitori concurenți
- Strategia de persistență pentru sistemul *MyTrip*
  - Traseul curent este salvat într-un fișier, pentru a permite recuperarea acestuia în cazul în care șoferul oprește mașina înainte de a ajunge la destinația finală
  - Toate traseele aferente subsistemului de planificare, precum și hărțile necesare generării acestora sunt memorate la nivelul unei baze de date
  - Strategia de persistență aleasă determină introducerea în sistem a două noi subsisteme

# MyTrip: a doua rafinare a descompunerii inițiale



*SubsistemMemorareTraseuInFisier*

Responsabil de memorarea traseelor în fișiere pe calculatorul de bord. Deoarece această funcționalitate este folosită doar pentru a salva trasee atunci când mașina se oprește, acest subsistem suportă doar memorarea și încărcarea rapidă a unor trasee complete.

*SubsistemMemorareHartiSiTraseeInBD*

Responsabil de memorarea hărților și a traseelor într-o bază de date pentru subsistemul de planificare. Acest subsistem suportă accesul concurrent al mai multor șoferi și agenți de planificare.

# Gestiunea datelor persistente: activități

- *Identificarea obiectelor persistente*
  - Date candidat pentru persistență
    - Entitățile (nu neapărat toate) identificate în etapa de analiză (pentru sistemul *MyTrip*: *Destinație*, *Intersecție*, *Segment*, *Traseu*; *Locație* și *Instrucțiune* se recalculează funcție de poziția curentă a mașinii => nu trebuie persistate)
    - Informații legate de utilizatori, într-un sistem multi-utilizator (*Șoferi*, spre exemplu), atribute ale unor obiecte *boundary* ce rețin preferințe ale utilizatorilor, etc.
    - În general, toate clasele ale căror obiecte trebuie să supraviețuiască unei opriri a sistemului, controlată sau neașteptată
- *Selectarea strategiei de memorare*
  - Decizie complexă, determinată, în general, de cerințe nefuncționale:  
Obiectele trebuie regăsite rapid? Sunt necesare interogări complexe pentru regăsirea acestor obiecte? Obiectele necesită mult spațiu pentru memorare?
  - În general, 3 opțiuni

# Gestiunea datelor persistente: activități (cont.)

---

- *Fișiere*
  - + Avantaje: permit o mare varietate de optimizări de viteză și dimensiune
  - Dezavantaje: lasă pe umerii aplicației gestiunea accesului concurrent sau recuperarea datelor în urma unei căderi a sistemului
- *Baze de date relationale*
  - + Avantaje: tehnologie matură, ce oferă servicii pentru controlul accesului, gestiunea concurenței și recuperarea datelor în urma unei căderi a sistemului
  - Dezavantaje: deși scalabile și ideale pentru volume mari de date, sunt relativ lente pentru seturi de date mici sau date nestructurate (imagini, text în limbaj natural); în plus, există necesitatea mapării obiect-relaționale
- *Baze de date orientate obiect*
  - + Avantaje: un nivel mai înalt de abstractizare: memorează datele ca și obiecte, înălțând necesitatea translației între obiecte și entitățile memorate
  - Dezavantaje: mai lente decât bazele de date relationale

# Euristici de selectare a mecanismului de persistență

---

- Când se folosesc fișiere?
  - Date voluminoase (imagini)
  - Date temporare
  - Densitate mică a informației (log-uri)
- Când se folosesc baze de date?
  - Acces concurent
  - Mai multe platforme sau aplicații care accesează aceleași date
- Când se folosesc baze de date relaționale?
  - Interogări complexe
  - Seturi mari de date
- Când se folosesc baze de date orientate obiect?
  - Utilizare masivă a asocierilor pentru regăsirea datelor
  - Seturi de date medii

# Definirea politicilor privind controlul accesului

---

- În sistemele multi-utilizator, actori diferiți au, de obicei, drepturi de acces diferite la diferite funcționalități și date
- Cum modelăm accesul?
  - În timpul analizei - prin asocierea diferitor cazuri de utilizare la diferiți actori
  - În timpul proiectării - prin determinarea acelor obiecte partajate între actori, precum și a modului în care actorii pot controla accesul. Funcție de cerințele de securitate ale sistemului, definim și modul în care actorii se autentifică în sistem, precum și modalitatea de criptare a anumitor date
- Sistemul *MyTrip*
  - Se introduce o clasă *Şofer*, asociată clasei *Traseu*, pentru a asigura trimiterea traseelor doar la şoferii care le-au creat
  - Subsistemul de planificare devine responsabil de autentificarea şoferilor, înainte de a le trimite traseele solicitate
  - Subsistemul de comunicare criptează traficul dintre subsistemul de asistență și cel de planificare

# MyTrip - o nouă rafinare

*Şofer*

*Un Şofer reprezintă un utilizator autentificat. Este utilizat de către subsistemul de comunicare, pentru a memora cheia utilizată pentru criptare și de către subsistemul de planificare, pentru a asocia Traseele cu utilizatorii.*

*SubsistemPlanificare*

*Este responsabil de construirea unui Traseu ce unește o serie de Destinații. Răspunde la cereri de replanificare din partea SubsistemuAsistenta. Înainte de a procesa o cerere, subsistemul de planificare autentifică Şoferul de la subsistemul de asistență. Şoferul autentificat este utilizat pentru a determina acele Trasee care pot fi trimise subsistemului de asistență.*

*SubsistemComunicare*

*Responsabil de transferul obiectelor de la SubsistemuPlanificare la SubsistemuAsistenta. Acesta utilizează Şoferul asociat Traseului care se transferă pentru a selecta o cheie și a cripta traficul.*

# Matrici de acces

---

- Accesul la clase poate fi modelat folosind *matrici de acces*
  - Liniile reprezintă actorii din sistem
  - Coloanele reprezintă clasele ale căror drepturi de acces dorim să fie modelate
  - O intrare în matrice listează operațiile care pot fi executate pe o instanță a clasei de pe coloană de către actorul de pe linie
- Variații de implementare
  - *Tabel de acces global* - reprezintă explicit fiecare triplet (actor, clasă, operație)
  - *Liste de control a accesului* - asociază o listă de perechi (actor, operație) fiecărei clase. De fiecare dată când o instanță a clasei este accesată, se verifică existența perechii (actor, operație) în listă
  - *Capabilități* - asociază perechi (clasă, operație) unui actor

# Stabilirea fluxului global de control

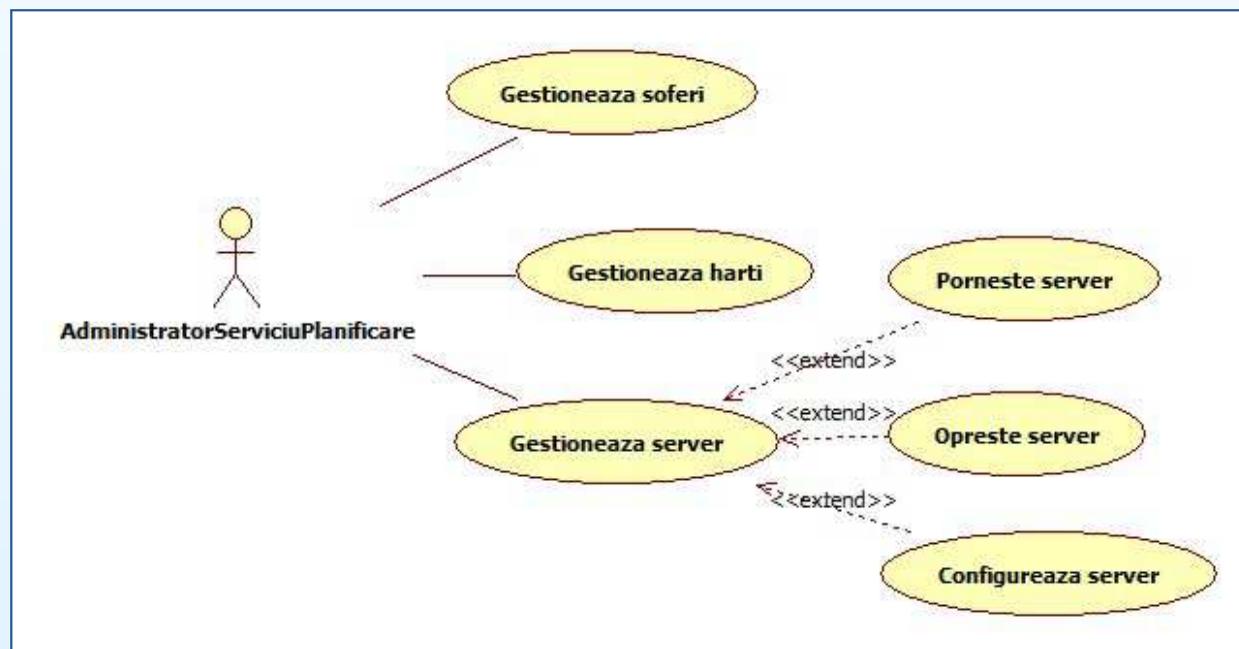
- *Flux de control* = secvențierea acțiunilor într-un sistem (ce operații se execută și în ce ordine)
- Mecanisme
  - *Flux de control procedural*
    - Operațiile așteaptă input ori de câte ori au nevoie de date de la un actor
    - Utilizat mai mult în limbajele procedurale
    - Util pentru testarea subsistemelor
  - *Flux de control dirijat de evenimente*
    - O buclă așteaptă evenimente externe. Ori de câte ori un astfel de eveniment apare, este direcționat către obiectul aferent, pe baza unor informații asociate evenimentului
    - Avantaje: abordare matură, structură simplă, centralizarea input-urilor
  - *Thread-uri*
    - Varianta concurentă a controlului procedural
    - Sistemul poate crea un număr arbitrar de thread-uri, fiecare răspunzând la un eveniment diferit. Dacă un thread are nevoie de date suplimentare, așteaptă input de la un actor
    - Avantaje: mecanism intuitiv; dezavantaje: dificil de testat și depanat

# Descrierea cazurilor limită

- Examinarea condițiilor limită
  - Cum este sistemul pornit, inițializat și opus?
  - Cum sunt gestionate datele corupte sau căderile de rețea?
  - Cazurile de utilizare care gestionează aceste condiții se numesc *cazuri de utilizare limită* (eng. *boundary use cases*)
- Sistemul *MyTrip*
  - Cum sunt încărcate hărțile în serviciul de planificare?
  - Cum este instalat sistemul într-o mașină?
  - De unde cunoaște sistemul *MyTrip* serviciul de planificare la care trebuie să se conecteze?
  - Cum sunt adăugați șoferii în serviciul de planificare?
- Euristică pentru determinarea cazurilor limită
  - *Configurare*: Pentru fiecare obiect persistent, se identifică acele cazuri de utilizare în care obiectul este creat/ distrus. Pentru obiectele care nu sunt create/distruse în cazurile de utilizare existente (ex.: hărțile în sistemul *MyTrip*), se adaugă un caz de utilizare gestionat de un administrator al sistemului (ex.: *GestionareHarti*)

## Descrierea cazurilor limită (cont.)

- *Pornire/oprire*: Pentru fiecare componentă, se adaugă trei cazuri de utilizare pentru a porni, opri și configura componenta
  - *Gestiunea excepțiilor*: Pentru fiecare tip de eșec (ex.: cădere de rețea), se creează un caz de utilizare excepțional, care extinde un caz de utilizare existent (alternativa a utilizării scenariilor de excepție)
- Cazuri de utilizare limită pentru *MyTrip*



## Referințe

---

- [Gamma et al., 1994] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1994.

*Cursuri 7-8*

*Proiectarea obiectuală: Reutilizare*

*Suport de curs bazat pe B. Bruegge and A.H. Dutoit*

*"Object-Oriented Software Engineering using UML, Patterns, and Java"*

# Proiectarea obiectuală

- Activitățile tehnice ale ingineriei software reduc, în manieră graduală, discrepanța problemă - mașină fizică / calculator, prin identificarea / definirea obiectelor care compun viitorul sistem
  - *Analiză*
    - identificarea obiectelor aferente conceptelor din domeniul problemei
  - *Proiectare de sistem*
    - definirea mașinii virtuale, prin selectarea de componente predefinite pentru servicii standard (sisteme de operare, medii de execuție, cadre de aplicație, kituri de instrumente pentru interfața cu utilizatorul, biblioteci de clase de uz general)
    - identificarea unor componente predefinite aferente obiectelor din domeniul problemei (ex. o bibliotecă reutilizabilă de clase reprezentând concepte caracteristice domeniului bancar)
  - *Proiectare obiectuală*
    - identificarea de noi obiecte din domeniul soluției, rafinarea celor existente, adaptarea componentelor reutilizate, specificarea riguroasă a interfețelor subsistemelor și claselor componente

## Proiectarea obiectuală (cont.)

---

- Activități ale proiectării obiectuale

- *Reutilizare*
  - reutilizarea unor componente de bibliotecă pentru structuri de date / servicii de bază
  - reutilizarea şablonelor de proiectare pentru rezolvarea unor probleme recurente și asigurarea modificabilității / extensibilității sistemului
  - adaptarea componentelor reutilizate prin încapsulare / specializare
- *Specificarea interfețelor*
  - specificarea completă a interfețelor și claselor ce compun subsistemele
- *Restructurarea modelului obiectual*
  - transformarea modelului obiectual în scopul creșterii inteligibilității și menținabilității acestuia
- *Optimizarea modelului obiectual*
  - transformarea modelului obiectual în scopul asigurării criteriilor de performanță (în termeni de timp de execuție / memorie utilizată)

## Concepte legate de reutilizare

---

- Obiecte din domeniul problemei vs. obiecte din domeniul soluției
- Moștenirea specificării vs. moștenirea implementării
- Delegare
- Prințipiu Liskov al substituției
- Prințipiile SOLID de proiectare a claselor

# Obiecte din domeniul problemei / soluției

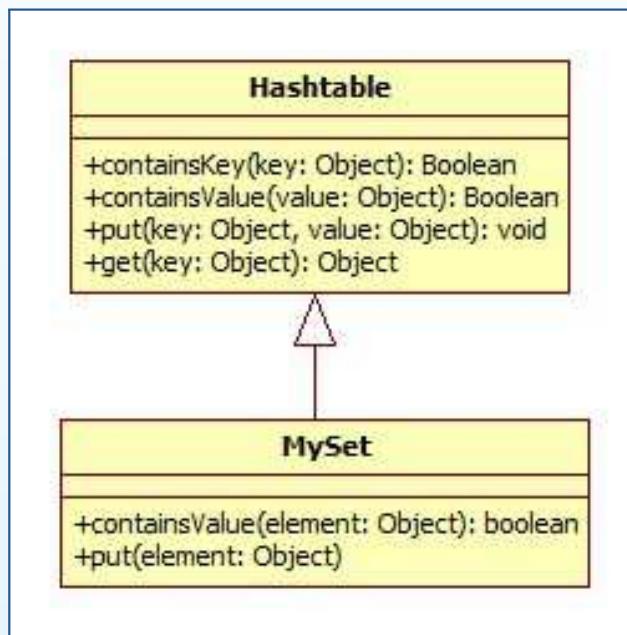
- Diagramele UML de clase pot fi utilizate pentru modelarea atât a domeniului problemei, cât și a domeniului soluției
- Dualitatea *domeniul problemei / domeniul soluției*
  - *Obiectele din domeniul problemei* (eng. *domain objects, application objects*) corespund unor concepte ale domeniului relevante pentru sistemul în cauză
  - *Obiectele din domeniul soluției* (eng. *solution objects*) corespund unor concepte fără corespondent la nivelul domeniului problemei (ex. depozite de date persistente, obiecte ale interfeței grafice utilizator)
- Etapele identificării obiectelor din domeniul problemei / soluției
  - *În analiză* - se identifică obiectele din domeniul problemei + acele obiecte din domeniul soluției ce sunt vizibile utilizatorului (obiecte *boundary* și *control* ce corespund interfețelor și tranzacțiilor definite de sistem)
  - *În proiectarea de sistem* - se identifică obiecte din domeniul soluției în termenii platformei hardware/ software
  - *În proiectarea obiectuală* - se rafinează și detaliază obiectele din domeniul problemei / soluției identificate anterior și se identifică restul obiectelor din domeniul soluției

# Moștenirea specificării vs. moștenirea implementării

- În etapa de analiză, moștenirea este utilizată pentru clasificarea obiectelor în taxonomii
  - Rolul generalizării (identificarea unei superclase comune pentru un număr de clase existente) și al specializării (identificarea unor subclase ale unei clase existente) este acela de organiza obiectele din domeniul problemei într-o ierarhie ușor de înțeles și de a diferenția comportamentul comun unei mulțimi de obiecte (expus de clasa de bază sau superclasă) de comportamentul specific caracteristic claselor derivate (sau subclaselor)
- În etapa de proiectare obiectuală, rolul moștenirii este acela de a elimina redundanțele din modelul obiectual și de a asigura modificabilitatea/extensibilitatea sistemului
  - Factorizarea comportamentului comun la nivelul clasei de bază reduce riscul apariției unor inconsistențe în eventualitatea efectuării unor modificări la nivelul implementării respectivului comportament, prin localizarea modificărilor într-o singură clasă
  - Definirea de clase abstracte / interfețe asigură extensibilitatea, permitând specializarea comportamentului prin definirea de noi subclase, conforme interfețelor abstracte

## Moștenirea specificării / implementării (cont.)

- Ex.: Se dorește implementarea unei clase *MySet*, reutilizând funcționalitatea oferită de clasa existentă *Hashtable*
- *Soluția 1: Clasa MySet e o specializare a lui Hashtable*



```
public class MySet extends Hashtable {

    public MySet() {}

    public void put(Object element)
    {
        if (!containsKey(element))
            put(element, this);
    }

    @Override
    public boolean containsValue(Object element)
    {
        return containsKey(element);

    }

    /* ... */
}
```

## Moștenirea specificării / implementării (cont.)

---

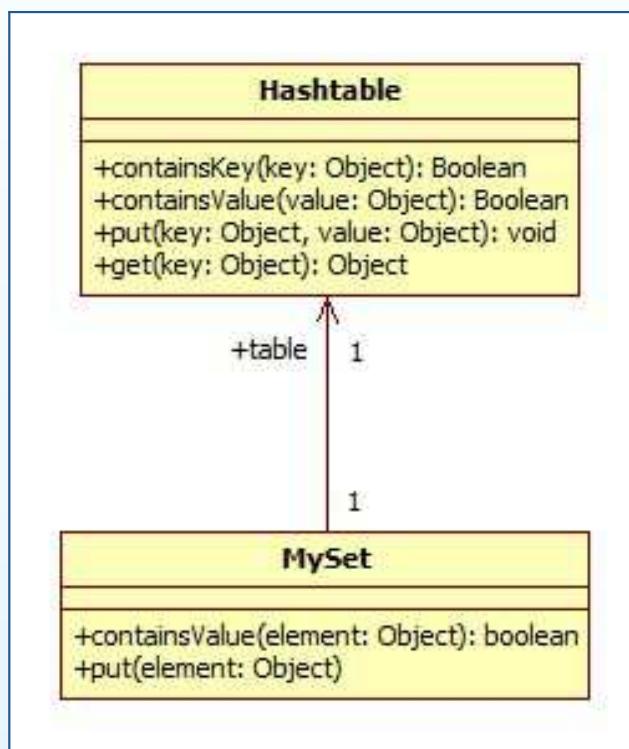
- *Avantaje*
  - Funcționalitatea dorită
  - Reutilizarea codului
- *Probleme*
  - Clasa *MySet* moștenește *containsKey()* și suprascrie *containsValue()*, oferind același comportament - contraintuitiv
  - Clientii *MySet* pot utiliza *containsKey()*, ceea ce face dificilă modificarea reprezentării *MySet*
  - Substituirea unui obiect *Hashtable* cu unul *MySet* conduce la un comportament *containsValue()* nedorit
- *Moștenirea implementării* (eng. *implementation inheritance*) = utilizarea moștenirii având drept unic scop reutilizarea codului
- *Moștenirea specificării* sau *moștenirea interfeței* (eng. *specification inheritance, interface inheritance*) = clasificarea conceptelor în ierarhii

# Delegare

- Delegarea reprezintă o alternativă la moștenirea implementării, atunci când se dorește implementarea unei funcționalități prin reutilizare
- Se spune că o clasă *delegă* unei alte clase în cazul în care implementează o operație retrimițând mesajul aferent către cea din urmă
- Implementarea clasei *MySet* prin delegare către *Hashtable* rezolvă problemele menționate anterior
  - *Modificabilitate* - Reprezentarea internă a clasei *MySet* poate fi schimbată fără a-i afecta clienții
  - *Subtipizare* - un obiect *MySet* nu poate fi substituit unui obiect *Hashtable* => codul client care utilizează *Hashtable* nu va fi afectat
- Delegarea este de preferat moștenirii implementării, din rațiuni de modificabilitate și neafectare a codului existent; moștenirea specificării este de preferat delegării, în situații de subtipizare, din rațiuni de extensibilitate

## Delegare (cont.)

- *Soluția 2: Clasa MySet delegă către Hashtable*



```
public class MySet {  
  
    private Hashtable table;  
  
    public MySet()  
    {  
        table = new Hashtable();  
    }  
  
    public void put(Object element)  
    {  
        if (!table.containsKey(element))  
            table.put(element, this);  
    }  
  
    public boolean containsValue(Object element)  
    {  
        return table.containsKey(element);  
    }  
  
    /* ... */  
}
```

# Principiul Liskov al substituției

- Oferă o definiție formală conceptului de *moștenire a specificării*
- (B. Liskov, 1988) Dacă pentru fiecare obiect  $o_1$  de tipul  $S$  există un obiect  $o_2$  de tipul  $T$ , astfel încât orice program  $P$  definit în termenii lui  $T$  își păstrează comportamentul atunci când  $o_2$  este substituit cu  $o_1$ , atunci  $S$  este un subtip al lui  $T$ .
- $\Leftrightarrow$  (B. Bruegge) Dacă un obiect de tip  $S$  poate fi utilizat oriunde este așteptat un obiect de tip  $T$ , atunci  $S$  este un subtip al lui  $T$
- $\Leftrightarrow$  (R. Martin) Subtipurile trebuie să poată substitui tipurile lor de bază (eng. *Sub-types must be substitutable for their base types*)



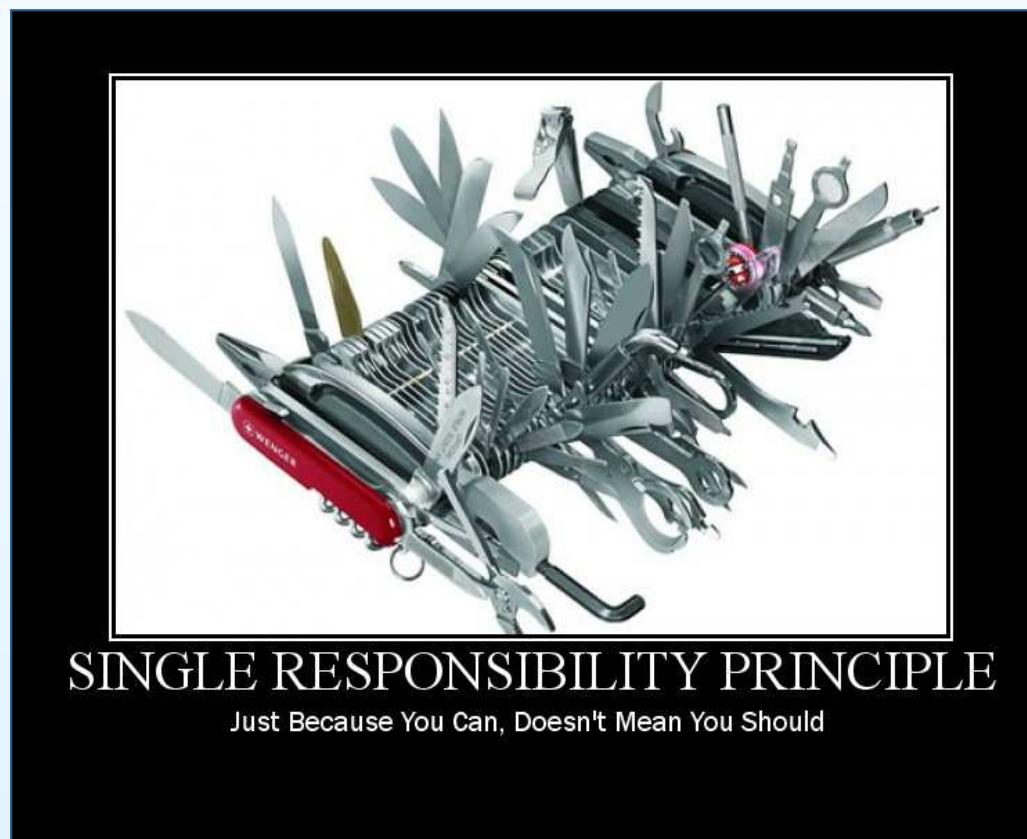
# Principiile SOLID de proiectare a claselor

- SOLID =
  - Single Responsibility Principle (Principiul responsabilității unice) +
  - Open Closed Principle (Principiul deschis/închis) +
  - Liskov Substitution Principle (Principiul Liskov al substituției) +
  - Interface Segregation Principle (Principiul separării interfețelor) +
  - Dependency Inversion Principle (Principiul inversării dependențelor)



## Principiul responsabilității unice

- Nu trebuie să existe niciodată mai mult de un motiv pentru ca o clasă să sufere modificări (eng. *There should never be more than one reason for a class to change*)
- <=> Nu trebuie să existe mai mult de o responsabilitate per clasă



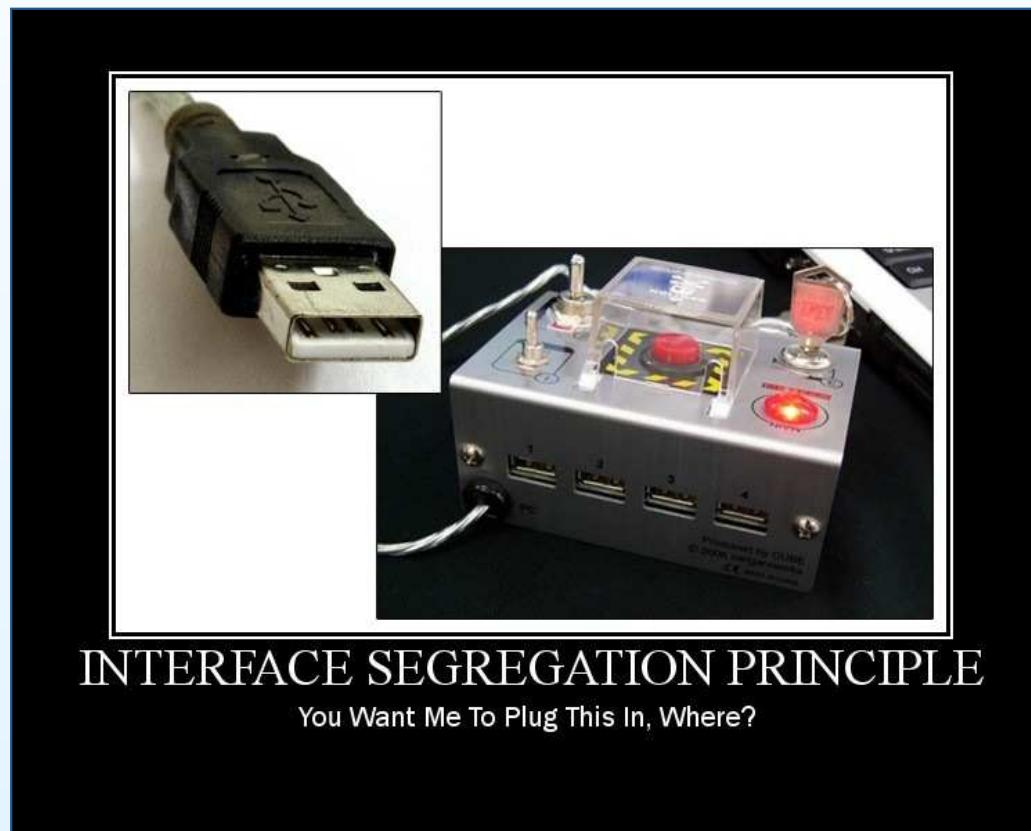
## Principiul deschis/închis

- Entitățile software (clase, module, funcții) trebuie să fie deschise pentru extindere, dar închise pentru modificare (eng. *Software entities (classes, modules, functions) should be open for extension, but closed for modification*)



# Principiul separării interfețelor

- Clienții nu trebuie constrânși să depindă de interfețe pe care nu le utilizează (eng. *Clients should not be forced to depend upon interfaces that they do not use*)



# Principiul inversării dependențelor

- Modulele de nivel înalt nu ar trebui să depindă de module de nivel jos, ambele ar trebui să depindă de abstractizări (eng *High level modules should not depend upon low level modules, both should depend upon abstractions*)
- Abstractizările nu ar trebui să depindă de detalii, detaliile ar trebui să depindă de abstractizări (eng. *Abstractions should not depend upon details, details should depend upon abstractions*)



## Şabloane de proiectare [Gamma et al., 1994]

- Proiectarea de sistem vs. proiectarea obiectuală - paradox generat de obiective conflictuale
  - Obiectivul proiectării de sistem: gestionarea complexității prin crearea unei arhitecturi stabile, descompunând sistemul în subsisteme cu interfețe bine stabilite și slab cuplate => un anumit grad de rigiditate
  - Obiectiv al proiectării obiectuale: flexibilitate - proiectarea unui soft modificabil și extensibil, în vederea minimizării costurilor unor viitoare schimbări în sistem
- Soluție: anticiparea schimbării și proiectarea pentru schimbare (eng. *anticipate change and design for change*)
- Surse comune ale schimbărilor în sistemele soft
  - Furnizor nou / tehnologie nouă
    - unele dintre componentele achiziționate pentru reutilizare în cadrul sistemului sunt înlocuite cu versiuni oferite de alți furnizori
  - Implementare nouă
    - după integrare, unele structuri de date / unii algoritmi sunt înlocuiți cu versiuni mai eficiente, pentru satisfacerea constrângerilor legate de performanță

## Şabloane de proiectare (cont.)

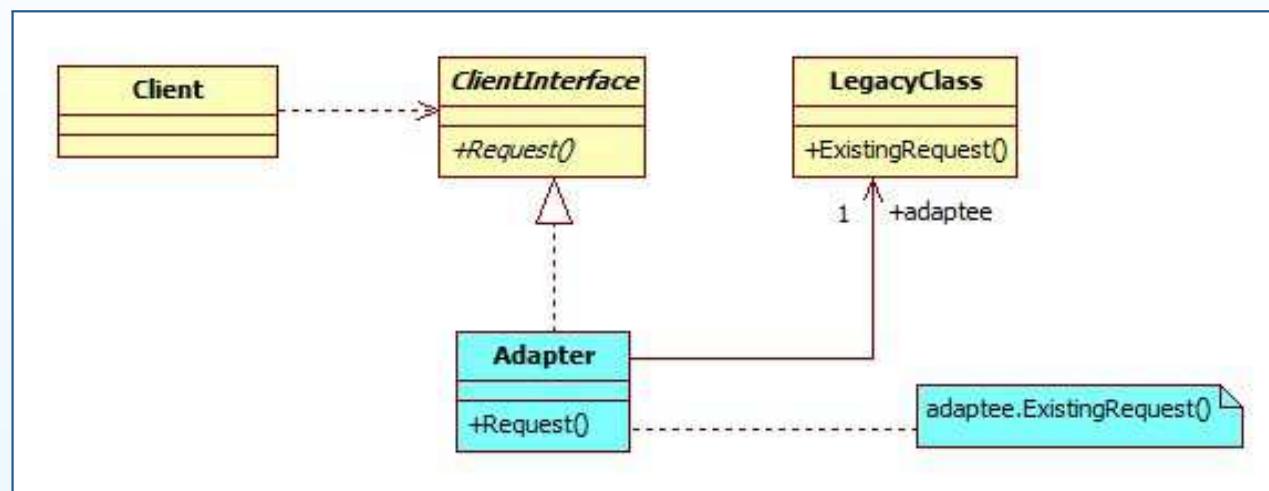
- Funcționalitate nouă
  - testele de validare pot dezvălu absența unor funcționalități din sistem
- O nouă complexitate a domeniului aplicației
  - identificarea unor generalizări posibile ale sistemului în cadrul domeniului de aplicație sau modificarea regulilor de business ale domeniului
- Erori la nivelul cerințelor sau erori de proiectare/implementare
- Şabloane de proiectare adecvate acestor tipuri de schimbări
  - Furnizor nou / tehnologie nouă / implementare nouă
    - *Adapter*
    - *Bridge*
    - *Strategy*
  - Furnizor nou / tehnologie nouă
    - *Abstract Factory*
  - Funcționalitate nouă
    - *Command*
  - O nouă complexitate a domeniului aplicației
    - *Composite*

# Încapsularea componentelor existente cu *Adapter*

- Creșterea complexității sistemelor cerute și scurtarea termenelor de realizare conduce la necesitatea reutilizării unor componente existente, fie din sisteme mai vechi (eng. *legacy systems*), fie achiziționate de la terți
  - În ambele cazuri, este vorba de cod care nu a fost scris pentru sistemul în cauză și care, în general, nu poate fi modificat
  - Arhitectura sistemului curent considerându-se a fi stabilă, nici interfețele acestuia nu ar trebui modificate
  - Gestionație unor astfel de componente presupune încapsularea lor, folosind şablonul *Adapter*
- Şablonul *Adapter*
  - Scop: Transformă interfața unei clase existente (eng. *legacy class*) într-o altă interfață care este așteptată de client, astfel încât clientul și clasa să poată colabora fără nici o modificare la nivelul acestora.
  - Soluție: O clasă *Adapter* implementează interfața *ClientInterface* așteptată de client. Obiectele *Adapter* delegă cererile primite de la client obiectelor *LegacyClass* și efectuează conversiile necesare.

## Şablonul Adapter (cont.)

- Structură:



- Consecințe:

- Clasele *Client* și *LegacyClass* pot colabora fără nici un fel de modificare la nivelul acestora
- *Adapter* lucrează cu *LegacyClass* și oricare dintre subclasele acesteia
- Pentru fiecare specializare a *ClientInterface* trebuie scris un nou *Adapter*

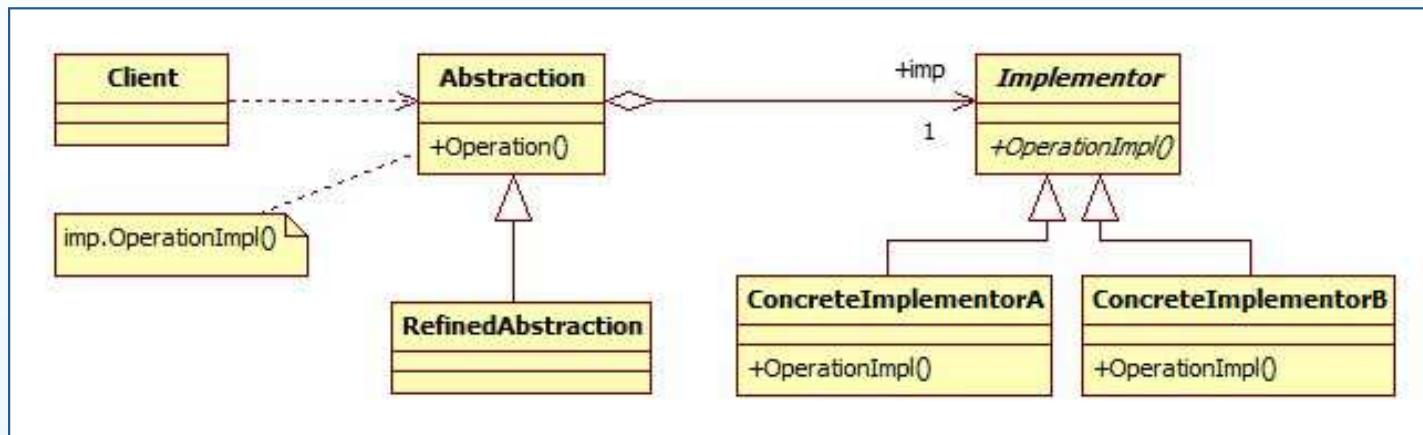
# Decuplarea abstractizărilor de implementări cu *Bridge*

---

- Considerăm problema dezvoltării, integrării (din subsistemele componente) și testării unui sistem
  - Subsistemele pot fi finalizate la momente de timp diferite; pentru a nu întârzia integrarea și testarea sistemului, subsistemele nefinalizate pot fi înlocuite pe moment cu implementări *stub*
  - Pot exista implementări diferite ale același subsistem (ex.: o implementare de referință, care realizează funcționalitatea dorită folosind un algoritm simplu și o implementare mai eficientă, dar cu un grad sporit de complexitate)
  - Este necesară o soluție care să permită substituirea dinamică a implementărilor posibile ale unei aceleiași interfețe, în funcție de necesități
- **Şablonul *Bridge***
  - **Scop:** Decouplează o abstractizare de implementarea ei, astfel încât cele două să poată varia independent. Implementările posibile vor putea fi astfel substituite la execuție.

# Şablonul *Bridge* (cont.)

- *Structură:*



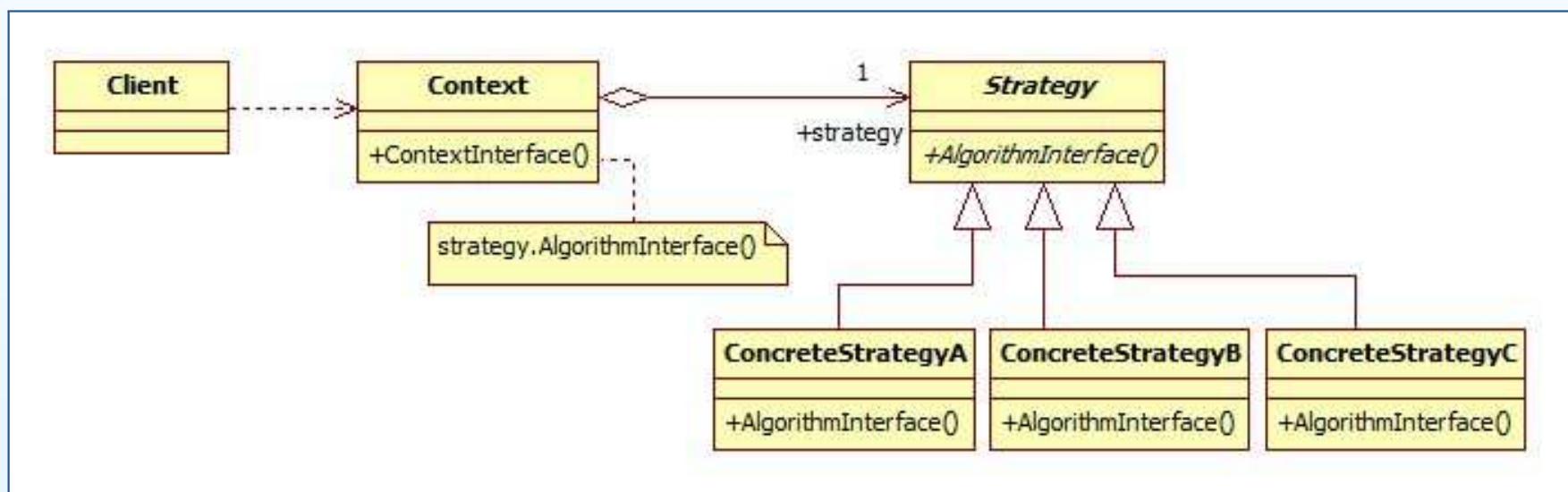
- *Soluție:* Clasa *Abstraction* definește interfața vizibilă clientului. *Implementor* este o clasă abstractă, care declară metode de nivel jos disponibile clasei *Abstraction*. O instanță *Abstraction* reține o referință către instanța *Implementor* curentă. Clasele *Abstraction* și *Implementor* pot fi rafinate independent.
- *Consecințe:*
  - Clientul este protejat de implementările abstracte și concrete
  - Abstractizarea și implementările pot fi rafinate independent

# Încapsularea algoritmilor cu *Strategy*

- Presupunem existența unui număr de algoritmi diferenți pentru rezolvarea aceleiași probleme și necesitatea de a inter schimba în mod dinamic algoritmul utilizat
  - Ex.:
    - Un editor de documente poate folosi algoritmi diferenți pentru împărțirea textului pe rânduri, funcție de preferințe (rânduri de lungime fixă, împărțire în silabe sau nu, etc.)
    - Într-un joc de șah, calculatorul poate folosi strategii diferențe de alegere a următoarei mutări, funcție de nivelul selectat (începător, expert, etc.)
  - Abordări posibile
    - Includerea tuturor algoritmilor la nivelul clasei care îi utilizează și folosirea de instrucțiuni condiționale pentru a-i inter schimba => algoritmii pot fi inter schimbați dinamic, însă clasa respectivă va avea o complexitate sporită și va fi incomod de modificat în condițiile adăugării unui algoritm nou
    - Clasa care utilizează algoritmii implementează o versiune (varianta default), iar pentru celelalte versiuni se definesc clase derivate aferente, care suprascriu doar algoritmul din clasa de bază => un număr mare de clase înrudite care diferă doar prin comportarea aferentă aceluiași algoritm, iar algoritmul nu va putea fi variat dinamic

## Şablonul *Strategy* (cont.)

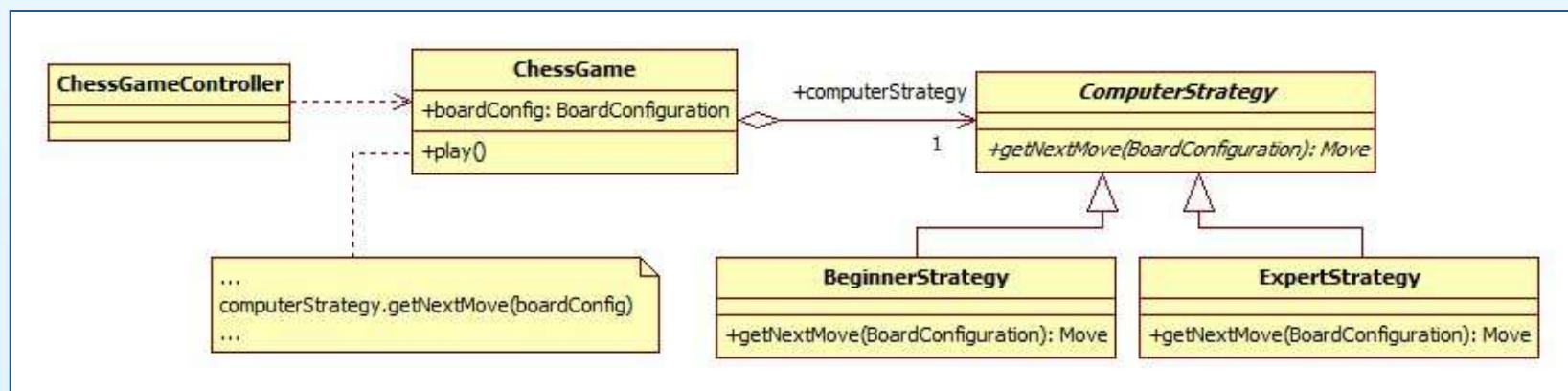
- **Scop:** Defineşte o familie de algoritmi, încapsulează fiecare algoritm şi îi face interschimbabili. Permite algoritmului să varieze independent de clientii care îl utilizează.
- **Structură:**



- **Soluție:**
  - Fiecare dintre algoritmi este încapsulat/implementat de o clasă *ConcreteStrategy*. *Strategy* defineşte interfaţa comună a tuturor algoritmilor suportaţi.

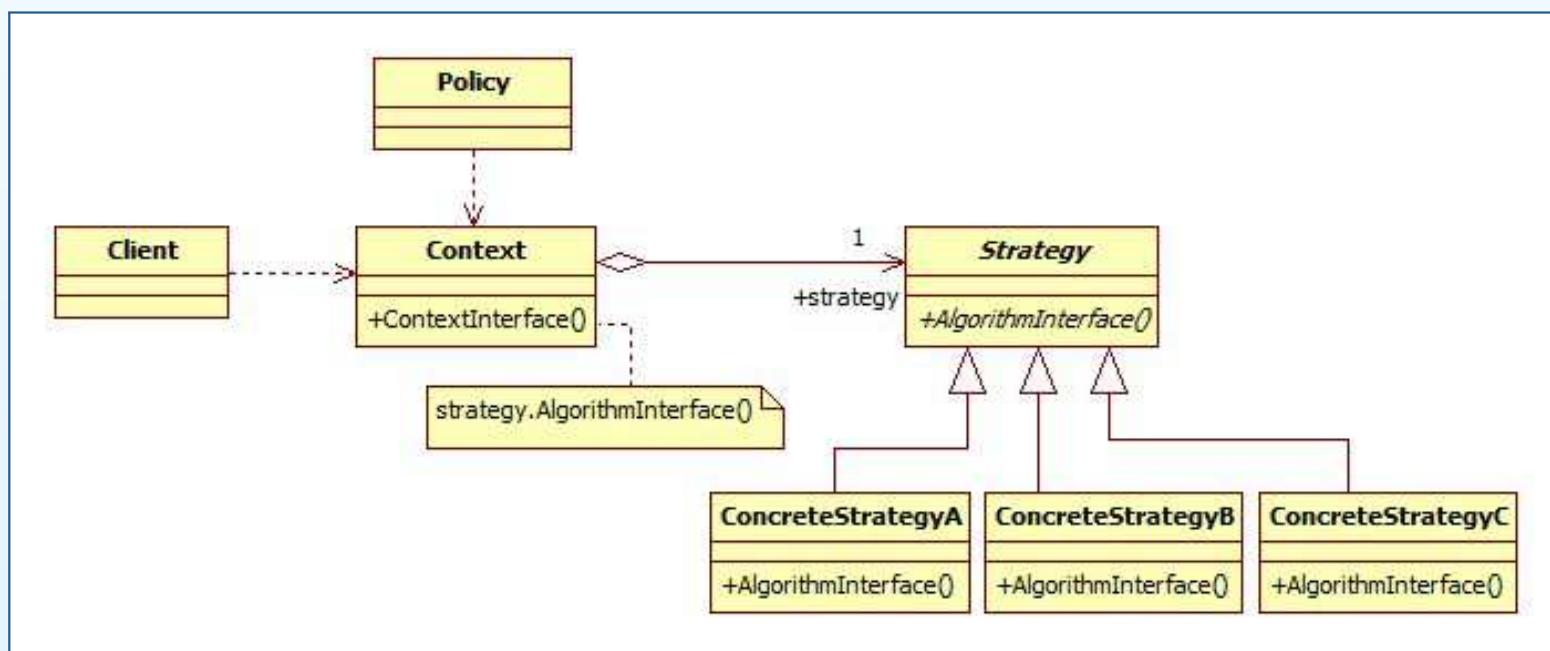
## Şablonul *Strategy* (cont.)

- Obiectele *Context* utilizează această interfață pentru a apela algoritmul definit de o clasă *ConcreteStrategy*. Contextul păstrează o referință către un obiect *Strategy* și îi delegă acestuia responsabilitatea execuției algoritmului, atunci când este cazul.
  - Obiectul strategie este creat și plasat în clasa *Context* de către *Client*.
  - Contextul poate trece către strategie toate date necesare algoritmului, atunci când acesta este apelat. Ca și alternativă, contextul se poate trece pe sine ca și argument în operațiile interfeței *Strategy* și poate oferi o interfață care să-i permită obiectului *Strategy* să-i acceseze datele.
- Ex.: Instanțierea şablonului *Strategy* pentru un joc de sah cu calculatorul



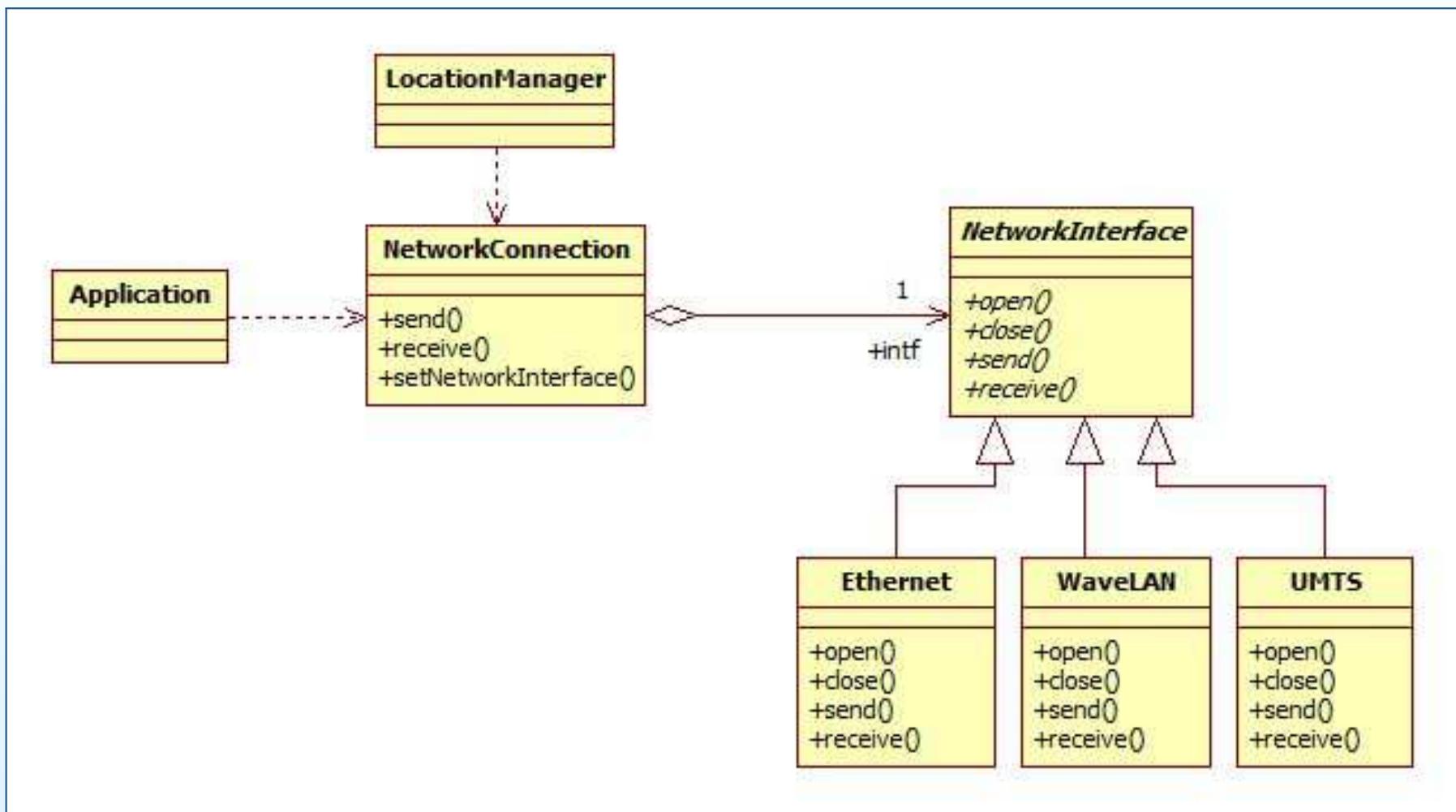
# Şablonul *Strategy* (cont.)

- *Consecințe*
  - Strategiile concrete pot fi substituite în mod transparent relativ la context
  - Pt fi adăugați algoritmi noi fără a modifica contextul sau clientul
- *Variație*
  - Responsabilitatea configurării contextului cu o strategie concretă este atribuită unei clase specializede *Policy*



## Şablonul *Strategy* (cont.)

- Ex.: Schimbarea reţelei în aplicaţii pentru dispozitive mobile (instantiere a variaţiei şablonului *Strategy*)



## Şablonul *Strategy* (cont.)

```
/** The NetworkConnection object represents a single abstract connection
 * used by the Client. This is the Context object in Strategy pattern. */
public class NetworkConnection {
    private String destination;
    private NetworkInterface intf;
    private StringBuffer queue;

    public NetworkConnect(String destination, NetworkInterface intf) {
        this.destination = destination;
        this.intf = intf;
        this.intf.open(destination);
        this.queue = new StringBuffer();
    }
    public void send(byte msg[]) {
        // queue the message to be send in case the network is not ready.
        queue.concat(msg);
        if (intf.isReady()) {
            intf.send(queue);
            queue.setLength(0);
        }
    }
    public byte [] receive() {
        return intf.receive();
    }
    public void setNetworkInterface(NetworkInterface newIntf) {
        intf.close();
        newIntf.open(destination);
        intf = newIntf;
    }
}
```

## Şablonul *Strategy* (cont.)

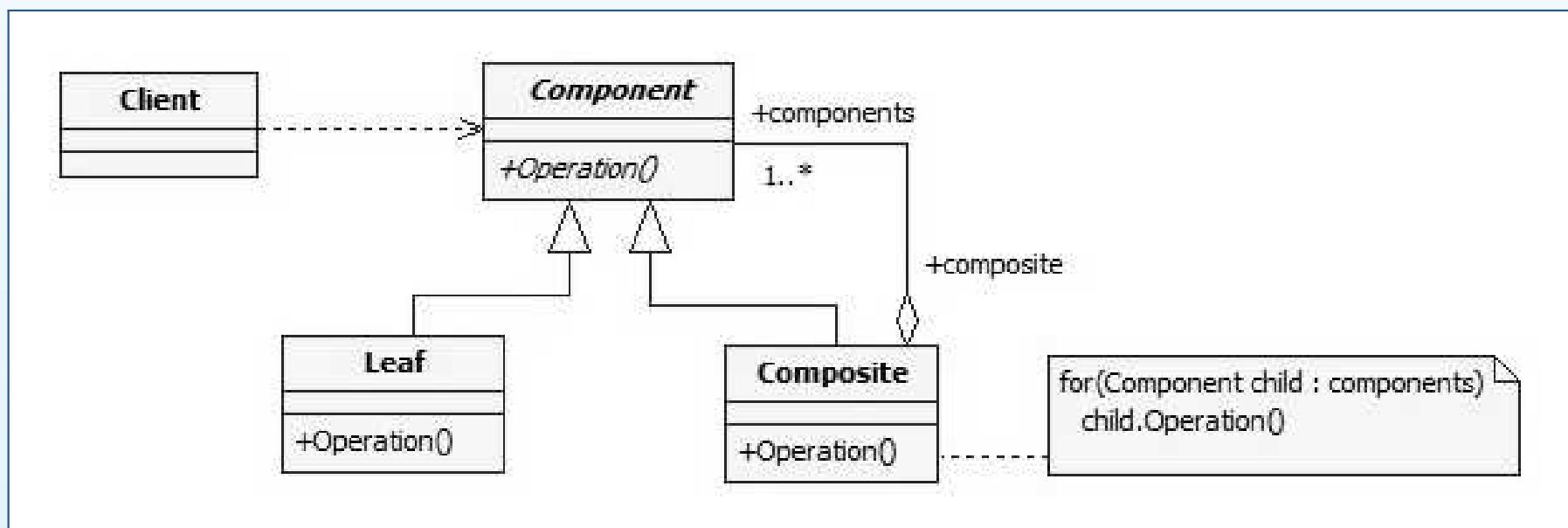
```
/** The LocationManager decides on which NetworkInterface to use based on
 * availability and cost. */
public class LocationManager {
    private NetworkInterface networkIntf;
    private NetworkConnection networkConn;
    /* ... */

    // This method is invoked by the event handler when the location
    // may have changed
    public void doLocation() {
        if (isEthernetAvailable()) {
            networkIntf = new EthernetNetwork();
        } else if (isWaveLANAvailable()) {
            networkIntf = new WaveLANNetwork();
        } else if (isUMTSAvailable()) {
            networkIntf = new UMTSNetwork();
        } else {
            networkIntf = new QueueNetwork();
        }
        networkConn.setNetworkInterface(networkIntf);
    }
}
```

# Încapsularea ierarhiilor cu *Composite*

- **Şablonul *Composite***

- *Tip:* şablon structural
- *Scop:* Permite reprezentarea unor ierarhii de lătime şi adâncime variabilă (recursive), astfel încât frunzele şi aggregatele să fie tratate uniform, prin intermediul unei interfeţe comune.
- *Structură:*



## Şablonul *Composite* (cont.)

- *Soluție:* Interfața *Component* specifică serviciile partajate de *Leaf* și *Composite* (ex. *move(x,y)*, pentru un obiect grafic). Clasa *Composite* agregă obiecte *Component* și implementează aceste servicii iterând peste componentele conținute și delegându-le serviciul în cauză (ex. *move(x,y)* din *Composite* invocă iterativ *move(x,y)* pentru fiecare obiect *Component* conținut). Funcționalitatea concretă este asigurată de implementările serviciilor din *Leaf* (implementarea *move(x,y)* din *Leaf* modifică coordonatele unei primitive grafice și o redesenează).
- *Exemple:*
  - Ierarhii de componente grafice: componentele grafice pot fi organizate în containere, ce pot fi scalate și repoziționate uniform. Un container poate conține alte containere
  - Ierarhii de fișiere și directoare: directoarele pot conține fișiere și alte directoare. Aceleași operații sunt folosite pentru copierea/ștergerea amândurora
  - Descompunerea unui sistem: un subsistem constă din clase și alte subsisteme

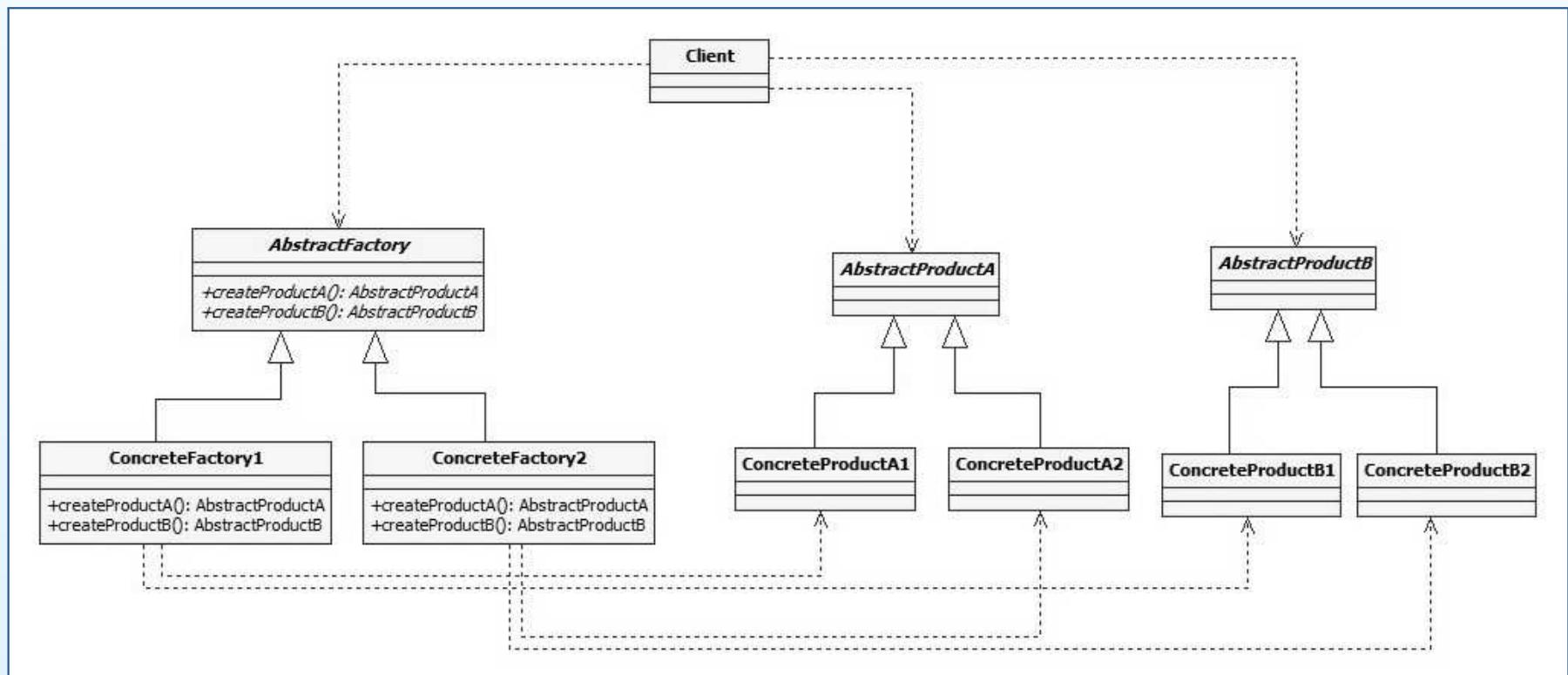
## Şablonul *Composite* (cont.)

- *Consecințe:*
  - Un client utilizează același cod pentru a lucra cu obiecte *Leaf* și *Composite*
  - Noi clase *Leaf* pot fi adăugate fără a schimba ierarhia
  - Comportamentele specifice *Leaf* pot fi modificate fără a schimba ierarhia

# Încapsularea platformelor cu *Abstract Factory*

- *Şablonul Abstract Factory*

- *Tip:* şablon creaţional
- *Scop:* Furnizează o interfaţă pentru crearea familiilor de obiecte înrudite sau dependente, fără specificarea claselor lor concrete.
- *Structură:*



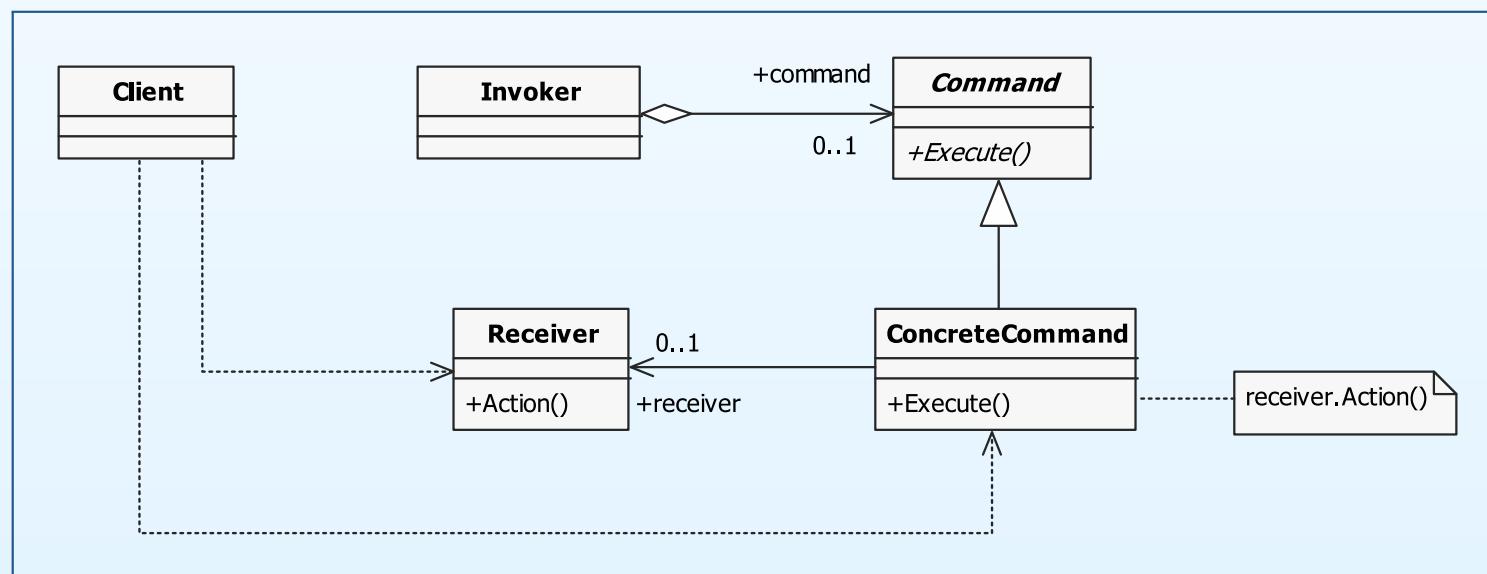
## Şablonul *AbstractFactory* (cont.)

- **Soluție:** O platformă (ex. kit pentru interfață grafică cu utilizatorul) constă dintr-o mulțime de produse (de tip *AbstractProduct*), fiecare reprezentând un anumit concept (ex. buton), suportat de către toate platformele. O clasă *AbstractFactory* declară o interfață cu operații pentru crearea fiecărui tip de produs. O platformă specifică e reprezentată de o mulțime de produse concrete (câte unul pentru fiecare produs abstract), instanțiate de un *ConcreteFactory* (acesta din urmă depinde doar de produsele concrete pe care le instanțiază). Clientul depinde doar de produsele abstracte și de clasa *AbstractFactory*, fapt ce facilitează substituirea platformelor.
- **Consecințe:**
  - Clientul este decuplat de clasele produs concrete.
  - Este posibilă substituirea familiilor de produse la runtime, prin înlocuirea clasei *ConcreteFactory* utilizate.
  - Adăugarea unor noi tipuri de produse este relativ dificilă, întrucât presupune modificarea interfeței *AbstractFactory* și a claselor *ConcreteFactory* existente.

# Încapsularea fluxului de control cu *Command*

- **Şablonul *Command***

- *Tip*: şablon comportamental
- *Scop*: Încapsulează o cerere ca și un obiect, permitând parametrizarea clienților cu diferite cereri, precum și formarea unei cozi sau a unui registru de cereri și asigurarea suportului pentru operațiile ce pot fi anulate ( facilități *undo*).
- *Structură*:



## Şablonul *Command* (cont.)

- *Soluție:* O clasă abstractă *Command* declară interfața suportată de clasele *ConcreteCommand*. O clasa *ConcreteCommand* încapsulează un serviciu ce poate fi aplicat unui *Receiver*. Clasa *Client* creează obiecte *ConcreteCommand* și le asociază obiectelor *Receiver* adecvate. Un obiect *Invoker* execută sau anulează (*undo*) o comandă.
- *Consecințe:*
  - Comanda decouplează obiectul care invocă operația, de cel care știe cum să o efectueze.
  - Comenzile sunt obiecte de prima clasă. Ele pot fi manipulate și extinse, la fel ca și oricare alt obiect.
  - Comenzile pot fi asamblate într-o comandă compusă (instanță a şablonului *Composite*).

## Şabloanele *State* și *Singleton*

---

- Vezi Seminar 7 și [Gamma et al., 1994]

## Referinçe

---

- [Bruegge, 2010] Berndt Bruegge and Allen H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns and Java*, Prentice Hall, 2010.
- [Gamma et al., 1994] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1994.

# Design by Contract (DbC) [1,2]

- The Design by Contract (DbC) methodology has entered software development due to Bertrand Meyer, along with the Eiffel language
- It proposes a contractual approach to the development of object-oriented software components, based on the use of *assertions*
- The approach has been aimed at increasing the *reliability* of object-oriented software components - a critical requirement in the context of large-scale software reuse, as promoted by the object-oriented paradigm
  - *reliability* = *correctness* (software's ability to behave according to the specification) + *robustness* (the ability to properly handle situations outside of the specification)
- Expected to positively contribute to
  - the development of correct and robust OO systems
  - a deeper understanding of inheritance and related concepts (overriding, polymorphism, dynamic binding), by means of the *subcontracting* concept
  - a systematic approach to *exception handling*

# Software Contracts. Assertions

- A generic algorithm to solve a non-trivial task

---

```
Algorithm mainTask is:  
    @subTask_1;  
    @subTask_2;  
    ...  
    @subTask_n;  
End-mainTask
```

---

- Each subtask may be either inlined or triggering the call of a subroutine
- Analogy: calling a subroutine to solve a subtask vs. a real-life situation with a person (client) requiring the services of a third-party (provider) to accomplish a task that he cannot / would not do personally
  - e.g. contracting the services of a fast courier to deliver a package to a particular destination in a foreign city

# Software Contracts. Assertions

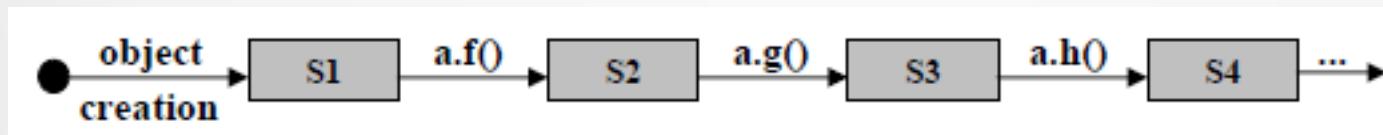
- Characteristics of human contracts involving two parts
  - stipulate the benefits and obligations of each part; a benefit for one part is an obligation for the other
  - may also reference general laws that should be obeyed by both parts
- *Same principles should apply to software development: each time a routine depends on the call of a subroutine to accomplish a subtask, there should be a contractual specification among the client (caller) and supplier (callee)*
- The clauses of software contracts are formalized by means of *assertions*
  - *assertion = expression involving a number of software entities, which state a property that these entities should fulfill at runtime*
  - *closest concept – predicate, implementation – boolean expressions*
  - *some apply to individual routines (pre/post-conditions), other to the class as a whole (invariants)*

# Pre/post-condition assertions

- A <pre, post> pair for a method expresses the software contract that the method in question (provider of some service) publishes for its callers (clients of the service)
- Characteristics of software contracts
  - the precondition defines the situation when a call is legitimate - obligation for the client (caller) and benefit for the provider (method)
  - the postcondition states which properties should be fulfilled once the execution has ended - obligation for the provider (method) and benefit for the client (caller)
- The major contribution brought by DBC in the field of software reliability: precondition = benefit of the service provider
- *DBC non-redundancy principle: The body of a method should never check for a precondition* (as opposed to defensive programming)
- In Eiffel, assertions are part of the language, allowing for runtime monitoring
  - precondition violation = bug in the client, postcondition violation = bug in the supplier

# Invariant assertions

- In addition to pre/post-conditions, that capture the behavior of individual methods, it is possible to express global properties of a class' instances, that should be preserved by all its public methods
- An *invariant* encloses all the semantic constraints and integrity rules applying to the class in question
- Lifecycle of an object



- An assertion *I* is a correct invariant for a class *C* if and only if the following conditions hold
  - each constructor of *C*, applied to arguments that fulfill its precondition, in a state in which the class attributes have default values, leads to a state in which *I* is fulfilled
  - each public method of *C*, applied to a set of arguments and to a state satisfying both *I* and the method precondition, leads to a state satisfying *I*

# Correctness of a class

- Informally, a class is said to be correct with respect to its specification if and only if its implementation, as given by the method bodies, is consistent with its preconditions, postconditions and invariant
- **Definition:** The class  $C$  is said to be correct with respect to its assertions (pre/post-conditions and invariant) if and only if the following conditions hold:
  - (1) [ $\text{default}_C$  and  $\text{pre}_p(x_p)$ ]  $\text{body}_p$  [ $\text{post}_p(x_p)$  and  $\text{INV}$ ]  
for each class constructor  $p$  and each set of valid arguments of  $p - x_p$  and
  - (2) [ $\text{pre}_r(x_r)$  and  $\text{INV}$ ]  $\text{body}_r$  [ $\text{post}_r(x_r)$  and  $\text{INV}$ ]  
for each public class method  $r$  and each set of valid arguments of  $r - x_r$ , where  
 $\text{default}_C$  is an assertion stating that the attributes of  $C$  have default values for their type,  $\text{INV}$  is the invariant of  $C$ ,  $\text{pre}_m$ ,  $\text{post}_m$ ,  $\text{body}_m$  are the precondition, postcondition and body of an arbitrary method  $m$  of  $C$ .

# The purpose of using assertions

- Support in writing correct software, including the means to formally define correctness
  - The writing of explicit contracts comes as a prerequisite of their enforcement in software
- Support for a better software documentation
  - Essential when it comes to reusable assets, see the case of Ariane!
- Support for testing, debugging and quality assurance
  - Levels of runtime assertion monitoring:
    - 1. preconditions only
    - 2. preconditions and postconditions
    - 3. all assertions
  - While testing, enforce level 3, in production, there is a tradeoff between trust in the code, efficiency level desired and critical nature of the application
- Support for the development of fault tolerant systems

# Defensive Programming [3]

- Analogy to *defensive driving*
  - *Defensive driving*: You can never be sure what the others might do, so take responsibility of protecting yourself, such that another driver's mistake won't hurt you!
  - *Defensive programming*: If a routine is passed bad data, it should not be hurt, even if the bad data is someone else's fault (humans, software).
- The core idea of defensive programming is guarding against unexpected errors
- Acknowledges that errors happen and invites programmers to write code accordingly
- Comprises a set of techniques that make errors easier to detect, easier to repair and less damaging in production code
- Should serve as a complement to defect-prevention techniques (iterative design, pseudocode first, design inspections, etc.)
- Protecting from invalid input involves
  - Checking all data received from the outside (from users, files, network, etc.)
    - numeric values should be between tolerances, strings – short enough to handle and obeying to their intended semantics
  - Checking all input parameters
  - Deciding on how to deal with bad data

# References

- [1] Meyer, B., *Object-Oriented Software Construction* (2nd ed.), Prentice-Hall, 1997. (Chapter 11 – Design by Contract: building reliable software)
- [2] Meyer, B., *Applying „Design by Contract”*, IEEE Computer 25(10):40-51, 1992.
- [3] McConnel, S., *Code Complete* (2nd ed.), Microsoft Press, 2004. (Chapter 8 – Defensive Programming)

## *Curs 9*

*Proiectarea obiectuală: Specificarea interfețelor*

*Suport de curs bazat pe B. Bruegge and A.H. Dutoit*

*"Object-Oriented Software Engineering using UML, Patterns, and Java"*

# Specificarea interfețelor

- Scopul proiectării obiectuale este reprezentat de identificarea și rafinarea obiectelor din domeniul soluției necesare realizării comportamentului subsistemelor identificate în etapa proiectării de sistem
- Produse existente până în momentul etapei de specificare a interfețelor
  - *modelul obiectual de analiză*: entități din domeniul problemei (cu atrbute, relații, unele operații) + obiecte boundary și control inteligibile utilizatorului
  - *descompunerea sistemului*: subsisteme, servicii oferite, dependențe între subsisteme, obiecte noi din domeniul soluției
  - *mapare hardware/software*: mașina virtuală = componente reutilizate pentru serviciile standard
  - *șabloane de proiectare reutilizate, componente de bibliotecă reutilizate pentru structuri de date și servicii de bază*
- Subactivități în specificarea interfețelor
  - Identificarea atrbutelor și operațiilor lipsă
  - Specificarea vizibilității și signaturii operațiilor
  - **Specificare pre/post-condițiilor pentru operații și a invarianților de tip**

# Object Constraint Language (OCL)

- OCL [Warmer, 1999] - limbaj formal, utilizat pentru definirea de expresii pe modelele UML
  - introdus inițial ca și limbaj de modelare la IBM, astăzi standard OMG [OMG, 2014]
- Caracteristici OCL
  - *Limbaj complementar* (UML-ului)
    - OCL nu e un limbaj de sine stătător; a apărut din necesitatea de a acoperi problemele de expresivitate ale UML, a cărui natură diagramatică nu permite formularea multora dintre constrângerile caracteristice sistemelor nontriviale
    - Pentru dezvoltatori, specificațiile OCL practice sunt doar cele formulate în contextul tipurilor de date utilizator introduse în modelul UML
  - *Limbaj declarativ* (limbaj de specificare pur, fără efecte secundare)
    - Evaluarea expresiilor OCL nu modifică starea modelului UML asociat
  - *Limbaj puternic tipizat*
    - Fiecare (sub)expresie OCL are un tip și face obiectul verificărilor privind conformanța tipurilor

# Object Constraint Language (OCL) (cont.)

- *Limbaj bazat pe logica de ordinul întâi*
- *Limbaj care suportă principalele caracteristici OOP*
  - Specificațiile OCL sunt moștenite în descendenți, unde pot fi supradefinite
  - Redefinirea constrângerilor se conformează regulilor DbC
  - Limbajul suportă up/down-casting și verificări de tip
- **Utilitate**
  - *navigarea modelului*
    - interogarea informației din model, prin navigări repetitive ale asocierilor, folosind nume de roluri
  - *specificarea aserțiunilor*
    - definirea explicită a pre/post-condițiilor și invariantei, conform principiilor DbC
  - *specificare comportamentală*
    - specificarea comportamentului observatorilor (operațiilor de interogare) din model, specificarea regulilor de derivare pentru attribute/referințele derivate, definirea de noi attribute sau operații
  - *specificarea gărzilor, specificarea invariantei de tip pentru stereotipuri*

# Sistemul de tipuri OCL

- OCL fiind complementar UML-ului, orice clasificator dintr-un model UML este un tip OCL valid în cadrul oricărei expresii atașate modelului în cauză
- Biblioteca standard OCL - tipuri predefinite, independente de model
  - Tipuri primitive: Integer, UnlimitedNatural, Real, Boolean, String
  - Tipuri specifice OCL: OclAny, OclVoid, OclInvalid, OclMessage
  - Tipuri colecție: Collection, Set, OrderedSet, Sequence, Bag
  - Enumeration, TupleType
- *Tipuri specifice OCL*
  - Tipul OclAny
    - Supertipul tuturor tipurilor OCL (=> în particular, fiecare clasă din modelul UML moștenește toate operațiile definite în OclAny)
    - Operații
      - = (object2:OclAny) : Boolean - egalitatea a două obiecte
      - <> (object2:OclAny) : Boolean - egalitatea a două obiecte
      - oclIsTypeOf (type:Classifier) : Boolean - conformanța tipurilor

# Sistemul de tipuri OCL (cont.)

- **Operații**

`oclIsKindOf(type:Classifier) : Boolean` - conformanța tipurilor

`oclType() : Classifier` - inferă tipul

`oclAsType(type:Classifier) : T` - conversie

`oclIsNew() : Boolean` - utilizat în postcondiția unei operații, verifică dacă obiectul a fost creat în timpul execuției operației respective

`oclIsUndefined() : Boolean` - verifică dacă obiectul există/e definit

`oclIsValid() : Boolean` - verifică dacă obiectul este valid

- **Tipul OclVoid**

- Tip care se conformează tuturor tipurilor OCL, mai puțin `OclInvalid`
- Denotă absența unei valori (sau o valoare necunoscută la momentul respectiv), singura valoare a tipului e literalul `null`

- **Tipul OclInvalid**

- Tip care se conformează tuturor tipurilor OCL, inclusiv `OclVoid`
- Singura valoare este `invalid`, ce poate rezulta din excepții privind împărțirea la zero, accesarea unei valori de pe un index nepermis, etc.

# Sistemul de tipuri OCL (cont.)

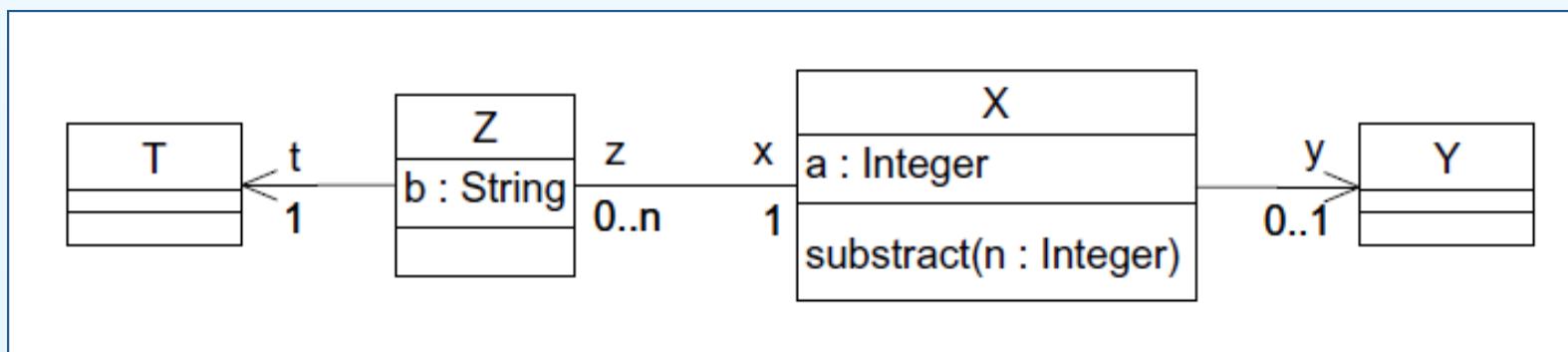
- Tipurile colecție (tipuri template )
  - Operațiile pe colecții sunt apelate folosind notația →
  - Modalități de a obține o colecție: prin navigare, ca rezultat al unei operații pe colecții, folosind specificații cu literali (`Set{ }`, `Bag{1,2,1}`, `Sequence{1..10}`)
  - Tipul `Collection`
    - Supertipul abstract al celorlalte tipuri colecție din biblioteca standard OCL (`Set`, `OrderedSet`, `Bag`, `Sequence`)
    - Definește operații cu semantică comună tuturor subtipurilor
    - Unele operații sunt redefinite în subtipuri, având o postcondiție mai puternică sau o valoare de return mai specializată
    - Operații uzuale
      - `size() : Integer`, `isEmpty() : Boolean`, `notEmpty() : Boolean`
      - `count(object : T) : Integer`
      - `includes(object : T) : Boolean`, `excludes(object : T) : Boolean`
      - `includesAll(c2 : Collection(T)) : Boolean`
      - `excludesAll(c2 : Collection(T)) : Boolean`
      - `asSet() : Set(T)`, `asOrderedSet() : OrderedSet(T)`
      - `asBag() : Bag(T)`, `asSequence() : Sequence(T)`

## Sistemul de tipuri OCL (cont.)

- Iteratori pe colecții
  - **select**: source->select(iterator | body) - returnează subcolecția colecției source pentru care body se evaluează la true
  - **reject**: source->reject(iterator | body) - returnează subcolecția colecției source pentru care body se evaluează la false
  - **forAll**: source->forAll(iterator | body) - returnează true dacă pentru toate elementele din colecția source body se evaluează la true
  - **exists**: source->exists(iterator | body) - returnează true dacă există cel puțin un element în colecția source pentru care body se evaluează la true
  - **one**: source->one(iterator | body) - returnează true dacă există exact un element al colecției source pentru care body se evaluează la true
  - **any**: source->any(iterator | body) - returnează un element arbitrar din colecția source pentru care body se evaluează la true
  - **isUnique**: source->isUnique(iterator | body) - returnează true dacă evaluările lui body conduc la elemente distințe
  - **collect**: source->collect(iterator | body) - returnează colecția rezultată prin aplicarea lui body pe fiecare element al colecției sursă

# Proprietăți și navigare

- O expresie OCL este formulată în contextul unui anumit tip
  - În cadrul respectivei expresii, instanța contextuală este referită de cuvântul cheie `self`
  - `self` poate fi omis, atunci când nu există risc de ambiguități
  - Pornind de la instanța contextuală, se pot accesa oricare dintre atributele, operațiile de tip interrogare sau capetele opuse de asociere, în stilul orientat-obiect clasic (folosind notația `".."`)
- Ex.:



- În contextul clasei `X`, `self.a` și `self.y` sunt două expresii OCL având tipurile `Integer`, respectiv `Y` (prima accesează un atribut, a doua presupune o navigare a unei asociieri folosind numele de rol al capătului opus)

## Proprietăți și navigare (cont.)

- Reguli de tipizare la navigare
  - Atunci când multiplicitatea capătului opus de asociere este cel mult 1, tipul expresiei rezultate prin navigare într-un singur pas este dat de clasificatorul de la capătul opus
  - Atunci când valoarea maximă a multiplicității capătului opus de asociere este cel puțin 1, tipul expresiei rezultate prin navigare într-un singur pas este Set sau OrderedSet, funcție de prezență sau absență constrângerei {ordered} pe capătul opus
    - În contextul `x`, tipul expresiei `self.z` este `Set(Z)`
  - Atunci când navigarea presupune mai mulți pași, tipul expresiei rezultat este Bag
    - În contextul `x`, tipul expresiei `self.z.t` este `Bag(T)`
- În afară de accesarea proprietăților instanței contextuale, este posibilă utilizarea operației `allInstances` pe un anumit clasificator => mulțimea tuturor obiectelor existente, având acel clasificator ca și tip
  - `x.allInstances() -> size()` - numărul obiectelor curente de tip `x`

# Design by Contract în OCL

- Constrângeri de tip invariant

```
context X  
inv invX1: self.a >= 0
```

- Un invariant se formulează în contextul unui clasificator, ce dă tipul instanței contextuale
- Un invariant este introdus de cuvântul cheie **inv**, urmat de un identificator optional și de expresia OCL a invariantului

- Constrângeri de tip precondition/postcondition

```
context X::subtract (n: Integer)  
pre subtractPre: self.a >= n  
post subtractPost: self.a = self.a@pre - n
```

- Clauza context menționează signatura operației aferente (**self** va fi o instanță a tipului care deține acea operație)
- Într-o postcondiție, notatia **@pre** referă valoare unui obiect/unei proprietăți înainte de execuția operației în cauză

# Structurarea specificațiilor OCL

- Mecanismul **let**

- Permite extragerea unei subexpresii OCL redundante într-o variabilă
- Crește intelibrabilitatea constrângerii și eficiența evaluării acesteia (prin efectuarea calculului aferent o singură dată)

```
context X

inv invX2: let allT:Bag(T) = self.z.t in
            allT->size() = allT->asSet()->size()
```

- Constrângeri de tip **definition**

- Permit reutilizarea unei expresii OCL la nivelul mai multor constrângerii
- Introduse prin cuvântul cheie **def**, permit definirea unor atribute/operații auxiliare la nivelul unui clasificator

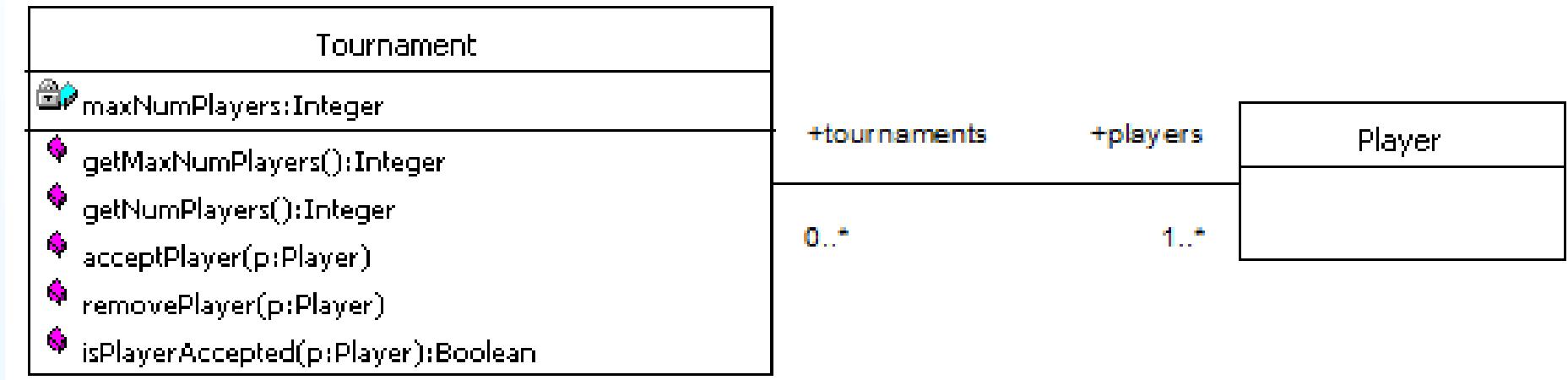
```
context X

def: hasY:Boolean = not self.y.oclIsUndefined()
def: hasZWithBValue(value:String):Boolean =
        self.z->exists(zz | zz.b = value)
```

# Iteratori - variante de sintaxă

- **select** (analog reject, forAll, exists)
  - collection->select(v:Type | boolean-expression-with-v)
  - collection->select(v | boolean-expression-with-v)
  - collection->select(boolean-expression)
- **collect**
  - collection->collect(v:Type | expression-with-v)
  - collection->collect(v | expression-with-v)
  - collection->collect(expression)
- **iterate**
  - collection->iterate(elem:Type; acc:Type = <expression> | expression-with-elem-and-acc)
  - Cel mai generic iterator, ceilalți pot fi exprimați folosindu-l pe iterate
  - Ex.: collection->collect(x:T | x.property) is equivalent to collection->iterate(x:T; acc:Bag(T2) = Bag{} | acc->including(x.property))

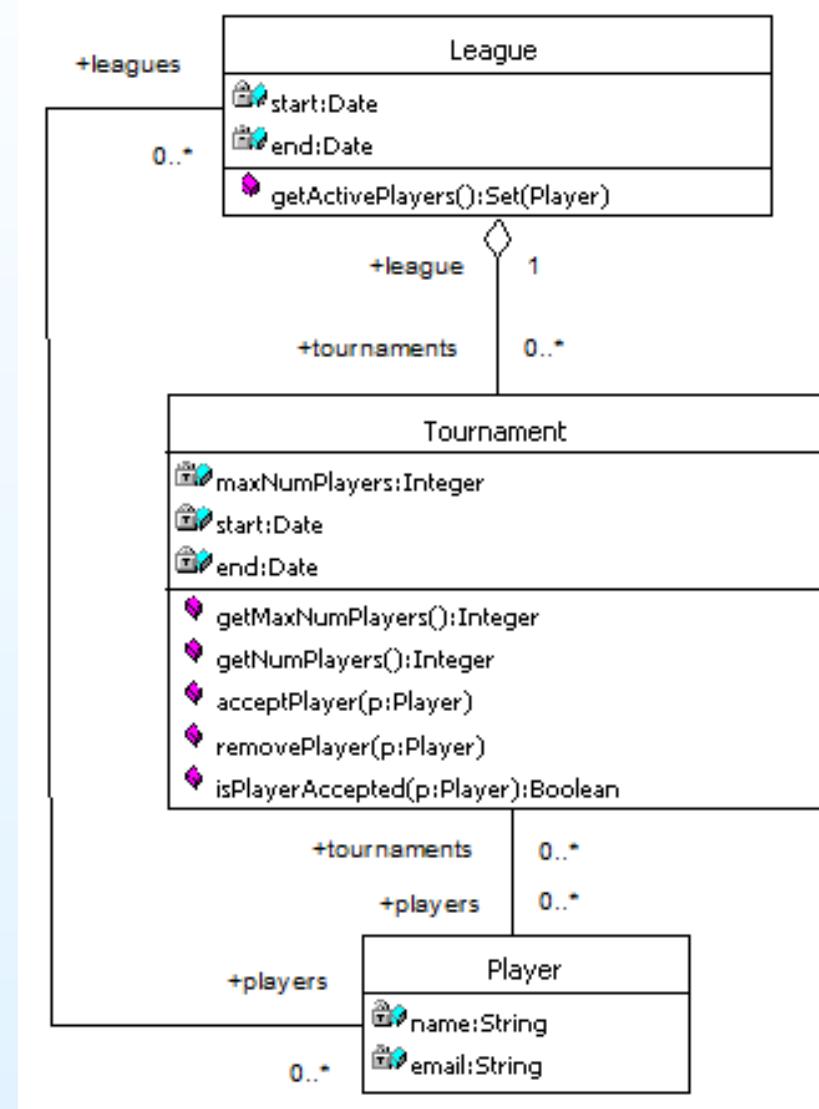
# Exemple OCL



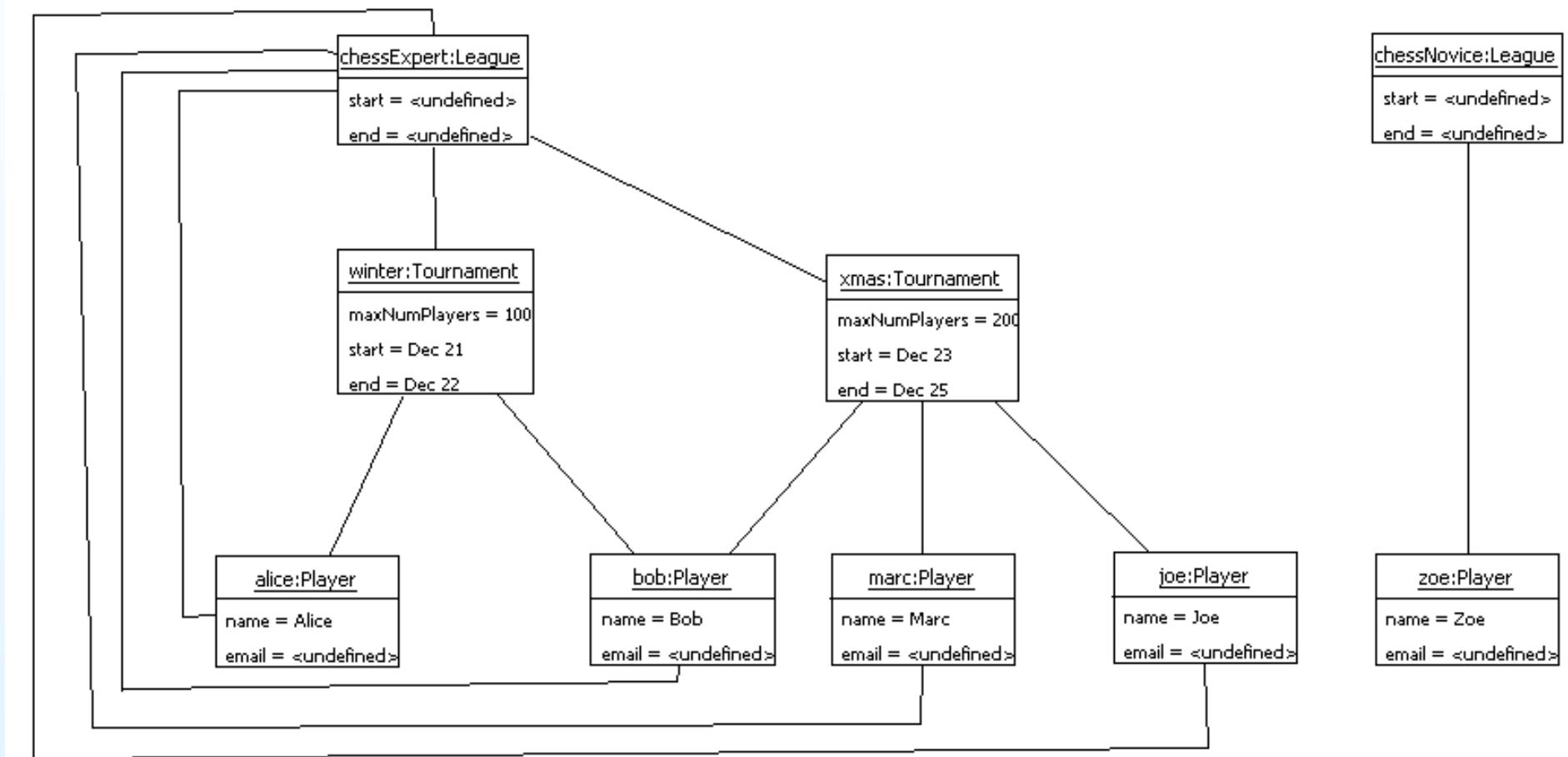
```
context Tournament
    inv maxNumPlayersPositive:
        self.getMaxNumPlayers() > 0

context Tournament::acceptPlayer(p: Player)
    pre: self.getNumPlayers() < self.getMaxNumPlayers() and
        not self.isPlayerAccepted(p)
    post: self.isPlayerAccepted(p) and
        self.getNumPlayers() = self@pre.getNumPlayers() + 1
```

## Exemple OCL (cont.)



## Exemple OCL (cont.)



## Exemple OCL (cont.)

- Durata unui turneu trebuie să fie sub o săptămână

```
context Tournament
    inv maxDuration: self.end - self.start < 7
```

- Toți jucătorii care participă la un turneu trebuie să fie înregistrați în liga aferentă acestuia

```
context Tournament
    inv allPlayersRegisteredWithLeague:
        self.league.players->includesAll(self.players)
```

```
context Tournament::acceptPlayer(p:Player)
    pre playerIsInLeague: self.league.players->includes(p)
```

## Exemple OCL (cont.)

- Jucătorii activi dintr-o ligă sunt cei care au participat la cel puțin un turneu al ligii

```
context League::getActivePlayers() : Set(Player)
    post: result = self.tournaments.players->asSet()
```

- Toate turneele unei ligi au loc în intervalul de timp aferent ligii

```
context League
    inv: self.tournaments->forAll(t:Tournament |
        t.start.after(self.start) and t.end.before(self.end))
```

- În orice ligă există cel puțin un turneu planificat în prima zi a ligii

```
context League
    inv: self.tournaments->exists(t:Tournament |
        t.start = self.start)
```

# Moștenirea contractelor

- Problema moștenirii contractelor
  - În limbajele polimorfice, o referință la un obiect al clasei de bază poate fi substituită de o referință la un obiect al unei clase derivate
  - Codul client, scris în termenii clasei de bază, poate folosi obiecte ale claselor derivate, fără a avea cunoștință de acest fapt
  - => Clientul se aşteaptă ca un contract formulat relativ la clasa de bază, să fie respectat și de clasele derivate
- Reguli privind moștenirea contractelor (consecință a principiului Liskov al substituției)
  - *Precondiții*: Unei metode dintr-o subclasă îi este permis să slăbească precondiția metodei pe care o supradefinește (o metodă care supradefinește poate gestiona mai multe cazuri decât cea supradefinită)
  - *Postcondiții*: O metodă care supradefinește trebuie să asigure o postcondiție cel puțin la fel de puternică precum cea supradefinită
  - *Invarianți*: O subclasă trebuie să respecte toți invarianții superclaselor sale; poate, eventual, introduce invarianți mai puternici decât cei moșteniți

## Referinte

---

- [OMG, 2014] Object Management Group, *Object Constraint Language - version 2.4*, February 2014.
- [Warmer, 1999] J. Warmer, A. Kleppe, *Object constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.

## *Curs 10*

### *Transformarea modelelor în cod*

*Suport de curs bazat pe B. Bruegge and A.H. Dutoit*

*"Object-Oriented Software Engineering using UML, Patterns, and Java"*

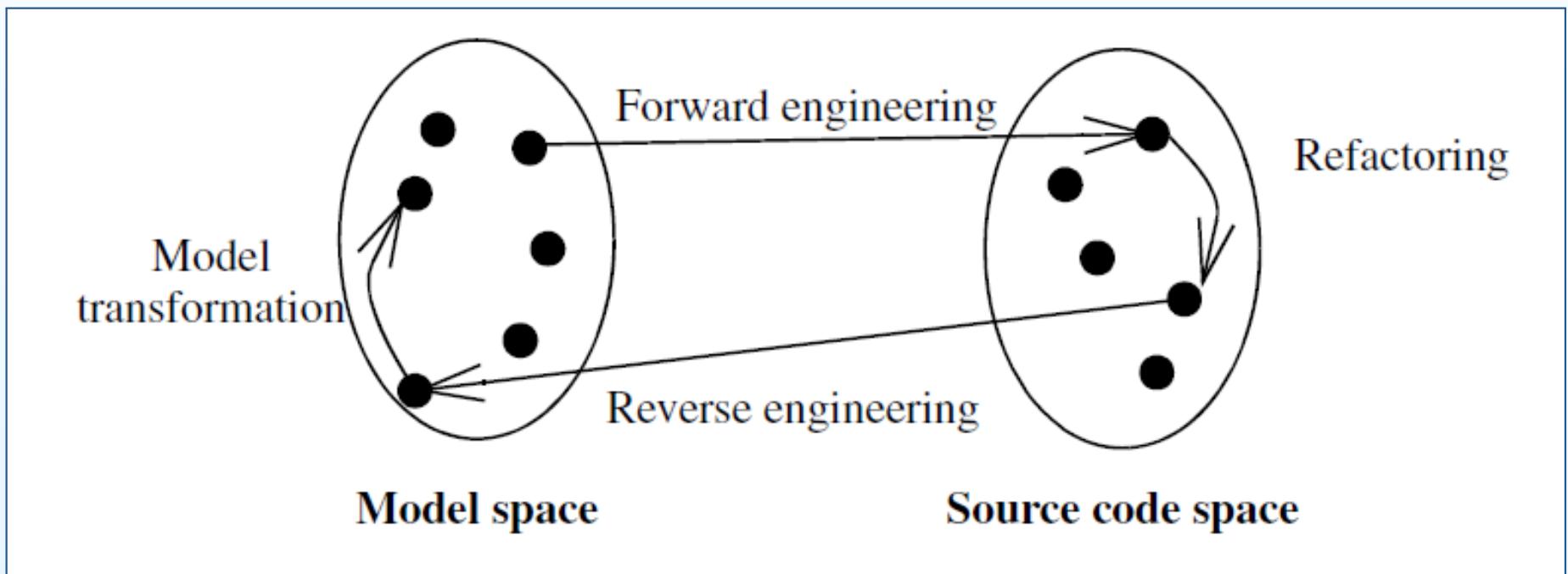
# Modele și transformări

- O *transformare* are drept scop îmbunătățirea unei caracteristici a unui model (ex.: modularitatea), cu păstrarea celorlalte proprietăți ale acestuia (ex.: funcționalitatea)
  - O transformare este, de obicei, localizată, afectează un număr relativ mic de clase/attribute/operații și se execută într-o succesiune de pași mărunți
- Astfel de transformări caracterizează preponderent activitățile legate de proiectarea obiectuală și implementarea sistemului
  - *Optimizare* - îndeplinirea cerințelor legate de performanța sistemului, prin
    - reducerea multiplicării asocierilor, pentru a crește viteza interogărilor
    - adăugarea unor asocieri redundante, pentru eficiență
    - introducerea unor attribute derivate, pentru a îmbunătăți timpul de acces la obiecte
  - *Reprezentarea asocierilor* - implementarea asocierilor în cod folosind (colecții de) referințe
  - *Reprezentarea contractelor* - descrierea comportamentului sistemului în cazul violării contractelor, folosind excepții
  - *Reprezentarea entităților persistente* - maparea claselor la nivelul depozitelor de date (baze de date, fisiere text, etc.)

# Tipuri de transformări

- *Transformări la nivelul modelului* (eng. *model transformations*)
  - Operează pe un model, au ca și rezultat un model
  - Ex.: transformarea unui atribut (atribut *adresă*, reprezentată ca și string) într-o clasă (clasa *Adresă*, cu attribute *stradă*, *număr*, *oraș*, *cod poștal* etc.)
- *Refactorizări* (eng. *refactorings*)
  - Operează pe cod sursă, au ca și rezultat cod sursă
  - Similar transformărilor la nivelul modelului, îmbunătățesc un aspect al sistemului, fără a-i afecta funcționalitatea
- *Inginerie directă* (eng. *forward engineering*)
  - Produce un fragment de cod aferent unui model obiectual
  - Multe dintre conceptele de modelare (ex.: attribute, asocieri, signaturi de operații) pot fi transformate automat în cod sursă; corpul metodelor, precum și metodele adiționale (private) sunt inserate manual de către dezvoltator
- *Inginerie inversă* (eng. *reverse engineering*)
  - Produce un model, pe baza unui fragment de cod sursă
  - Utilă atunci când modelul de proiectare nu (mai) există sau atunci când modelul și codul au evoluat desincronizat

## Tipuri de transformări (cont.)



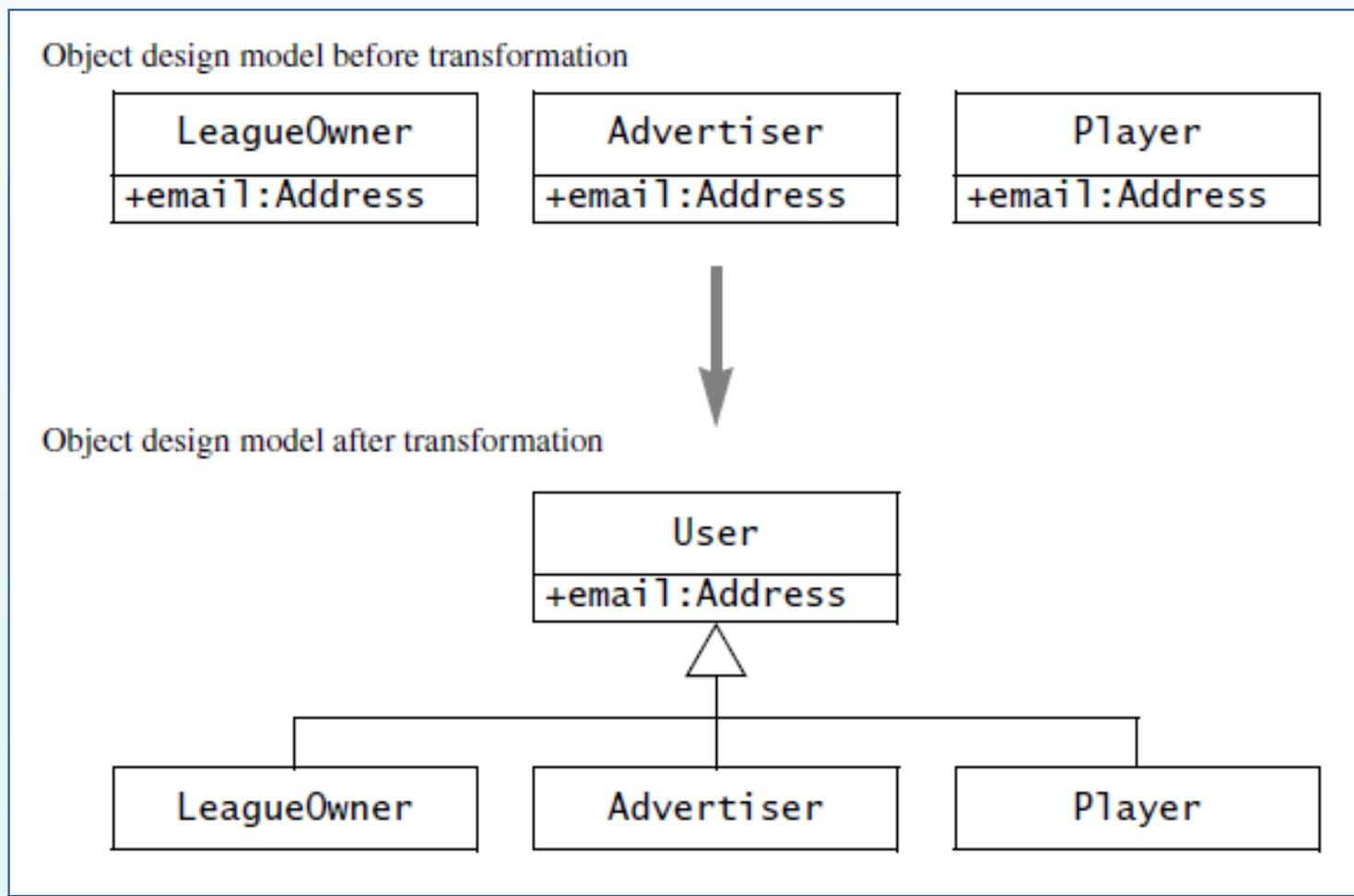
## Transformări la nivelul modelului

---

- O astfel de transformare este aplicată unui model obiectual și rezultă într-un nou model obiectual
- Obiectivul este simplificarea, detalierea sau optimizarea modelului inițial, în conformitate cu cerințele din specificație
- O transformare la nivelul modelului poate să adauge, să eliminate sau să redenumească clase, operații, asocieri sau atrbute
- Întreg procesul de dezvoltare poate fi considerat ca și o succesiune de transformări de modele, începând cu modelul de analiză și terminând cu cel obiectual de proiectare, fiecare astfel de transformare adăugând detalii care țin de domeniul soluției
- Deși aplicarea unei astfel de transformări poate fi, de cele mai multe ori, automatizată, identificarea tipului de transformare de aplicat, precum și a claselor concrete implicate necesită raționament și experiență

## Transformări la nivelul modelului (cont.)

- Ex. 10.1: utilizarea unei transformări pentru introducerea unei ierarhii de clase și eliminarea redundanței din modelul obiectual de analiză



# Refactorizări

- O *refactorizare* reprezintă o transformare a codului sursă, care crește inteligențialitatea sau modificabilitatea acestuia, fără a-i schimba comportamentul [Fowler, 2000]
  - O refactorizare are drept scop îmbunătățirea design-ului unui sistem funcțional, focusându-se pe o anumită metodă sau pe un anumit camp al unei clase
  - Pentru a asigura păstrarea neschimbării a comportamentului sistemului, o refactorizare se realizează incremental, pașii de refactorizare fiind intercalăți cu teste
- *Ex. 10.2:* Transformarea de model din *Ex.10.1* corespunde unei serii de 3 refactorizări
  1. Refactorizarea *Pull Up Field*
    - Transferă campul *email* din subclase în superclasa *User*
  2. Refactorizarea *Pull Up Constructor Body*
    - Transferă codul de initializare din subclase în superclasă
  3. Refactorizarea *Pull Up Method*
    - Transferă metodele care utilizează campul *email* din subclase în superclasă

# Pașii refactorizării *Pull Up Field*

1. Inspectează clasele *Player*, *LeagueOwner* și *Advertiser*, pentru a certifica echivalența semantică a atributelor de tip e-mail. Redenumește atributele echivalente la *email*, dacă este necesar
2. Creează clasa publică *User*
3. Asignează clasa *User* ca și superclasă pentru *Player*, *LeagueOwner* și *Advertiser*
4. Adaugă câmpul protected *email* clasei *User*
5. Șterge câmpul *email* din clasele *Player*, *LeagueOwner* și *Advertiser*
6. Compilează și testează

## Before refactoring

```
public class Player {  
    private String email;  
    //...  
}  
public class LeagueOwner {  
    private String eMail;  
    //...  
}  
public class Advertiser {  
    private String email_address;  
    //...  
}
```

## After refactoring

```
public class User {  
    protected String email;  
}  
public class Player extends User {  
    //...  
}  
public class LeagueOwner extends User {  
    //...  
}  
public class Advertiser extends User {  
    //...  
}
```

# Pașii refactorizării *Pull Up Constructor Body*

1. Adaugă clasei *User* constructorul *User(String email)*
2. În constructor, asignează câmpului *email* valoarea transmisă ca și parametru
3. Înlocuiește corpul constructorului clasei *Player* cu apelul *super(email)*
4. Compilează și testează
5. Repetă pașii 1-4 pentru *LeagueOwner* și *Advertiser*

Before refactoring	After refactoring
<pre>public class User {     private String email; }  public class Player extends User {     public Player(String email) {         this.email = email;         //...     } } public class LeagueOwner extends User {     public LeagueOwner(String email) {         this.email = email;         //...     } } public class Advertiser extends User {     public Advertiser(String email) {         this.email = email;         //...     } }</pre>	<pre>public class User {     public User(String email) {         this.email = email;     } } public class Player extends User {     public Player(String email) {         super(email);         //...     } } public class LeagueOwner extends User {     public LeagueOwner(String email) {         super(email);         //...     } } public class Advertiser extends User {     public Advertiser(String email) {         super(email);         //...     } }</pre>

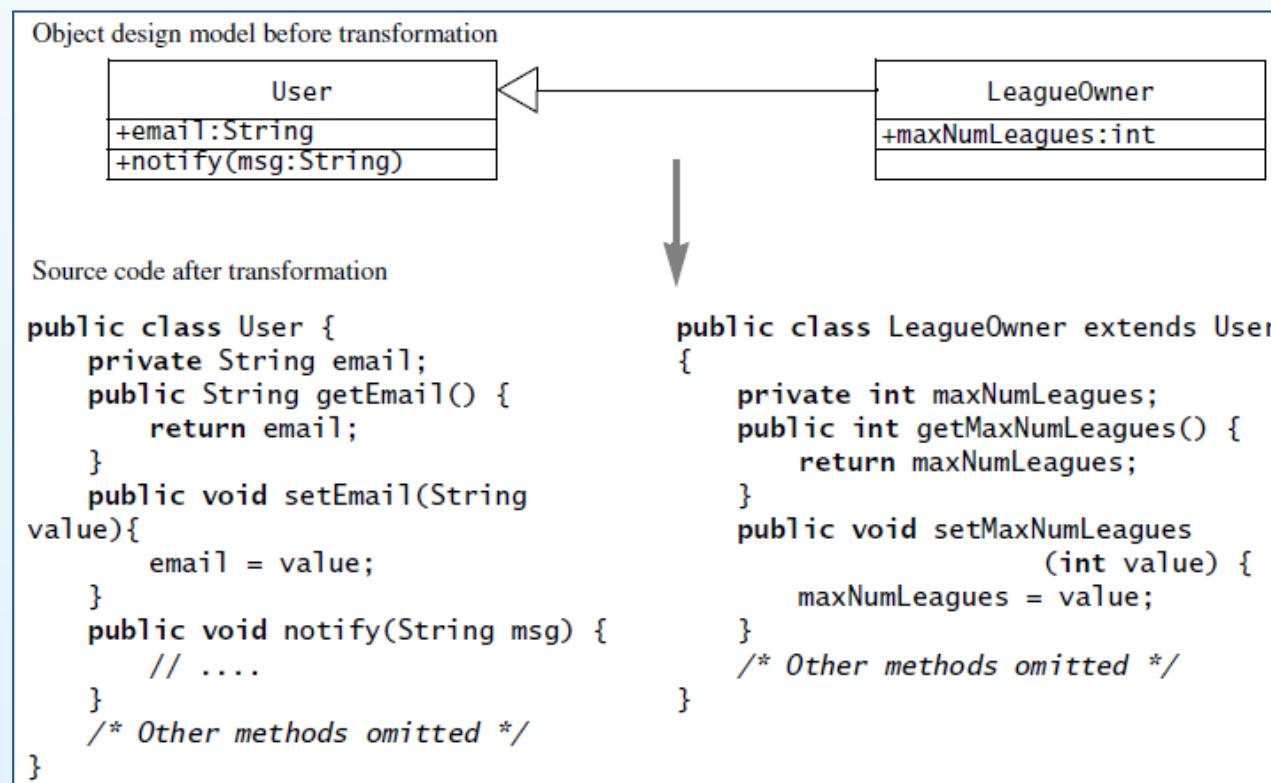
## Pașii refactorizării *Pull Up Method*

---

1. Examinează metodele din *Player* care utilizează câmpul *email*. Presupunem că *Player.notify()* utilizează acest câmp, însă nu folosește nici un alt câmp și nici o altă operație specifică lui *Player*
2. Copiază metoda *notify()* în clasa *User* și recompilează
3. Șterge metoda *Player.notify()*
4. Compilează și testează
5. Repetă pașii 1-4 pentru *LeagueOwner* și *Advertiser*

# Inginerie directă

- *Ingineria directă* se aplică unei mulțimi de elemente din model și rezultă într-o mulțime de instrucțiuni într-un limbaj de programare (cod sursă)
  - Scopul ingineriei directe este acela de a întreține o corespondență între modelul obiectual de proiectare și cod și de a reduce numărul de erori introduse la implementare (diminuând astfel efortul de implementare)



# Inginerie inversă

- *Ingineria inversă* se aplică unei mulțimi de elemente din codul sursă, rezultând într-o mulțime de elemente de model
  - Scopul ingineriei inverse este acela de a recrea modelul aferent unui sistem, ca urmare a inexistenței/pierderii sale sau a lipsei de sincronizare a acestuia cu codul sursă
  - Este transformarea opusă ingineriei directe (creează o clasă UML pentru fiecare declarație de clasă din codul sursă, adaugă un atribut pentru fiecare câmp al clasei, o operăție pentru fiecare metodă)
  - Dat fiind că, prin inginerie directă, se pierde informație din model (ex. asocierile sunt convertite în referințe sau colecții de referințe), ingineria inversă nu va produce, de regulă, același model
  - Majoritatea instrumentelor CASE existente cu suport integrat pentru ingineria inversă oferă cel mult o aproximare care permite dezvoltatorului reconstituirea modelului inițial

# Principii de transformare

- 1. Fiecare transformare trebuie să vizeze optimizări din perspectiva unui singur criteriu
  - Ex.: o aceeași transformare nu poate avea drept scop diminuarea timpului de răspuns al sistemului și creșterea inteligenției codului
  - Încercarea de a adresa mai multe criterii printr-o aceeași transformare crește complexitatea transformării și oferă condiții pentru introducerea unor erori
- 2. Fiecare transformare trebuie să fie locală
  - O transformare trebuie să afecteze doar un număr mic de metode/clase la un moment dat
  - O modificare la nivelul implementării unei metode nu va afecta clienții acesteia
  - Dacă transformarea vizează o interfață, clienții trebuie modificați pe rând
- 3. Fiecare transformare trebuie aplicată izolat de alte schimbări
  - Ex.: Adăugarea unei noi funcționalități și optimizarea codului existent nu se vor opera simultan

## Principii de transformare (cont.)

---

- 4. Fiecare transformare trebuie urmată de validări aferente
  - O transformare care operează doar asupra modelului trebuie urmată de modificarea diagramelor de interacțiune afectate de schimbarea de model efectuată și de revizuirea cazurilor de utilizare aferente, pentru a certifica oferirea funcționalității dorite
  - O refactorizare trebuie urmată de execuția cazurilor de test aferente claselor afectate de schimbările efectuate
  - Introducerea unor noi funcționalități trebuie urmată de proiectarea unor cazuri de test aferente

# Optimizarea modelului obiectual de proiectare

---

- Are drept scop îndeplinirea criteriilor de performanță ale sistemului (legate de timp de răspuns/execuție sau spațiu de memorare)
- Tipuri comune de optimizări
  - Optimizarea căilor de acces
  - Transformarea unor clase în attribute
  - Amânarea operațiilor costisitoare
  - Memorarea (eng. *caching*) rezultatelor operațiilor costisitoare
- Trebuie menținut un echilibru între eficiență și claritate, întrucât transformările care vizează eficientizarea codului, au, de obicei, efecte negative asupra intelibilității sistemului

# Optimizarea căilor de acces

- Surse comune de ineficiență la nivelul unui model obiectual
  - Traversarea repetată a unui număr mare de asocieri
  - Traversarea asocierilor cu multiplicitate *many*
  - Plasarea eronată a unor attribute
- Rezolvarea acestor probleme conduce la un model cu asocieri redundante intenționate, un număr mai mic de relații cu multiplicități *many* și un număr mai mic de clase
- *Traversarea repetată a unui număr mare de asocieri*
  - Operațiile care trebuie executate frecvent și presupun traversarea unui număr mare de asocieri introduc probleme de eficiență
  - Identificarea acestora se realizează urmărind diagramele de interacțiune aferente cazurilor de utilizare
  - Soluția: introducerea unor asocieri directe, redundante, între entitățile interogate și cele care interoghează
  - De cele mai multe ori, aceste transformări se aplică doar în urma testării sistemului, după confirmarea, la execuție, a problemelor de eficiență anticipate

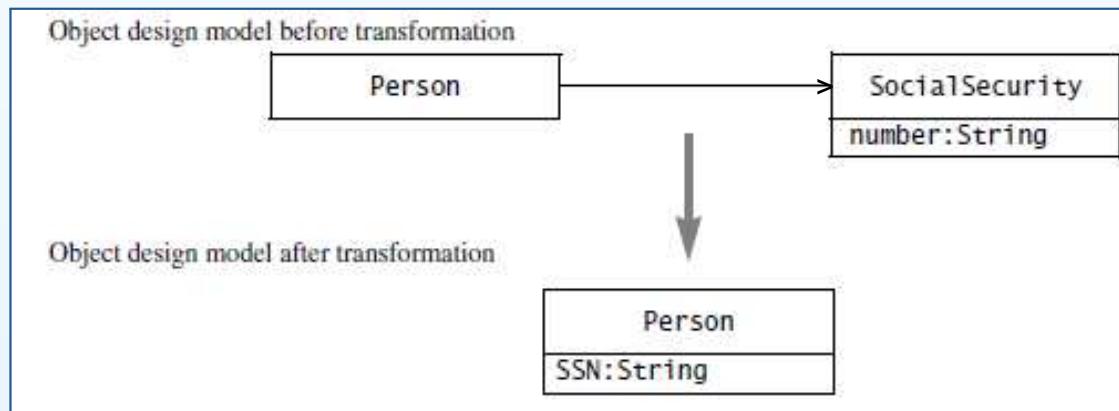
## Optimizarea căilor de acces (cont.)

---

- *Traversarea asocierilor cu multiplicitate many*
  - Soluții: calificarea asocierilor în scopul reducerii multiplicității; ordonarea sau indexarea obiectelor de la capătul aferent multiplicității *many*
- *Plasarea greșită a unor attribute*
  - Apare ca și rezultat al modelării excesive/exagerate în etapa de analiză
  - Soluție: attribute ale unor clase fără comportament relevant (doar metode get/set) pot fi relocate în clasa apelantă
  - Astfel de relocări pot conduce la eliminare din model a unor clase
- **ToDo :)** Imagineați-vă câte o situație de fiecare dintre cele trei tipuri enumerate și soluția de modelare aferentă (model inițial vs. model după transformare). Pentru obținerea unui bonus la curs, trimiteți răspunsul pe adresa [vldi@cs.ubbcluj.ro](mailto:vldi@cs.ubbcluj.ro) până la finalul zilei în care a fost postat materialul de curs pe pagină.

# Transformarea unor clase în attribute

- După restructurări/optimizări repetitive ale modelului obiectual, unele clase vor rămâne cu un număr mic de attribute/operații
- Astfel de clase, atunci când sunt asociate cu o singură altă clasă, pot fi contopite cu aceasta, reducând astfel complexitatea modelului
- Ex.:



- Clasa *SocialSecurity* nu are comportament propriu netrivial și nici asocieri cu alte clase, exceptând *Person*
- Astfel de transformări trebuie amânate până spre finalul fazei de proiectare/începutul fazei de implementare, atunci când responsabilitățile claselor sunt clare

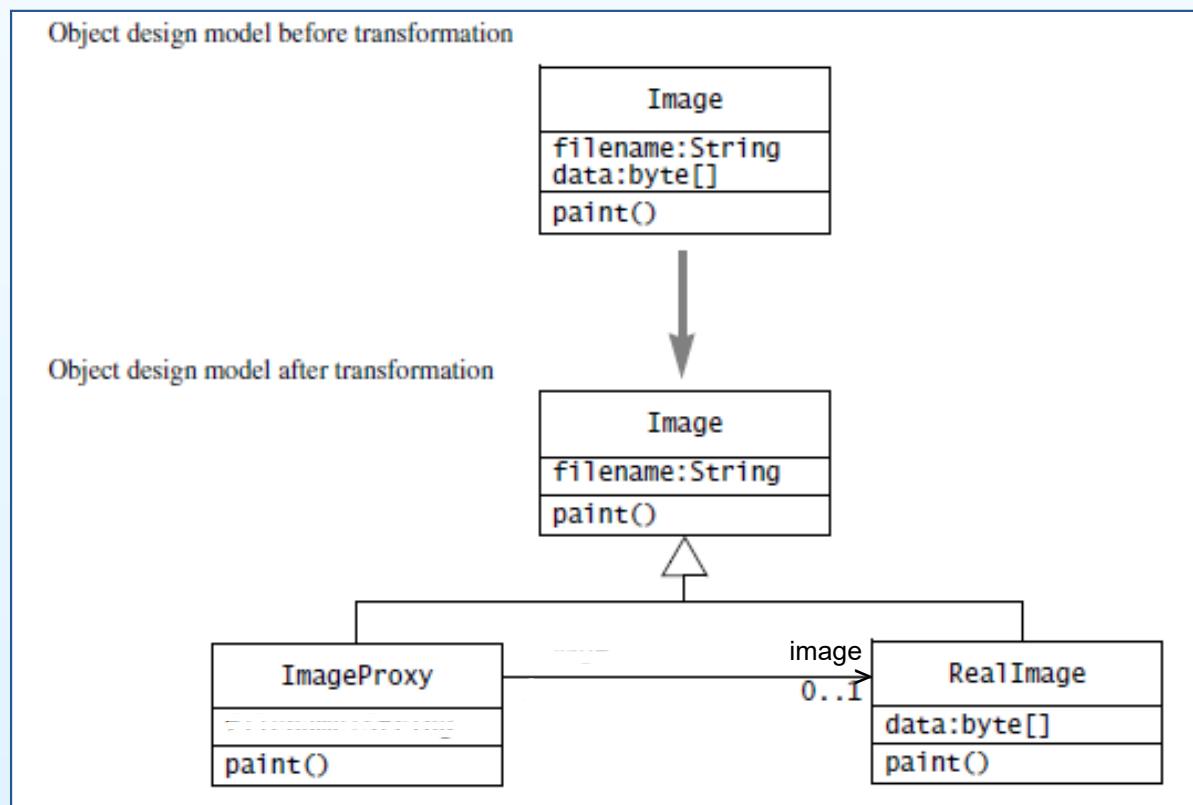
## Transformarea unor clase în attribute (cont.)

---

- Acestei transformări îi corespunde unu caz particular al refactorizării *Inline Class* [Fowler, 2000]
- Pași refactorizării *Inline Class*
  - Declară toate câmpurile și metodele publice ale clasei sursă în clasa destinație
  - Schimbă toate referințele spre clasa sursă către clasa destinație
  - Schimbă numele clasei sursă, pentru a identifica eventualele referințe nemodificate
  - Compilează și testează
  - Sterge clasa sursă

# Gestionare operațiilor costisitoare

- Operațiile costisitoare, de tipul încărcării unor obiecte grafice, pot fi amânate până în momentul în care este necesară vizualizarea acestora
  - Soluție: aplicarea şablonului *Proxy*
  - Ex.:



## Gestionare operațiilor costisitoare (cont.)

- Un obiect de tip *ImageProxy* (imagină surogat) ia locul obiectului *ReallImage* (imagină reală), oferind aceeași interfață cu acesta
- Obiectul surogat răspunde la solicitări simple și încarcă obiectul real doar în momentul în care i se apelează operația `paint()` (mesajul de desenare va fi apoi delegat obiectului imagine real)
- În cazul în care clienții nu apelează `paint()`, obiectul imagine real nu va fi creat niciodată
- Rezultatele unor operații complexe, apelate frecvent, însă bazate pe valori care nu se schimbă sau se schimbă destul de rar, pot fi memorate la nivelul unor atrbute private
  - Soluția optimizează timpul de răspuns la apelul operațiilor, însă consumă spațiu de memorie suplimentar pentru stocarea unor informații redundante

# Reprezentarea asocierilor

- UML: *asocieri* = mulțimi de legături între obiecte
- Limbaje de programare: *referințe* / *colecții de referințe*
- Implementarea asocierilor în cod ține cont de *navigabilitate*, *multiplicități*, *nume de roluri* și de semantica domeniului
  - Bidirectionalitatea introduce dependențe mutuale între clase (se traduce prin perechi de referințe ce trebuie sincronizate)
  - multiplicitatea *one* necesită o referință, cea *many* - o colecție de referințe
  - numele de roluri corespund numelor de câmpuri adăugate în clase

# Asocieri unidirectionale one-to-one

- Model



- Cod sursă după transformare

```
public class Advertiser {

    private Account account;

    public Advertiser()
    {
        account = new Account();
    }

    public Account getAccount()
    {
        return account;
    }

    // nu ofera setter
}
```

# Asocieri bidirectionale one-to-one

- Model



- Cod sursă după transformare

```
public class Advertiser {  
  
    /* The account field is initialized  
     * in the constructor  
     * and never modified */  
    private Account account;  
  
    public Advertiser()  
    {  
        account = new Account(this);  
    }  
  
    public Account getAccount()  
    {  
        return account;  
    }  
}
```

```
public class Account {  
  
    /* The owner field is initialized  
     * in the constructor  
     * and never modified. */  
    private Advertiser owner;  
  
    public Account(Advertiser owner)  
    {  
        this.owner = owner;  
    }  
  
    public Advertiser getOwner()  
    {  
        return owner;  
    }  
}
```

# Asocieri bidirectionale one-to-many

- Model



- Cod sursă după transformare

```
public class Advertiser {

    private Set<Account> accounts;

    public Advertiser()
    {
        accounts = new HashSet();
    }

    public void addAccount(Account account)
    {
        if(!accounts.contains(account))
        {
            accounts.add(account);
            account.internalSetOwner(this);
        }
    }
}
```

```
public class Account {

    private Advertiser owner;

    public void setOwner(Advertiser owner)
    {
        Advertiser oldOwner = this.owner;
        Advertiser newOwner = owner;
        if(oldOwner != null)
            oldOwner.internalRemoveAccount(this);
        if(newOwner != null)
            newOwner.internalAddAccount(this);
        this.owner = newOwner;
    }
}
```

## Asocieri bidirectionale one-to-many (cont.)

```
public void removeAccount(Account account)
{
    if(accounts.contains(account))
    {
        accounts.remove(account);
        account.internalSetOwner(null);
    }
}

void internalAddAccount(Account account)
{
    if(!accounts.contains(account))
        accounts.add(account);
}

void internalRemoveAccount(Account account)
{
    if(accounts.contains(account))
        accounts.remove(account);
}

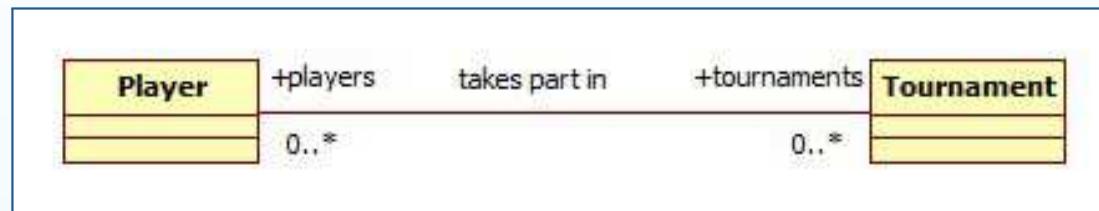
public Set<Account> getAccounts()
{
    return Collections.unmodifiableSet(accounts);
}
```

```
void internalSetOwner(Advertiser owner)
{
    this.owner = owner;
}

public Advertiser getOwner()
{
    return owner;
}
```

# Asocieri bidirectionale *many-to-many*

- Model



- Cod sursă după transformare

```
public class Player {  
  
    private Set<Tournament> tournaments;  
  
    public Player()  
    {  
        tournaments = new HashSet();  
    }  
  
    public void addTournament(Tournament tournament)  
    {  
        //pre: tournament != null  
        if(!tournaments.contains(tournament))  
        {  
            tournaments.add(tournament);  
            tournament.internalAddPlayer(this);  
        }  
    }  
}
```

```
public class Tournament {  
  
    private Set<Player> players;  
  
    public Tournament()  
    {  
        players = new HashSet();  
    }  
  
    public void addPlayer(Player player)  
    {  
        //pre: player != null  
        if(!players.contains(player))  
        {  
            players.add(player);  
            player.internalAddTournament(this);  
        }  
    }  
}
```

## Asocieri bidirectionale *many-to-many* (cont.)

```
public void removeTournament(Tournament tournament)
{
    //pre: tournament != null
    if(tournaments.contains(tournament))
    {
        tournaments.remove(tournament);
        tournament.internalRemovePlayer(this);
    }
}

void internalAddTournament(Tournament tournament)
{
    //pre: tournament != null
    if(!tournaments.contains(tournament))
        tournaments.add(tournament);
}

void internalRemoveTournament(Tournament tournament)
{
    //pre: tournament != null
    if(tournaments.contains(tournament))
        tournaments.remove(tournament);
}

public Set<Tournament> getTournaments()
{
    return Collections.unmodifiableSet(tournaments);
}
```

```
public void removePlayer(Player player)
{
    //pre: tournament != null
    if(players.contains(player))
    {
        players.remove(player);
        player.internalRemoveTournament(this);
    }
}

void internalAddPlayer(Player player)
{
    //pre: tournament != null
    if(!players.contains(player))
        players.add(player);
}

void internalRemovePlayer(Player player)
{
    //pre: tournament != null
    if(players.contains(player))
        players.remove(player);
}

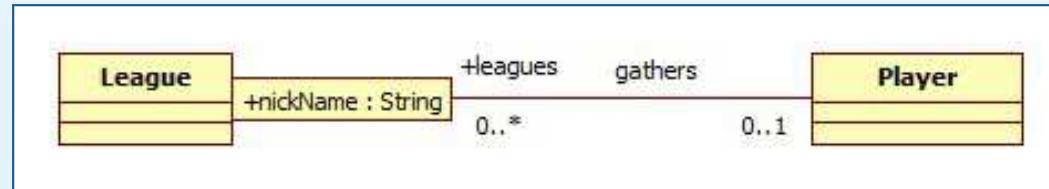
public Set<Player> getPlayers()
{
    return Collections.unmodifiableSet(players);
}
```

# Asocieri calificate

- Asocierile calificate sunt utilizate pentru a "reduce" multiplicitatea unui capăt *many* din cadrul unei asocieri *one-to-many* sau *many-to-many*
  - Calificatorul asocierii este un atribut al clasei din capătul *many* care se dorește a fi redus, atribut care are valori unice în contextul asocierii, însă nu neapărat unice la nivel global
  - Ex.: Pentru a putea fi ușor identificați în cadrul unei ligi, jucătorii își pot alege un *nickName* care trebuie să fie unic în cadrul ligii (jucătorii pot avea *nickName*-uri diferite în ligi diferite, iar fiecare astfel de *nickName* nu trebuie să fie unic la nivel global)
  - Modelul înainte de transformare



- Modelul după transformare



## Asocieri calificate (cont.)

```
public class League {

    private Map<String, Player> players;

    public League()
    {
        players = new HashMap();
    }

    public void addPlayer(String nickName, Player player)
    {
        if(!players.containsKey(nickName))
        {
            players.put(nickName, player);
            player.internalAddLeague(this);
        }
    }

    public Player getPlayer(String nickName)
    {
        return players.get(nickName);
    }

    void internalAddPlayer(String nickName, Player player)
    {
        if(!players.containsKey(nickName))
            players.put(nickName, player);
    }
}
```

## Asocieri calificate (cont.)

```
public class Player {

    private Set<League> leagues;

    public Player()
    {
        leagues = new HashSet();
    }

    public void addLeague(League league, String nickname)
    {
        if(league.getPlayer(nickname) == null)
        {
            leagues.add(league);
            league.internalAddPlayer(nickname, this);
        }
    }

    void internalAddLeague(League league)
    {
        leagues.add(league);
    }

}
```

# Reprezentarea contractelor

- *Verificarea precondițiilor*
  - Precondițiile trebuie verificate la începutul fiecărei metode, înaintea efectuării procesărilor caracteristice. În cazul în care precondiția nu este adevărată, se va arunca o excepție. Se recomandă ca fiecare precondiție să corespundă unui tip particular de excepție.
- *Verificarea postcondițiilor*
  - Postcondițiile trebuie verificate la sfârșitul fiecărei metode, după terminarea tuturor procesărilor caracteristice și finalizarea schimbărilor de stare. În cazul în care contractul este violat, se va arunca o excepție specifică.
- *Verificarea invariantei*
  - Invariantele se vor verifica odată cu postcondițiile (la finalizarea fiecărei metode publice a clasei)
- *Moștenirea contractelor*
  - Codul de verificare al aserțiunilor trebuie încapsulat la nivelul unor metode specifice, pentru a permite apelarea acestora din subclase

## Ex.: contract OCL



```
context Tournament
    inv maxNumPlayersPositive:
        self.getMaxNumPlayers() > 0

context Tournament::acceptPlayer(p: Player)
    pre: self.getNumPlayers() < self.getMaxNumPlayers() and
        not self.isPlayerAccepted(p)
    post: self.isPlayerAccepted(p) and
        self.getNumPlayers() = self@pre.getNumPlayers() + 1
```

## Ex.: implementarea contractului

```
public class Tournament {  
    //...  
    private List players;  
  
    public void acceptPlayer(Player p)  
        throws KnownPlayer, TooManyPlayers, UnknownPlayer,  
               IllegalNumPlayers, IllegalMaxNumPlayers  
    {  
        // check precondition !isPlayerAccepted(p)  
        if (isPlayerAccepted(p)) {  
            throw new KnownPlayer(p);  
        }  
        // check precondition getNumPlayers() < maxNumPlayers  
        if (getNumPlayers() == getMaxNumPlayers()) {  
            throw new TooManyPlayers(getNumPlayers());  
        }  
        // save values for postconditions  
        int pre_getNumPlayers = getNumPlayers();
```

## Ex.: implementarea contractului (cont.)

```
// accomplish the real work
players.add(p);
p.addTournament(this);

// check post condition isPlayerAccepted(p)
if (!isPlayerAccepted(p)) {
    throw new UnknownPlayer(p);
}
// check post condition getNumPlayers() = @pre.getNumPlayers() + 1
if (getNumPlayers() != pre_getNumPlayers + 1) {
    throw new IllegalNumPlayers(getNumPlayers());
}
// check invariant maxNumPlayers > 0
if (getMaxNumPlayers() <= 0) {
    throw new IllegalMaxNumPlayers(getMaxNumPlayers());
}
}
//...
```

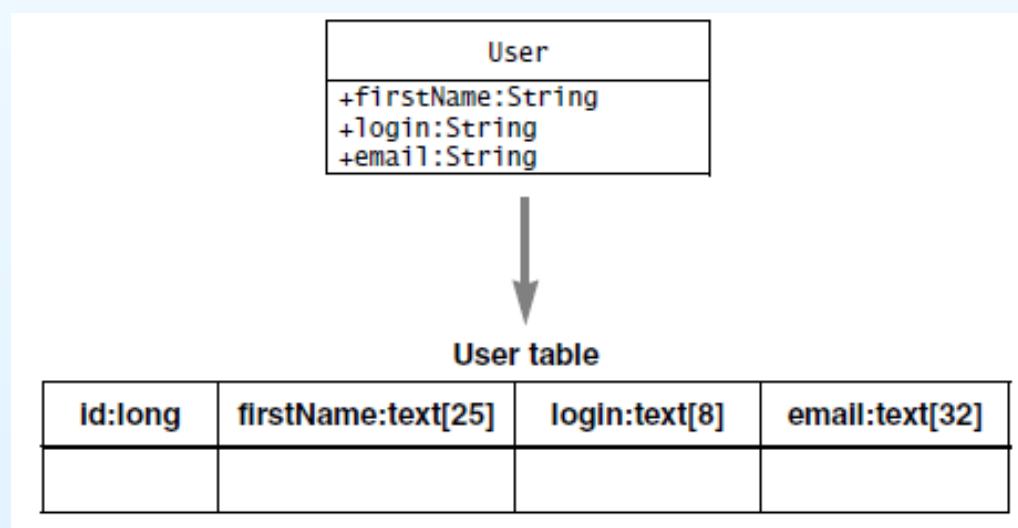
## Reprezentarea contractelor (cont.)

- Dezavantaje ale unei implementări/monitorizări manuale exhaustive a indeplinirii contractelor
  - *Efortul de codificare* - cod de verificare uneori mai complex decât logica operației în sine
  - *Șanse mari de introducere a unor erori*
  - *Possibilitatea de mascare a unor defecte în codul aferent funcționalității* - în cazul în care cele două sunt scrise de către același programator
  - *Dificultatea modificării codului în cazul modificării constrângerii*
  - *Probleme de performanță la monitorizarea exhaustivă*
- *Soluții*
  - Generarea automată a codului de verificare aferent contractelor folosind instrumente CASE dedicate (ex. OCLE)
  - Monitorizarea selectivă
    - la testare - toate aserțiunile
    - la exploatare - selectiv, funcție de performanțele dorite, gradul de încredere în calitatea codului și natura critică a aplicației

# Reprezentarea entităților persistente

- *Reprezentarea claselor și atributelor*

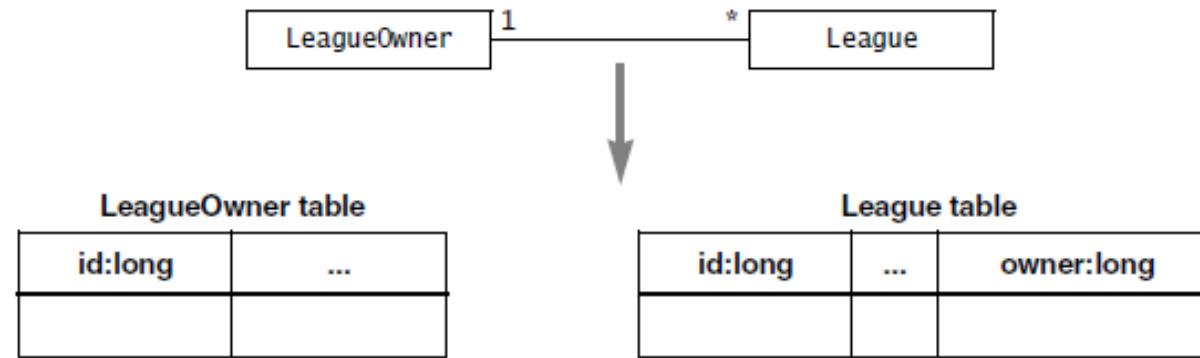
- Fiecare clasă se reprezintă folosind un tabel cu același nume
- Pentru fiecare atribut al clasei se adaugă în tabel o coloană cu același nume
- Fiecare linie a unui tabel va corespunde unei instanțe a clasei
- În mod ideal, cheia primară ar trebui să fie un identificator unic (eventual autoincrement), diferit de atributele proprii ale clasei în cauză. Alegerea că și cheia a unui atribut (grup) caracteristice tipului de entitate este problematică în momentul în care apar modificări la nivelul domeniului aplicației



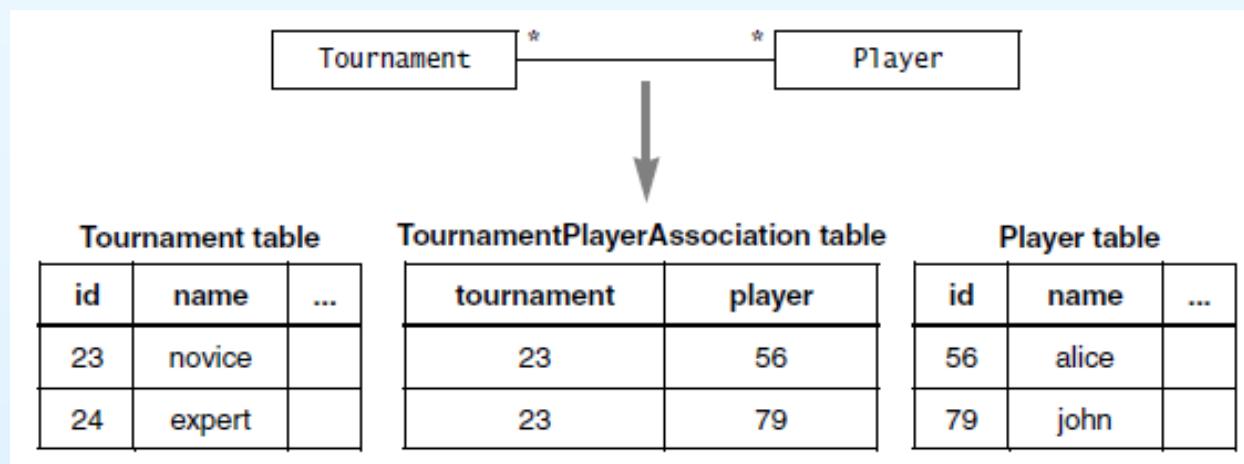
# Reprezentarea entităților persistente (cont.)

- Reprezentarea asocierilor

- Asocierile *one-to-one* și *one-to-many* se reprezintă folosind chei străine



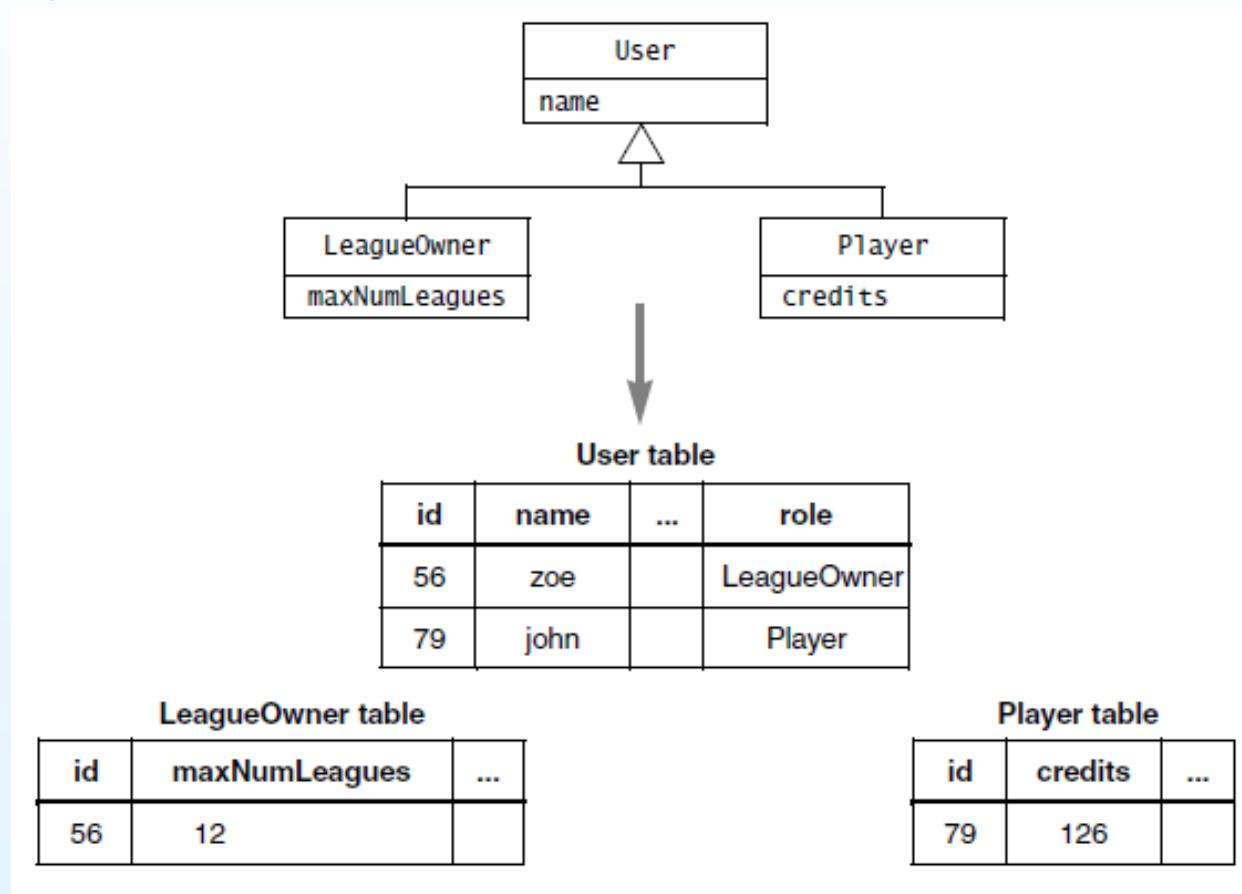
- Asocierile *many-to-many* se reprezintă folosind tabele de legătură



# Reprezentarea entităților persistente (cont.)

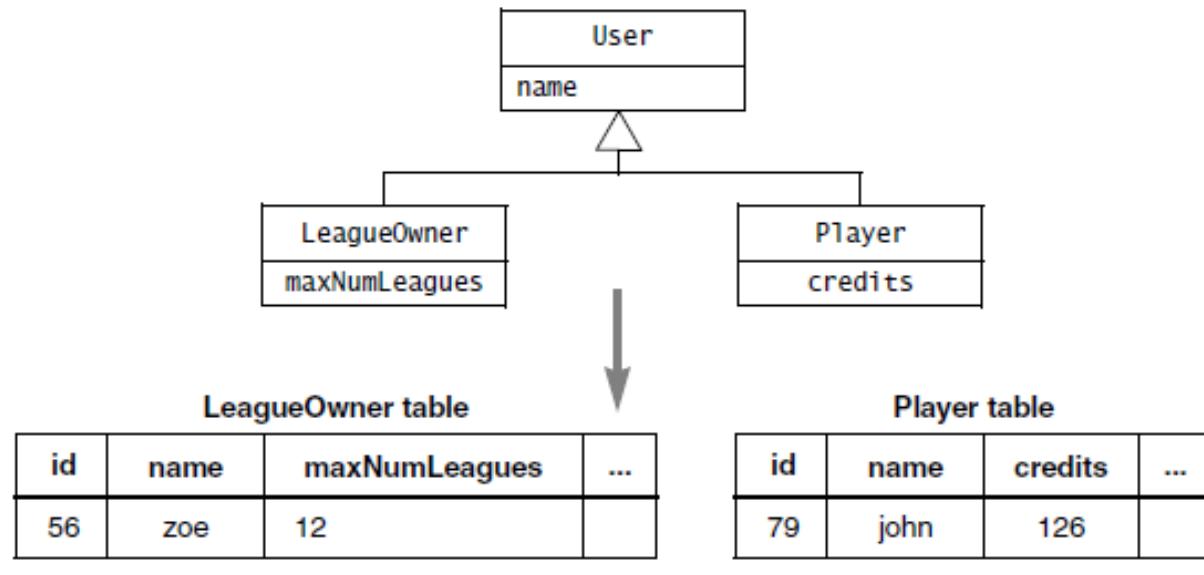
- Reprezentarea moștenirii

- Mapare verticală



## Reprezentarea entităților persistente (cont.)

- Mapare orizontală



- Mapare verticală vs. mapare orizontală = modificabilitate vs. performanță
  - Maparea verticală: adăugarea unui atribut în clasa de bază => adăugarea unei coloane în tabelul aferent; adăugarea unei noi clase derivate => definirea unui tabel cu atributele proprii ale acesteia; fragmentarea obiectelor individuale => interogări mai lente

## Reprezentarea entităților persistente (cont.)

- Maparea orizontală: adăugarea unui atribut în clasa de bază => adăugarea unei coloane în fiecare dintre tabelele aferente claselor derivate; adăugarea unei noi clase derivate => definirea unui tabel cu attributele proprii + cele moștenite; nefragmentarea obiectelor individuale => interogări rapide

## Referinte

---

- [Fowler, 2000] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Weslwy Reading, MA, 2000.

## *Curs 11*

### *Testarea sistemelor soft*

*Suport de curs bazat pe B. Bruegge and A.H. Dutoit*

*"Object-Oriented Software Engineering using UML, Patterns, and Java"*

# Testarea sistemelor soft

- *Testarea* = procesul de identificare a diferențelor dintre comportamentul dorit/așteptat al sistemului (specificat cu ajutorul modelelor) și comportamentul observat al acestuia
  - *Testarea unitară* - identifică diferențe dintre specificarea unui obiect și implementarea acestuia ca și componentă
  - *Testarea structurală* - identifică diferențe dintre modelul aferent proiectării de sistem și comportamentul unei grup de subsisteme integrate
  - *Testarea funcțională* - identifică diferențe dintre modelul cazurilor de utilizare și sistem
  - *Testarea performanței* - identifică diferențe dintre cerințele nefuncționale și performanțele sistemului
- Din perspectiva modelării, testarea reprezintă o încercare de a demonstra că implementarea sistemului este inconsistentă cu modelele acestuia
  - Scopul este proiectarea unor teste care să pună în evidență defectele sistemului

# Fiabilitatea sistemelor soft

- *Corectitudinea* unui sistem în raport cu specificația - vizează concordanța dintre comportamentul observat al sistemului și specificarea acestuia.
  - *Fiabilitatea softului* = probabilitatea ca acel soft să nu cauzeze eșecul sistemului, pentru o anumită perioadă de timp și în condiții specificate [IEEE Std. 982.2-1988]
- *Eșec* (eng. *failure*) = orice deviere a comportamentului observat de la cel specificat/așteptat
- *Eroare/stare de eroare* (eng. *erroneous state*) = orice stare a sistemului în care procesările ulterioare ar conduce la eșec
- *Defect* (eng. *fault/defect/bug*) = cauza mecanică sau algoritmică a unei stări de eroare
- *Testare* = încercarea sistematică și planificată de a găsi defecte în softul implementat
  - "Testing can only show the presence of bugs, not their absence." (E. Dijkstra)

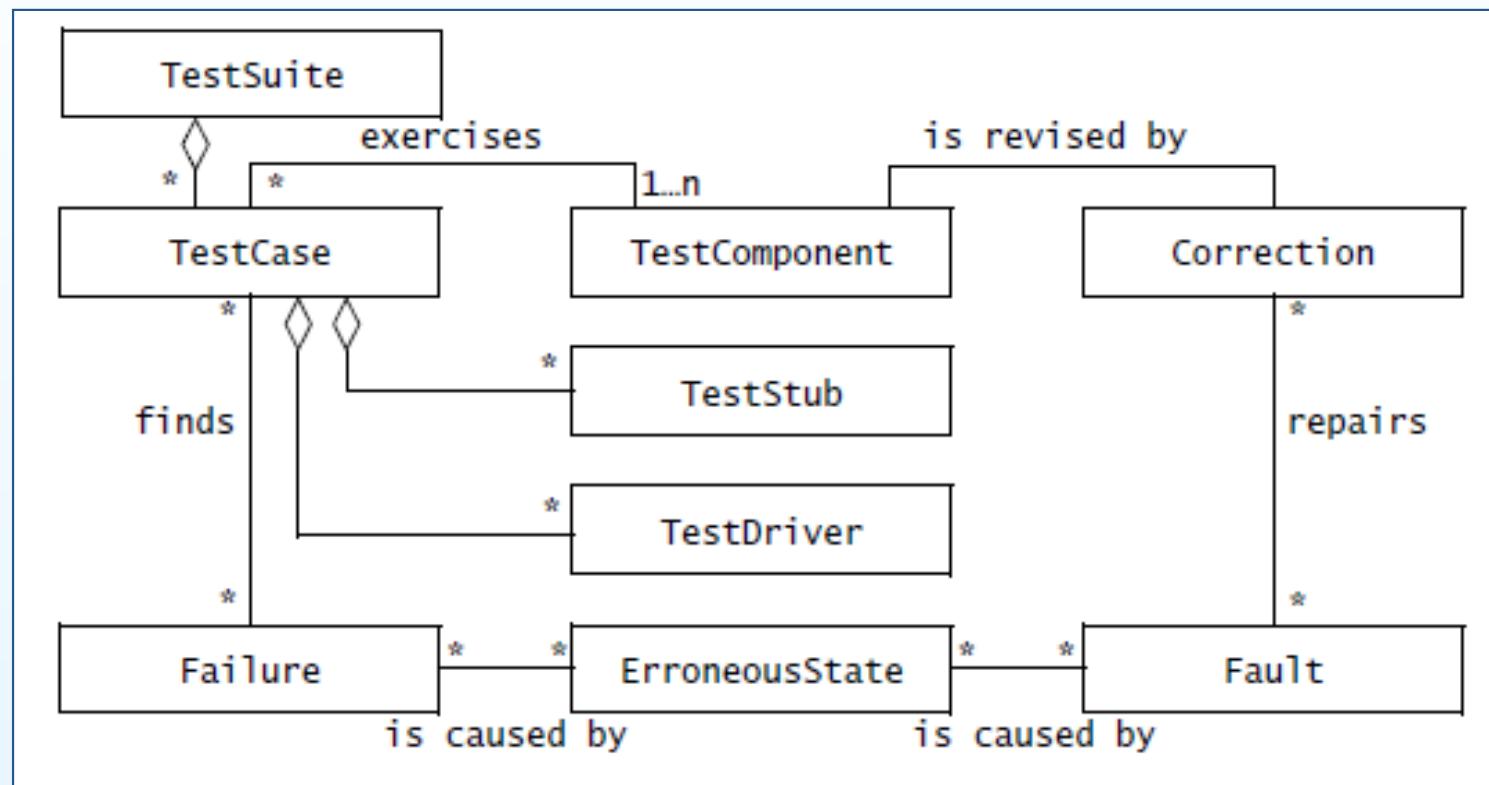
## Fiabilitatea sistemelor soft (cont.)

- Tehnici de creștere a fiabilității sistemelor soft
  - *Tehnici privind evitarea defectelor* (eng. *fault avoidance techniques*)
    - identifică posibilele defecte la nivel static (fără o execuție a modelelor/codului) și încearcă să prevină introducerea acestora în sistem
    - ex.: metodologii formale - Cleanroom, Correctness by Construction
  - *Tehnici privind detectarea defectelor* (eng. *fault detection techniques*)
    - sunt utilizate în timpul procesului de dezvoltare, pentru a identifica stările de eroare și defectele care le-au provocat, anterior livrării sistemului (ex. review, testare, depanare)
    - nu își propun recuperarea sistemului din stările de eșec induse de defectele identificate
  - *Tehnici privind tolerarea defectelor* (eng. *fault tolerance techniques*)
    - pornesc de la premisa că sistemul poate fi livrat cu defecte și că eventualele eșecuri pot fi gestionate prin recuperare în urma lor la execuție
    - ex.: sistemele redundante utilizează mai multe calculatoare și softuri diferite pentru realizarea acelorași sarcini

# Testarea sistemelor soft - concepte

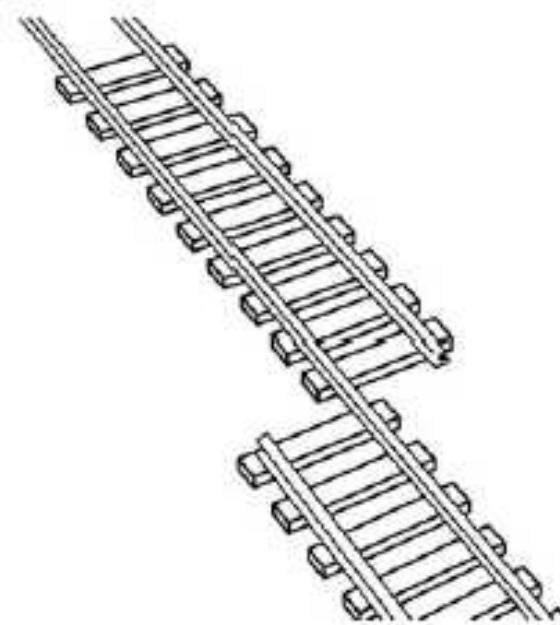
- *Componentă de test* (eng. *test component*) = parte a unui sistem care poate fi izolată pentru testare (obiect/subsistem, grup de obiecte/subsisteme)
- *Defect* (eng. *fault, bug, defect*) = greșală de proiectare sau codificare ce poate determina un comportament anormal al unei componente
- *Stare de eroare* (eng. *error state*) = manifestare a unui defect în timpul execuției sistemului
- *Eșec* (eng. *failure*) = deviere a comportamentului observat al componentei de la cel specificat
- *Caz de test* (eng. *test case*) = o mulțime de date de intrare și rezultate așteptate, proiectate cu intenția de a provoca eșecul sistemului și a descoperi defecte la nivelul componentelor
- eng. *Test stub* = implementare parțială a unei componente, de care depinde componenta de test
- eng. *Test driver* = implementare parțială a unei componente care depinde de componenta de test (împreună cu stub-urile, permit izolarea unei componente pentru testare)
- *Corectură* (eng. *Correction*) = modificare a unei componente, în scopul de a remedia un defect (poate introduce noi defecte)

## Testarea sistemelor soft - concepte (cont.)



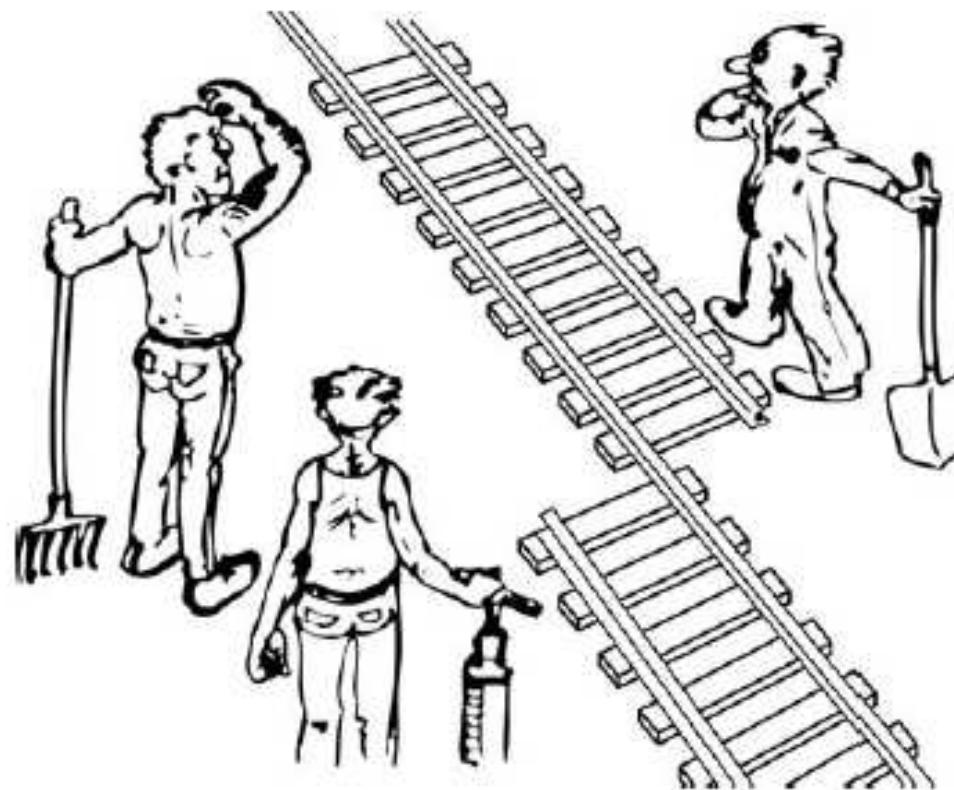
## Testarea sistemelor soft - concepte (cont.)

- Eșec, eroare sau defect?



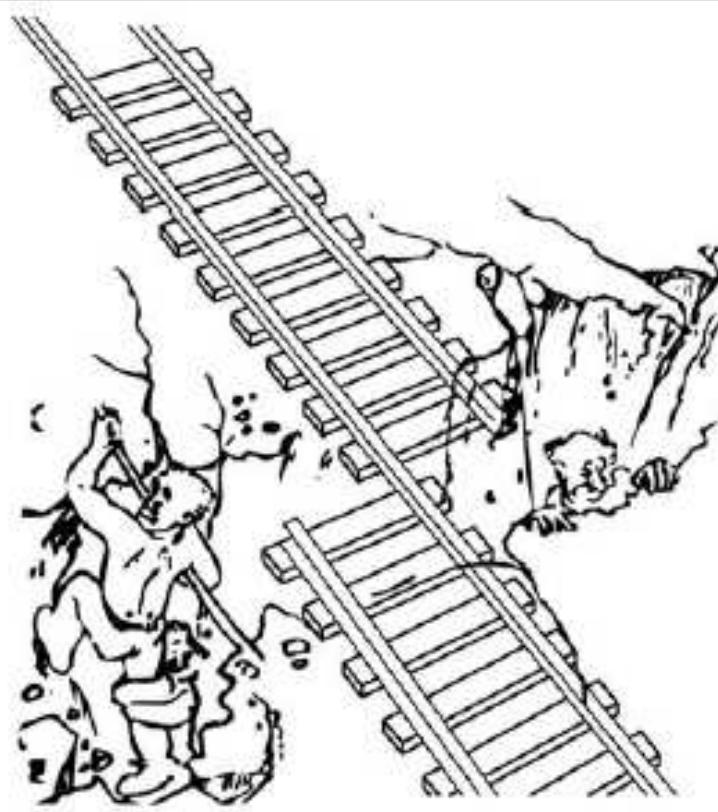
## Testarea sistemelor soft - concepte (cont.)

- Defect algoritmic
  - omiterea inițializării unei variabile
  - setarea unei variabile folosite drept index în afara valorilor permise



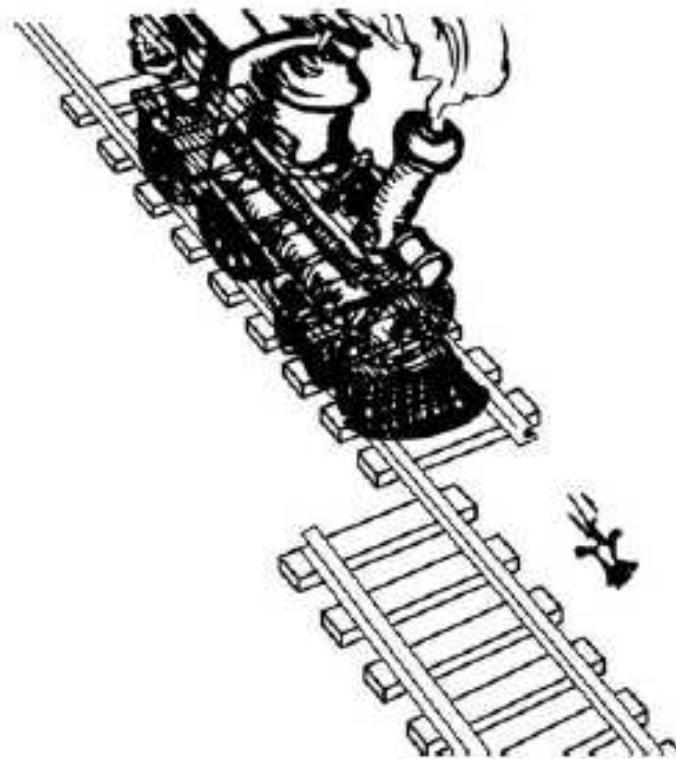
## Testarea sistemelor soft - concepte (cont.)

- Defect mecanic
  - defect la nivelul mașinii virtuale
  - pană de curent



## Testarea sistemelor soft - concepte (cont.)

- Eroare



# Testarea sistemelor soft - activități

- *Planificarea testării* (eng. *test planning*)
  - alocă resurse și programează activitățile de testare
  - această activitate ar trebui să aibă loc devreme în procesul de dezvoltare, astfel încât testării să i se aloce suficient timp și resursă umană calificată (ex. cazurile de test ar trebui proiectate imediat ce modelele aferente devin stabile)
- *Inspectarea componentelor* (eng. *component inspection*)
  - Identifică defecte la nivelul componentelor prin inspectarea manuală a codului sursă al acestora
- *Testarea utilizabilității* (eng. *usability testing*)
  - încearcă să detecteze defecte în proiectarea interfeței utilizator a sistemului
  - uneori, sistemele eșuează din cauză că utilizatorii sunt induși în eroare de interfața grafică și introduc neintenționat date eronate
- *Testarea unitară* (eng. *unit testing*)
  - încearcă să detecteze defecte la nivelul obiectelor sau subsistemelor individuale, raportându-se la specificarea acestora (modelul aferent proiectării obiectuale)

# Testarea sistemelor soft - activități (cont.)

---

- *Testarea de integrare* (eng. *integration testing*)
  - încearcă să identifice defecte prin integrarea diferitor componente, raportându-se la modelul aferent proiectării de sistem
  - *testarea structurală* (eng. *structural testing*) = testare de integrare implicând toate componentele sistemului
- *Testarea de sistem* (eng. *system testing*)
  - testează sistemul integrat în ansamblu
  - *testarea funcțională* (eng. *functional testing*) - realizată de către dezvoltatori, testează sistemul în raport cu modelul său funcțional
  - *testarea performanței* (eng. *performance testing*) - realizată de către dezvoltatori, testează sistemul în raport cu specificarea cerințelor nefuncționale și cu obiectivele adiționale de proiectare
  - *testarea de acceptare* și *testarea de instalare* (eng. *acceptance testing*, *installation testing*) - realizate de către clienți în mediul de dezvoltare, respectiv de exploatare a sistemului

## Inspectarea componentelor

- Inspectarea identifică defectele unei componente prin analiza codului acesteia, în cadrul unei reuniuni formale
  - Inspecțiile pot avea loc anterior sau ulterior testării unitare
- *Metoda inspectării a lui Fagan* [Fagan, 1976] - primul proces structurat de inspectare
  - Inspecția este condusă de către o echipă de dezvoltatori, incluzând autorul componentei, un moderator și unul sau mai mulți recenzori
  - Pași
    - *Overview* - autorul componentei prezintă, pe scurt, scopul acesteia și obiectivele inspecției
    - *Pregătire* - recenzorii se familiarizează cu codul componentei (fără a se focaliza pe identificarea defectelor)
    - *Şedința de inspectare* - unul dintre cei prezenți parafrazează codul sursă al componentei și membrii echipei semnalează probleme cu privire la acesta
    - *Revizuire* - autorul revizuește codul componentei, conform observațiilor primite
    - *Urmări* - moderatorul verifică varianta revizuită și poate stabili necesitatea de reinspectare

## Inspectarea componentelor (cont.)

- Etape critice: pregătirea și ședința de inspectare
- În afara identificării defectelor, recenzorii pot semnala abateri de la standardele de codificare sau ineficiențe
- Eficiența unei inspecții depinde de pregătirea recenzorilor
- *Active Design Review* [Parnas and Weiss, 1985] - proces de inspectare îmbunătățit
  - Elimină ședința de inspectare, recenzorii identifică defecte în faza de pregătire
  - La finalul etapei de pregătire, fiecare recenzor completează un chestionar care atestă gradul de înțelegere a componenetei
  - Autorul colectează feedback asupra componentei în urma unor întâlniri individuale cu fiecare recenzor
- Ambele metode de inspectare s-au dovedit a fi mai eficiente în descoperirea defectelor decât testarea
  - În proiectele critice se recurge atât la inspecții, cât și la testare, întrucât au tendința de a identifica tipuri diferite de erori

# Testarea utilizabilității

- Identifică diferențele dintre sistem și așteptările utilizatorilor cu privire la comportamentul acestuia (spre deosebire de celelalte tipuri de testare, nu compară sistemul cu o specificație)
- Tehnica de realizare a testelor privind utilizabilitatea
  - Dezvoltatorii formulează un set de obiective descriind informația pe care se așteaptă să o obțină în urma testelor (ex.: evaluarea layout-ului interfeței grafice, evaluarea impactului pe care timpul de răspuns îl are asupra eficienței utilizatorilor, evaluarea măsurii în care documentația online răspunde nevoilor utilizatorilor)
  - Obiectivele anterioare sunt evaluate într-o serie de experimente în care reprezentați ai utilizatorilor sunt antrenați să execute anumite sarcini
  - Dezvoltatorii observă participanții și colectează date privind performanțele acestora (timp de îndeplinire a unei sarcini, rata erorilor) și preferințele lor
- Tipuri de teste de utilizabilitate
  - Teste bazate pe scenarii
  - Teste bazate pe prototipuri (verticale sau orizontale)
  - Teste pe baza sistemului real

## Testarea utilizabilității (cont.)

- Elemente fundamentale ale testelor de utilizabilitate
  - obiectivele de test
  - reprezentanți ai utilizatorilor
  - mediul de lucru, real sau simulant
  - interogare extensivă a utilizatorilor de către responsabilul cu testele de utilizabilitate
  - colectare și analiză a rezultatelor cantitative și calitative
  - recomandări cu privire la modul de îmbunătățire a asistemului
- Obiective de test uzuale
  - compararea a două stiluri de interacțiune utilizator
  - identificarea celor mai bune/rele funcționalități într-un scenariu/prototip
  - identificarea funcționalităților utilie pentru începători/experți
  - identificarea situațiilor care necesită help online, etc.

# Testarea unitară

- Se focusează pe componentele elementare ale sistemului soft - obiecte și subsisteme
  - Candidați pentru testarea unitară: toate clasele modelului obiectual și toate subsistemele identificate în proiectarea de sistem
- Avantaje
  - Reducerea complexității activităților de testare, prin focusarea pe componente cu granularitate mică
  - Ușurința identificării și corectării defectelor, ca urmare a numărului mic de componente implicate într-un test
  - Posibilitatea introducerii paralelismului în activitatea de testare (componentele pot fi testate independent și simultan)
- Tehnici de testare unitară
  - *Testarea bazată pe echivalențe* (eng. *equivalence testing*)
  - *Testarea frontierelor* (eng. *boundary testing*)
  - *Testarea căilor de execuție* (eng. *path testing*)
  - *Testarea bazată pe stări* (eng. *state-based testing*)
  - *Testarea polimorfismului* (eng. *polymorphism testing*)

# Testarea bazată pe echivalențe

- Tehnică de testare blackbox care minimizează numărul de cazuri de test, prin partaționarea intrărilor posibile în clase de echivalență și selectarea unui caz de test pentru fiecare astfel de clasă
  - Se presupune că sistemul se comportă în mod similar pentru toți membrii unei clase de echivalență => testarea comportamentului aferent unei clase de echivalență se poate realiza prin testarea unui singur membru al clasei
- Pași
  - I. Identificarea claselor de echivalență
  - II. Selectarea intrărilor pentru test
- Criterii utilizate în stabilirea claselor de echivalență
  - *Acoperire*: fiecare intrare posibilă trebuie să aparțină uneia dintre clasele de echivalență
  - *Caracter disjunct*: o aceeași intrare nu poate apartine mai multor clase de echivalență
  - *Reprezentare*: Dacă, prin utilizarea ca și intrare a unui anumit membru al unei clase de echivalență, execuția conduce la o stare de eroare, atunci aceeași stare va putea fi detectată utilizând ca și intrare orice alt membru al clasei

## Testarea bazată pe echivalențe (cont.)

- Ex.: testarea unei metode care returnează numărul de zile dintr-o lună, date fiind luna și anul (întregi)

```
class MyGregorianCalendar {  
    ...  
    public static int getNumDaysInMonth(int month, int year) {...}  
    ...  
}
```

- I.1 Identificarea claselor de echivalență pentru lună
  - clasa lunilor cu 31 de zile (1,3,5,7,8,10,12)
  - clasa lunilor cu 30 de zile (4,6,9,11)
  - clasa lunilor cu 28/29 de zile (2)
- I.2 Identificarea claselor de echivalență pentru an
  - clasa anilor bisecți
  - clasa anilor non-bisecți
- Valori invalide
  - pentru lună: < 1, > 12
  - pentru an: < 0

## Testarea bazată pe echivalențe (cont.)

- II. Selectarea reprezentanților pentru test
  - II.1 Pentru lună: 2 (February), 6 (June), 7 (July)
  - II.2 Pentru an: 1904, 1901
  - => 6 clase de echivalență prin combinație

Equivalence class	Value for month input	Value for year input
Months with 31 days, non-leap years	7 (July)	1901
Months with 31 days, leap years	7 (July)	1904
Months with 30 days, non-leap years	6 (June)	1901
Month with 30 days, leap year	6 (June)	1904
Month with 28 or 29 days, non-leap year	2 (February)	1901
Month with 28 or 29 days, leap year	2 (February)	1904

# Testarea frontierelor

- Caz particular al metodei de testare bazată pe echivalențe, focusată pe explorarea cazurilor limită
  - În loc să se aleagă un reprezentant arbitrar al clasei de echivalență pentru testare, metoda cere alegerea unui element aflat "la limită" (caz particular)
  - Presupunerea care stă la baza acestui tip de testare este aceea că dezvoltatorii omit adesea cazurile speciale ("frontierele" claselor de echivalență) (ex.: 0, stringuri vide, anul 2000, etc.)
- Ex.: pentru exemplul considerat anterior
  - Luna 2 (February) și anii 1900 și 2000 (un an multiplu de 100 nu este bisect decât dacă este și multiplu de 400)
  - Lunile 0 și 13, aflate la limita clasei de echivalență conținând lunile invalide

Equivalence class	Value for month input	Value for year input
Leap years divisible by 400	2 (February)	2000
Non-leap years divisible by 100	2 (February)	1900
Nonpositive invalid months	0	1291
Positive invalid months	13	1315

# Testarea căilor de execuție

- Metodă de tip whitebox, care identifică defecte în implementarea unei componente, prin testarea tuturor căilor de execuție din cod
  - Presupunerea din spatele acestei tehnici este aceea că, prin execuția, cel puțin o dată, a fiecărei căi (eng. *path*) din cod, majoritatea defectelor vor genera eșecuri
  - Punctul de start în aplicarea acestei metode îl constituie construirea unei reprezentări de tip schemă logică (eng. *flow graph*) asociate codului testat
    - nodurile corespund instrucțiunilor
    - arcele corespund fluxului de control
  - Ex.: implementare greșită a metodei *getNumDaysInMonth()*

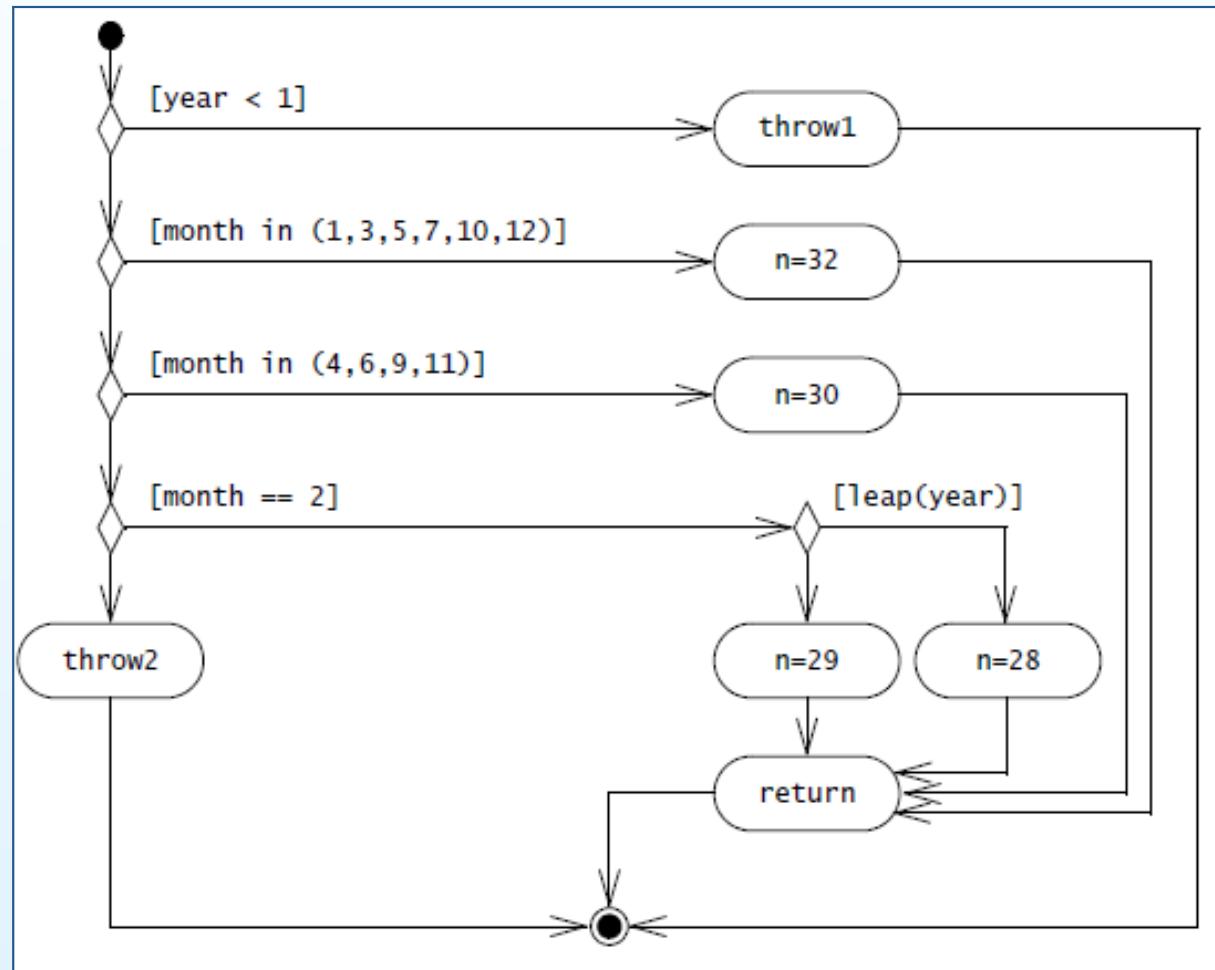
```
public class MonthOutOfBoundsException extends Exception {...};  
public class YearOutOfBoundsException extends Exception {...};  
  
class MyGregorianCalendar {  
    public static boolean isLeapYear(int year) {  
        boolean leap;  
        if ((year%4) == 0){  
            leap = true;  
        } else {  
            leap = false;  
        }  
        return leap;  
    }  
}
```

## Testarea căilor de execuție (cont.)

```
public static int getNumDaysInMonth(int month, int year)
    throws MonthOutOfBoundsException, YearOutOfBoundsException {
    int numDays;
    if (year < 1) {
        throw new YearOutOfBoundsException(year);
    }
    if (month == 1 || month == 3 || month == 5 || month == 7 || month == 10 || month == 12) {
        numDays = 31;
    } else if (month == 4 || month == 6 || month == 9 || month == 11) {
        numDays = 30;
    } else if (month == 2) {
        if (isLeapYear(year)) {
            numDays = 29;
        } else {
            numDays = 28;
        }
    } else {
        throw new MonthOutOfBoundsException(month);
    }
    return numDays;
}
```

## Testarea căilor de execuție (cont.)

- Schema logică aferentă implementării (greșite a) metodei `getNumDaysInMonth()` (diagramă UML de activități)



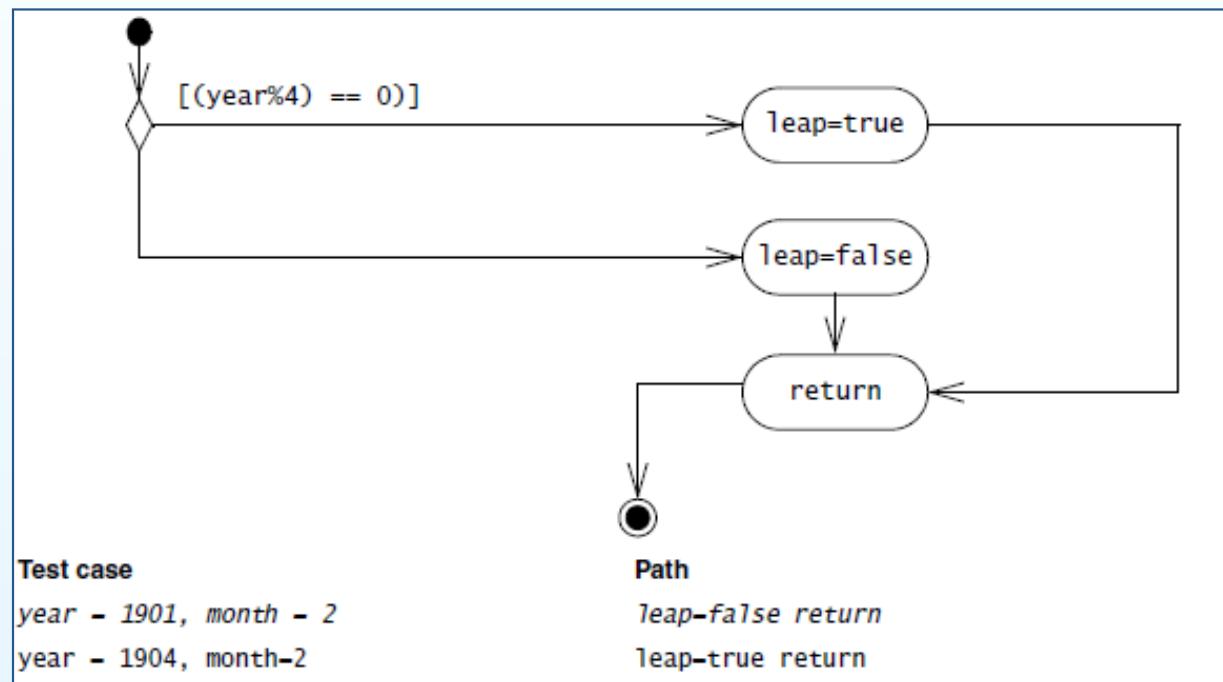
## Testarea căilor de execuție (cont.)

- Testarea completă a căilor de execuție presupune proiectarea cazurilor de test astfel încât fiecare arc al diagramei de activități să fie traversat cel puțin o dată
  - Aceasta presupune analiza fiecărui nod decizional și selectarea câte unei intrări pentru fiecare dintre ramurile *true* și *false*
    - combinația (1,1901) identifică defectul  $n=32$
  - Ex.: cazurile de test generate pentru metoda *getNumDaysInMonth()*

Test case	Path
(year = 0, month = 1)	{throw1}
(year = 1901, month = 1)	{n-32 return}
(year = 1901, month = 2)	{n-28 return}
(year = 1904, month = 2)	{n-29 return}
(year = 1901, month = 4)	{n-30 return}
(year = 1901, month = 0)	{throw2}

## Testarea căilor de execuție (cont.)

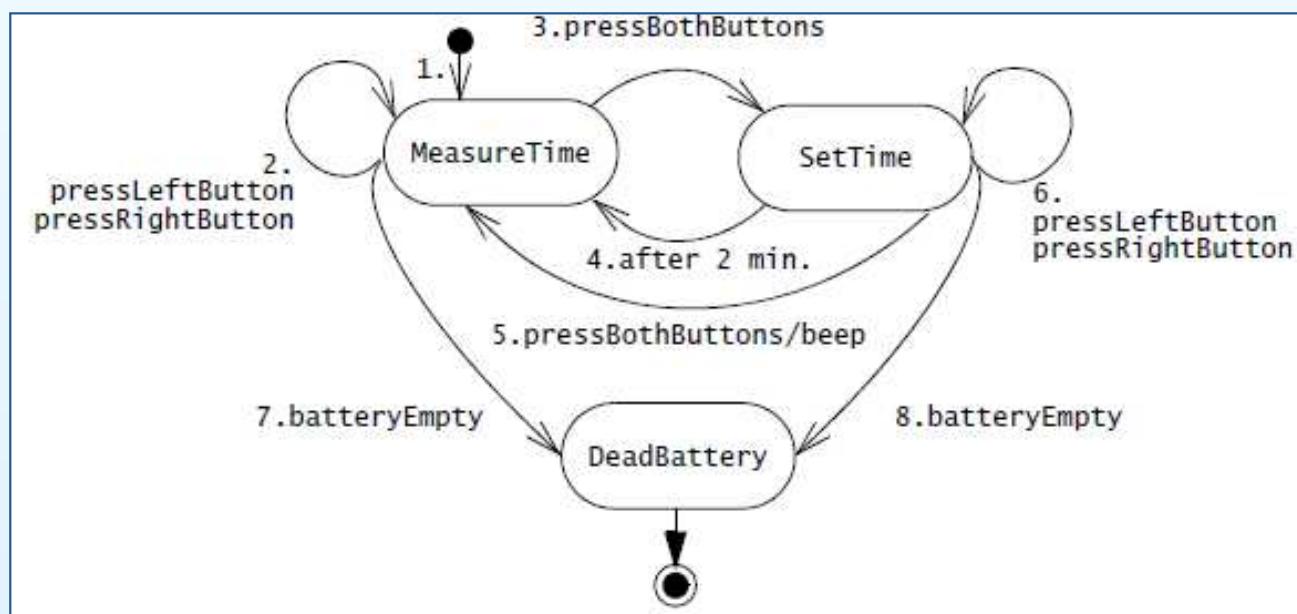
- Ex.: diagrama de activități și cazurile de test generate pentru metoda `isLeapYear()`



- Testarea căilor de execuție vs. testarea bazată pe echivalențe / a frontierelor
  - ambele detectează defectul  $n=32$
  - prima e posibil să nu genereze caz de test aferent anilor multiplu de 100
  - e posibil ca nici unul dintre cazurile de test generate de cele două metode să nu identifice eroarea asociată lunii 8

# Testarea bazată pe stări

- Tehnică de testare a sistemelor orientate obiect, care generează cazuri de test pentru o clasă pe baza diagramei UML de tranziție a stărilor asociată respectivei clase
  - Pentru fiecare stare, se stabilește un set reprezentativ de stimuli aferenți tranzițiilor posibile din acea stare (similar testării bazate pe echivalențe)
  - După aplicarea fiecărui stimул, se compară starea curentă a componentei cu cea indicată de diagramă, indicându-se eșec în caz de neconcordanță
- Ex.: diagramă de tranziție a stărilor aferentă clasei *2Bwatch*



## Testarea bazată pe stări (cont.)

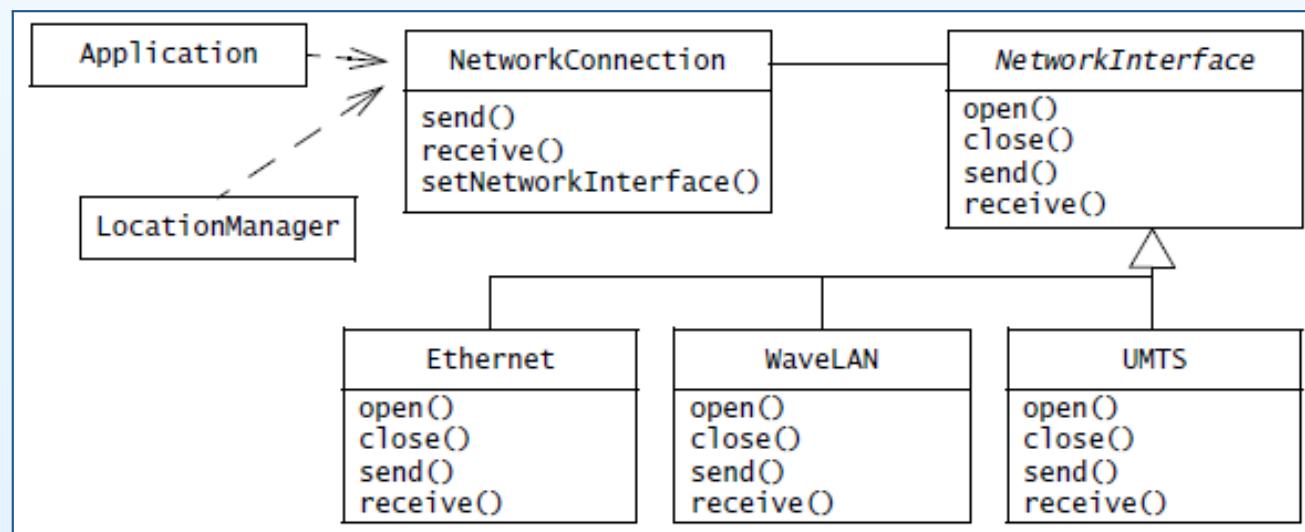
- Ex.: cazuri de test generate pentru sistemul *2Bwatch* (a.î. fiecare tranziție, exceptând 7 și 8, să fie traversată cel puțin o dată)

Stimuli	Transition tested	Predicted resulting state
Empty set	1. <i>Initial transition</i>	MeasureTime
Press left button	2.	MeasureTime
Press both buttons simultaneously	3.	SetTime
Wait 2 minutes	4. <i>Timeout</i>	MeasureTime
Press both buttons simultaneously	3. <i>Put the system into the SetTime state to test the next transition.</i>	SetTime
Press both buttons simultaneously	5.	SetTime->MeasureTime
Press both buttons simultaneously	3. <i>Put the system into the SetTime state to test the next transition.</i>	SetTime
Press left button	6. <del>Loop back onto MeasureTime</del>	<del>MeasureTime</del> SetTime

- Avantaje/dezavantaje ale testării bazate pe stări
  - - Starea fiind încapsulată, cazurile de test trebuie să includă aplicarea unor secvențe de stimuli care aduc componenta în starea dorită, înainte de a putea testa o anumită tranziție
  - + Potențial de automatizare

# Testarea polimorfismului

- Polimorfismul introduce o nouă provocare în procesul de testare, prin faptul că permite ca un același mesaj să se concretizeze înapeluri de metode diferite, funcție de tipul actual al apelatului
  - Atunci când se realizează testarea căilor de execuție pentru o metodă ce utilizează polimorfism, este necesar să se ia în calcul toate legăturile posibile => necesitatea de a expanda metoda pentru a aplica algoritmul clasic de test
  - Ex.: aplicare a şablonului *Strategy* pentru a încapsula diferite implementări *NetworkInterface*



## Testarea polimorfismului (cont.)

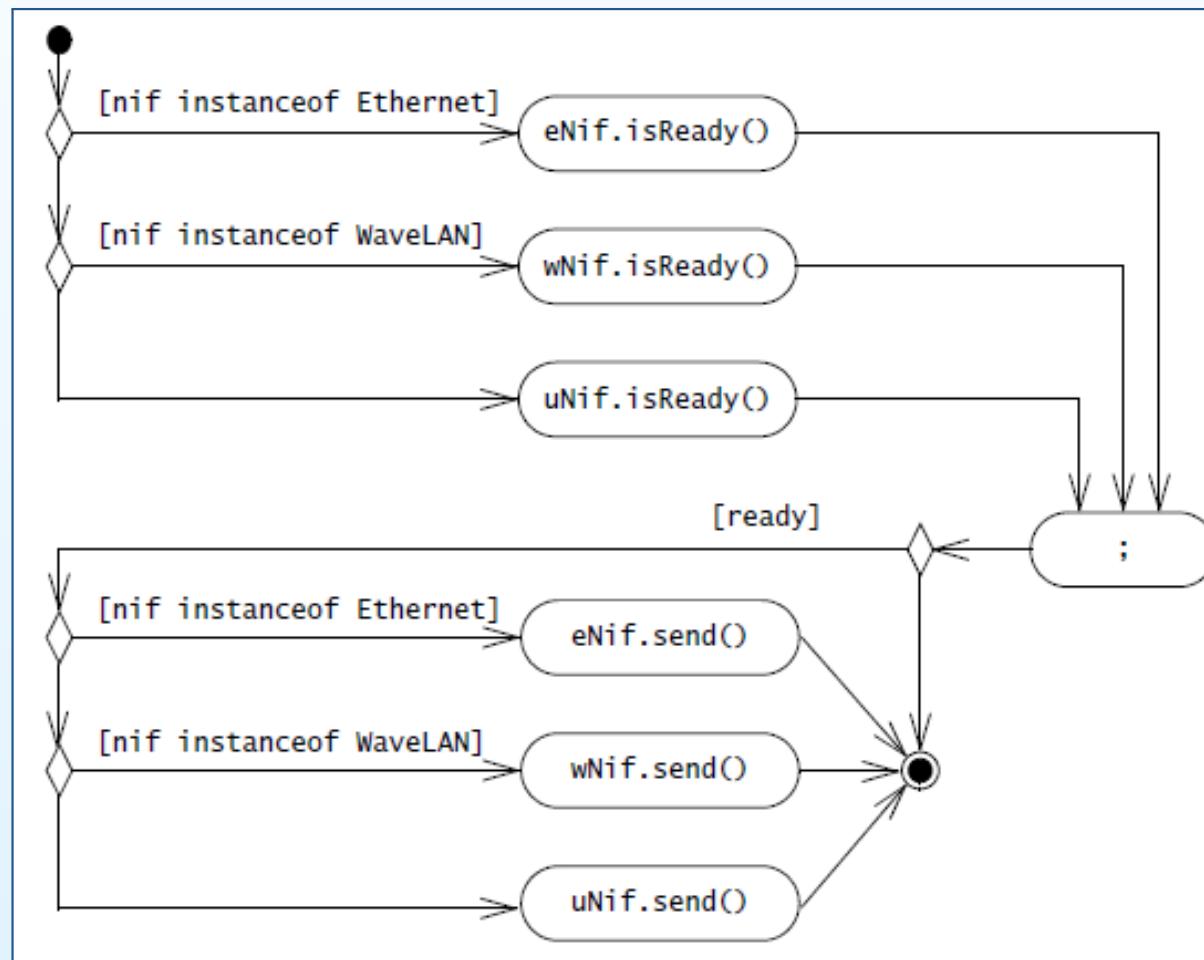
- Ex.: Codul sursă al metodei `NetworkConnection.send()`, cu și fără polimorfism (ultima variantă este cea folosită pentru generarea cazurilor de test)

```
public class NetworkConnection {  
    //...  
    private NetworkInterface nif;  
    void send(byte msg[]) {  
        queue.concat(msg);  
        if (nif.isReady()) {  
            nif.send(queue);  
            queue.setLength(0);  
        }  
    }  
}
```

```
public class NetworkConnection {  
    //...  
    private NetworkInterface nif;  
    void send(byte msg[]) {  
        queue.concat(msg);  
        boolean ready = false;  
        if (nif instanceof Ethernet) {  
            Ethernet eNif = (Ethernet)nif;  
            ready = eNif.isReady();  
        } else if (nif instanceof WaveLAN) {  
            WaveLAN wNif = (WaveLAN)nif;  
            ready = wNif.isReady();  
        } else if (nif instanceof UMTS) {  
            UMTS uNif = (UMTS)nif;  
            ready = uNif.isReady();  
        }  
        if (ready) {  
            if (nif instanceof Ethernet) {  
                Ethernet eNif = (Ethernet)nif;  
                eNif.send(queue);  
            } else if (nif instanceof WaveLAN){  
                WaveLAN wNif = (WaveLAN)nif;  
                wNif.send(queue);  
            } else if (nif instanceof UMTS){  
                UMTS uNif = (UMTS)nif;  
                uNif.send(queue);  
            }  
            queue.setLength(0);  
        }  
    }  
}
```

## Testarea polimorfismului (cont.)

- Ex.: Diagrama de activități aferentă variantei expandate a codului sursă al metodei `NetworkConnection.send()` (generarea cazurilor de test se face după metoda prezentată la "Testarea căilor de execuție")



## Referințe

---

- [Fagan, 1976] M. E. Fagan, *Design and code inspections to reduce errors in program development*, IBM Systems Journal, Vol. 15, No. 3, 1976.
- [Parnas and Weiss, 1985] D. L. Parnas and D. M. Weiss, *Active design reviews: principles and practice*, Proceedings of the Eighth International Conference on Software Engineering, London, U.K., pp 132-136, August 1985.