

Ingineria Programării

Cursul 5 – 17 Martie
adiftene@infoiasi.ro

Cuprins

- ▶ Din Cursurile trecute...
- ▶ Diagrame UML
 - Diagrame de Stări
 - Diagrame de Activități
 - Diagrame de Deployment
 - Diagrame de Pachete
- ▶ GRASP
 - Information Expert
 - Creator
 - Low coupling
 - High cohesion
 - Controller

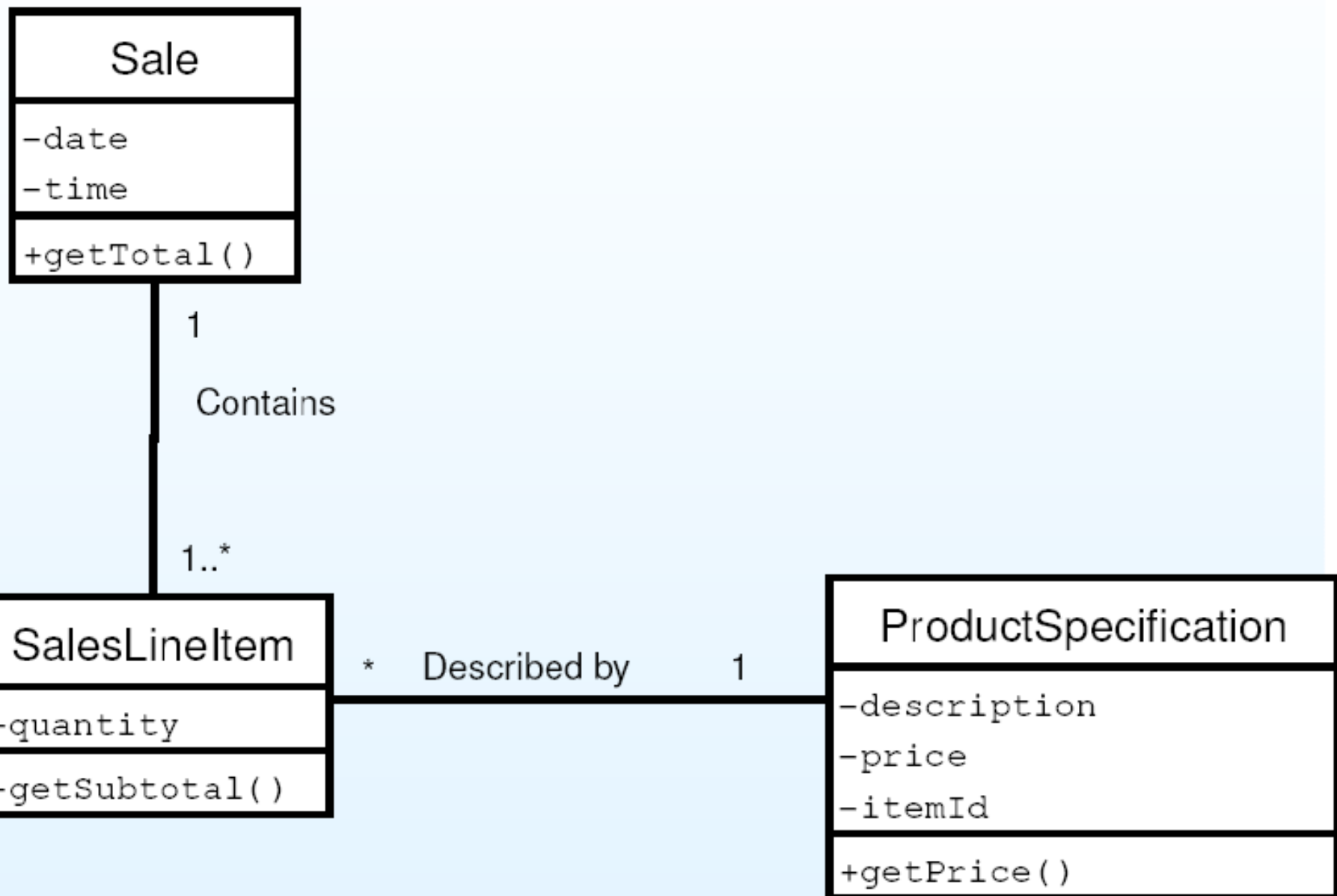
Atenție

- ▶ Săptămâna 7-a e termenul limită pentru alegerea proiectului
- ▶ După care urmează: documentare, înțelegere, knowledge transfer, diagrame use case, diagrame de clasă, implementare, unit testing, etc.
- ▶ Săptămâna a 7-a începe efectiv lucrul la proiect, iar evaluarea se încheie în săptămâna a 15-a
- ▶ În săptămâna a 8-a nu se fac ore...

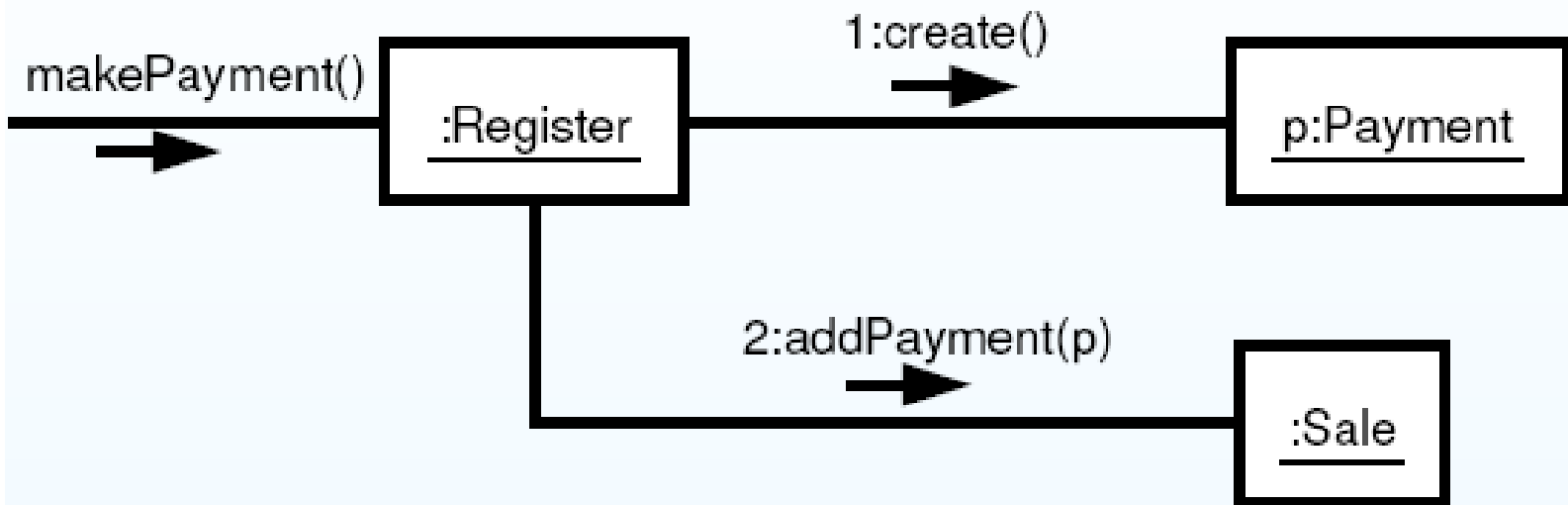
Din cursurile trecute 1

- ▶ Diagrame UML
- ▶ Diagrame Use Case
- ▶ Diagrame de Clase
- ▶ Diagrame de Secvență
- ▶ Diagrame de Colaborare

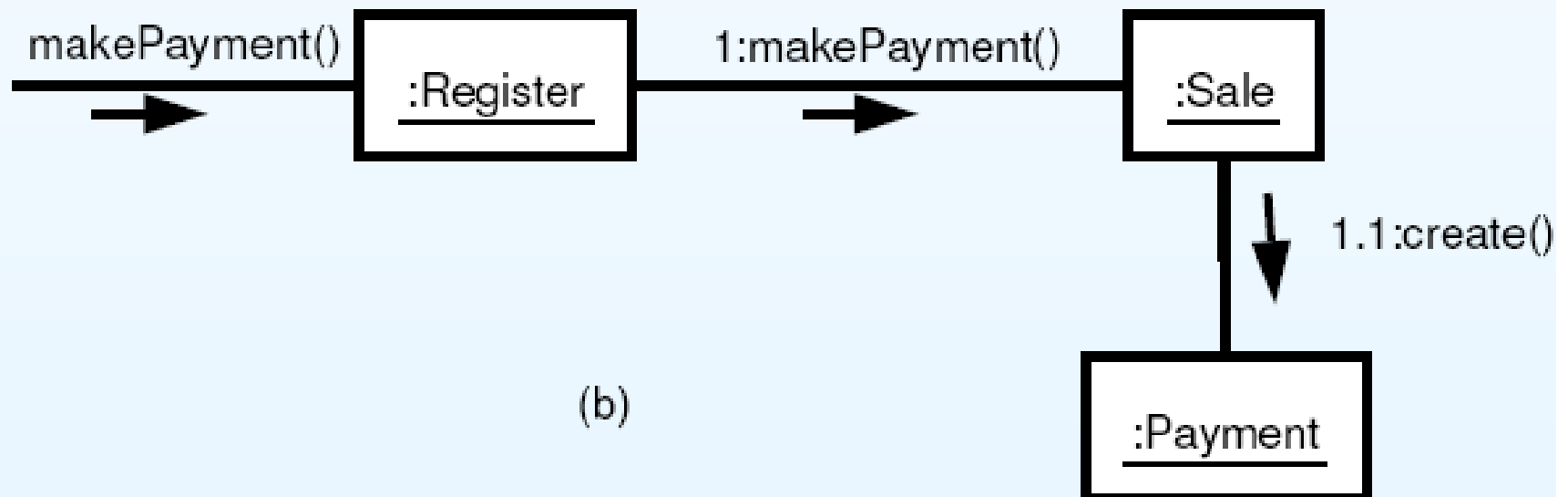
Din cursurile trecute 2



Din cursurile trecute 3

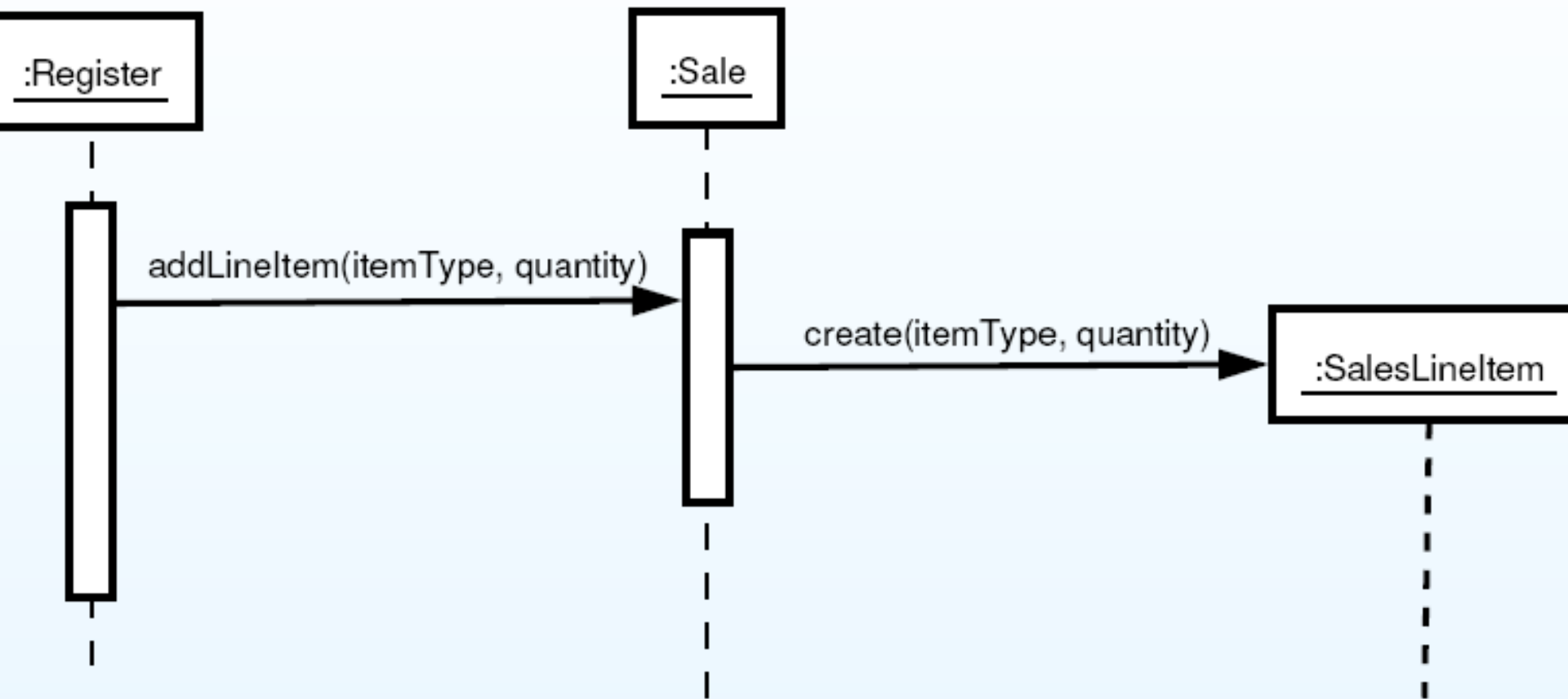


(a)

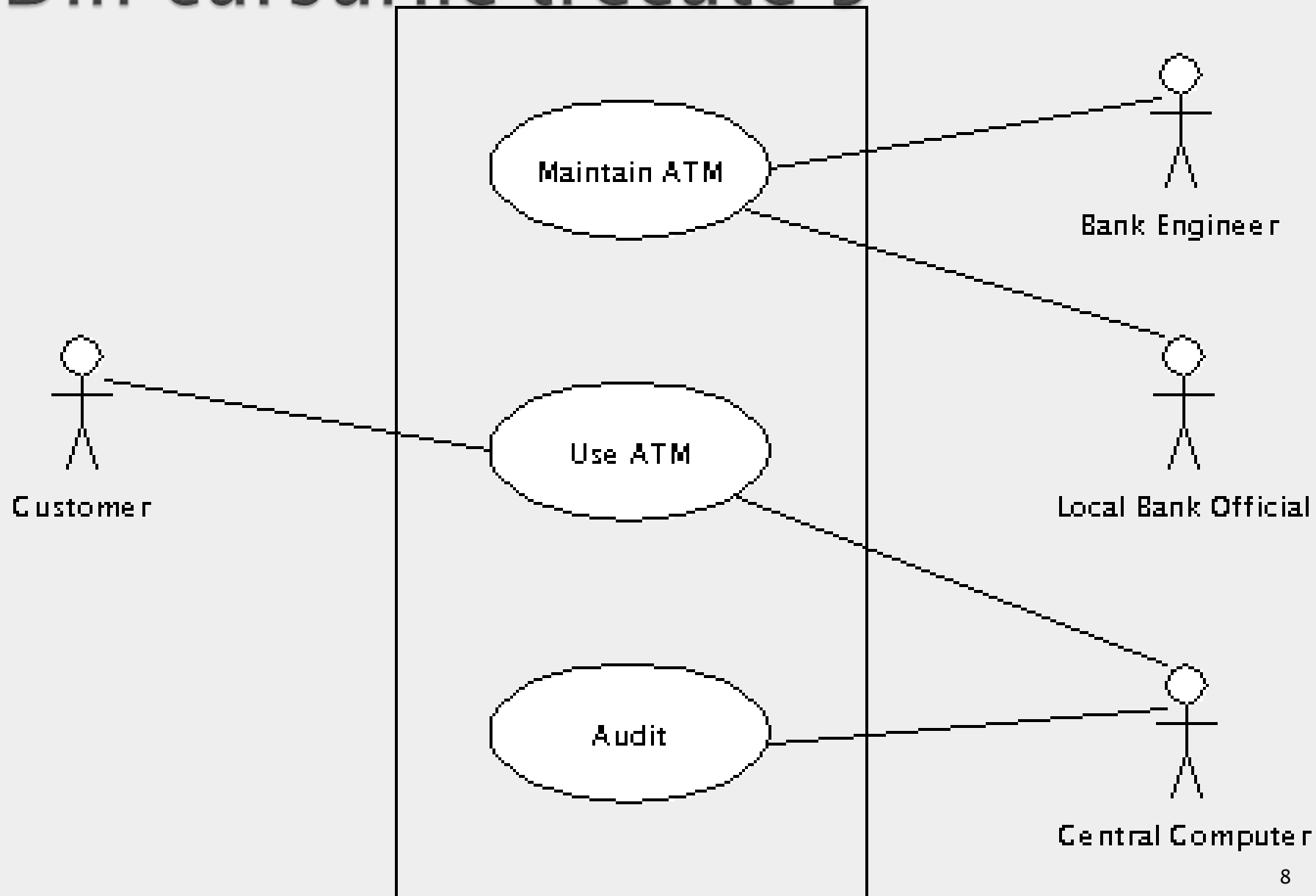


(b)

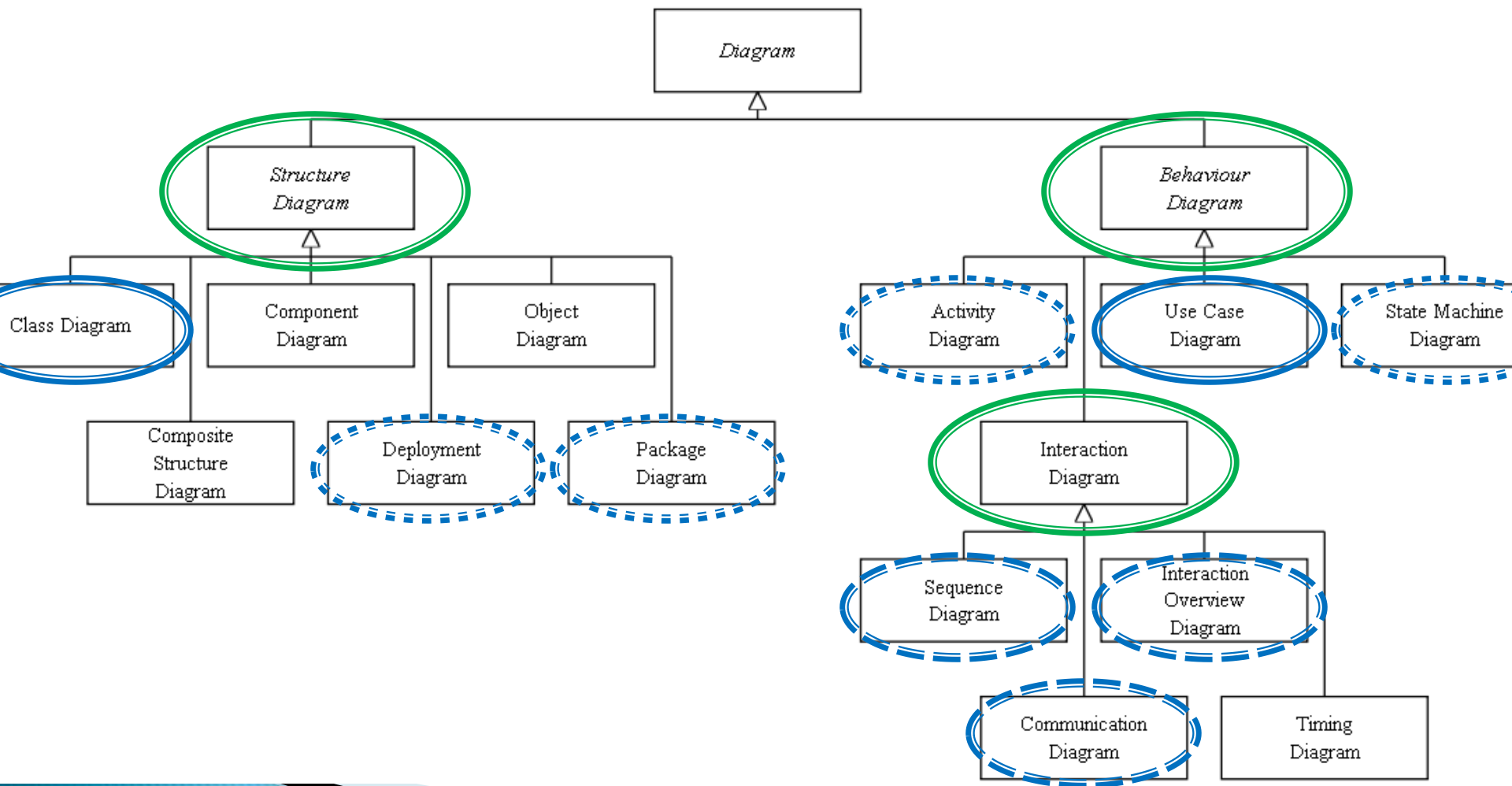
Din cursurile trecute 4



Din cursurile trecute 5



UML2.0 – 13 Tipuri de Diagrame



Diagrame comportamentale

- ▶ Diagrame de stări, diagrame de activități
- ▶ Elemente de bază
 - Eveniment
 - Acțiune
 - Activitate

Eveniment

- ▶ **Reprezintă ceva atomic care se întâmplă la un moment dat**
- ▶ **Modelează apariția unui stimul care poate conduce la efectuarea unei tranziții**
- ▶ **Are atașată o locație în timp și spațiu**
- ▶ **Nu are o durată în timp**
- ▶ **Evenimentele pot fi:**
 - **sincrone sau asincrone**
 - **externe sau interne**

Exemple de evenimente

- ▶ **Semnal** = stimul asincron care are un nume, este aruncat de un obiect și este recepționat de altul (ex. excepții)
- ▶ **Apel de operație** (de obicei sincron)
- ▶ **Trecerea timpului**
- ▶ **Schimbarea rezultatului evaluării unei condiții**

Acțiune

- ▶ Reprezintă execuția atomică a unui calcul
- ▶ Are ca efect:
 - returnarea unei valori
 - schimbarea stării
- ▶ Are o durată mică în timp
- ▶ Exemplu:
 - `i++;`

Activitate

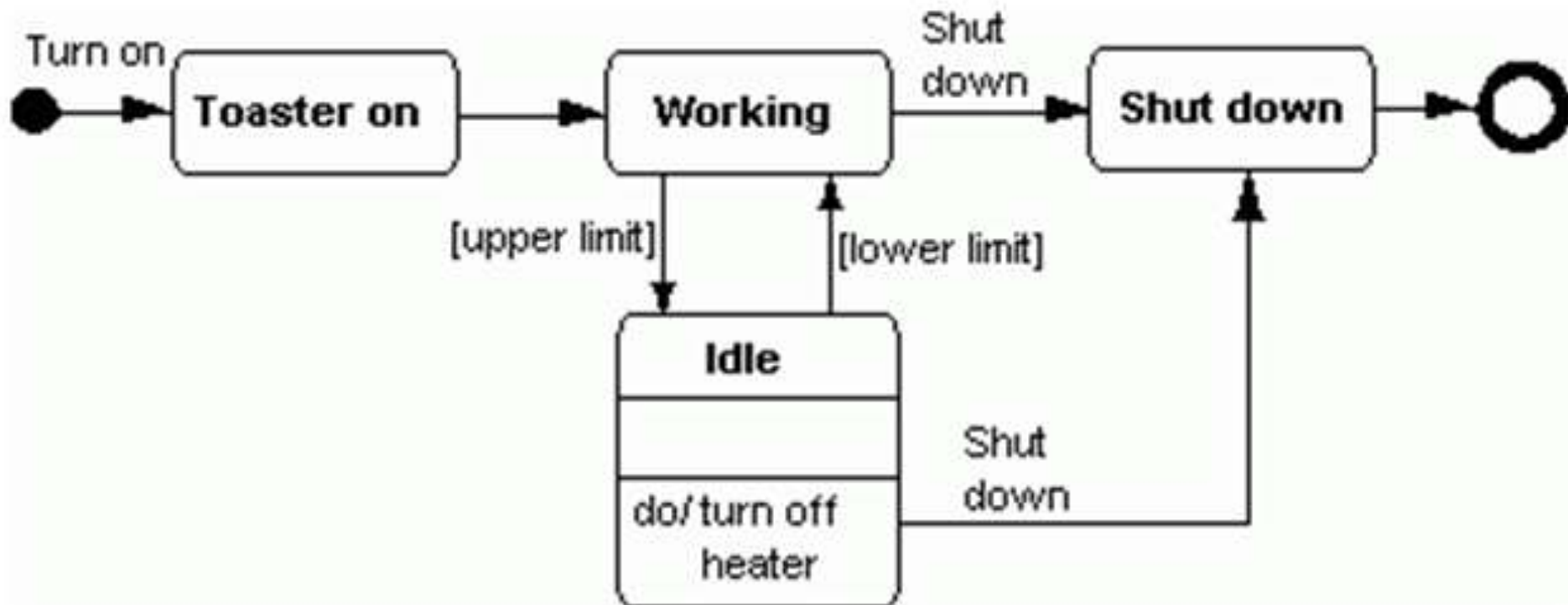
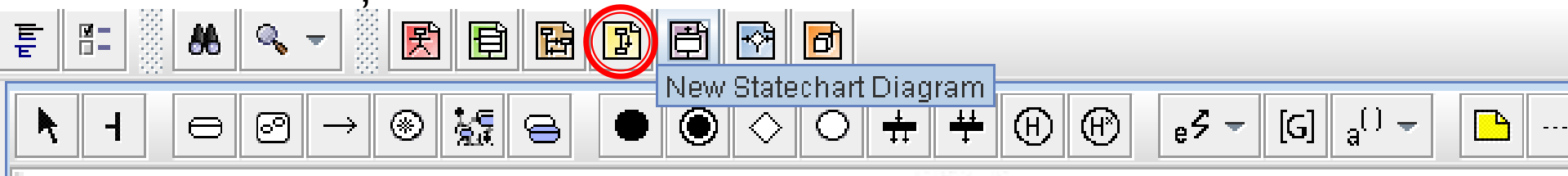
- ▶ Reprezintă execuția neatomică a unor acțiuni
- ▶ Are o durată în timp
- ▶ Exemplu:
 - vorbitul la telefon
 - execuția unei funcții

Diagrame de Stări (State Chart Diagram)

- ▶ Folosită pentru a modela comportamentul **unui singur obiect**
- ▶ Specifică o **secvență de stări prin care trece un obiect de-a lungul vieții sale** ca răspuns la apariția unor evenimente împreună cu răspunsul la acele evenimente
- ▶ *Una din cele mai răspândite metode de descriere a comportamentului dinamic al sistemelor complexe*

Diagramă de Stări 2

- ▶ Conține:
 - Stări
 - Tranziții

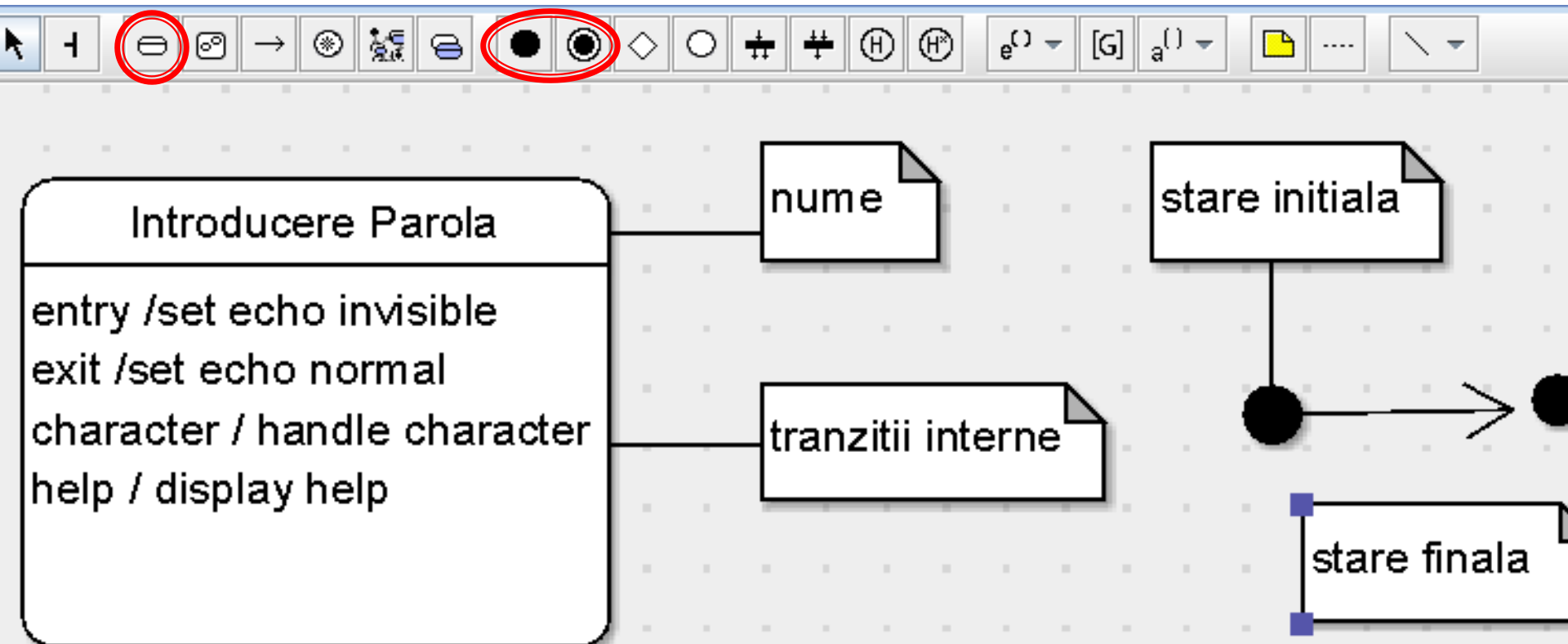


Stare

- ▶ Reprezintă o perioadă din viața unui obiect în care acesta:
 - Satisface anumite condiții,
 - Execută o acțiune sau
 - Așteaptă apariția unui eveniment
- ▶ Stările pot fi:
 - Simple
 - Compuse
 - Concurente
 - Secvențial active

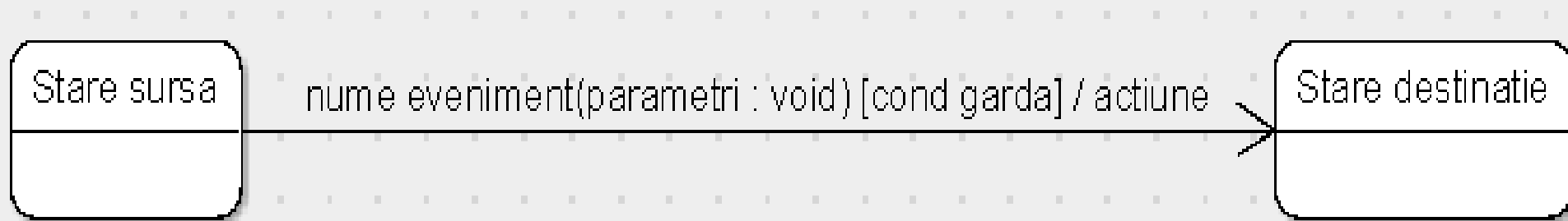
Stare: Notăție grafică

- ▶ Elementele unei stări:
 - **nume**: identifică o stare
 - **tranziții interne**: acțiuni și activități pe care obiectul le poate executa cât timp se află în acea stare



Tranziție

- ▶ Reprezintă o relație între două stări
- ▶ Indică faptul că un obiect aflat în prima stare va efectua niște acțiuni și apoi va intra în starea a doua atunci când un anumit eveniment se produce
- ▶ Notăție grafică:



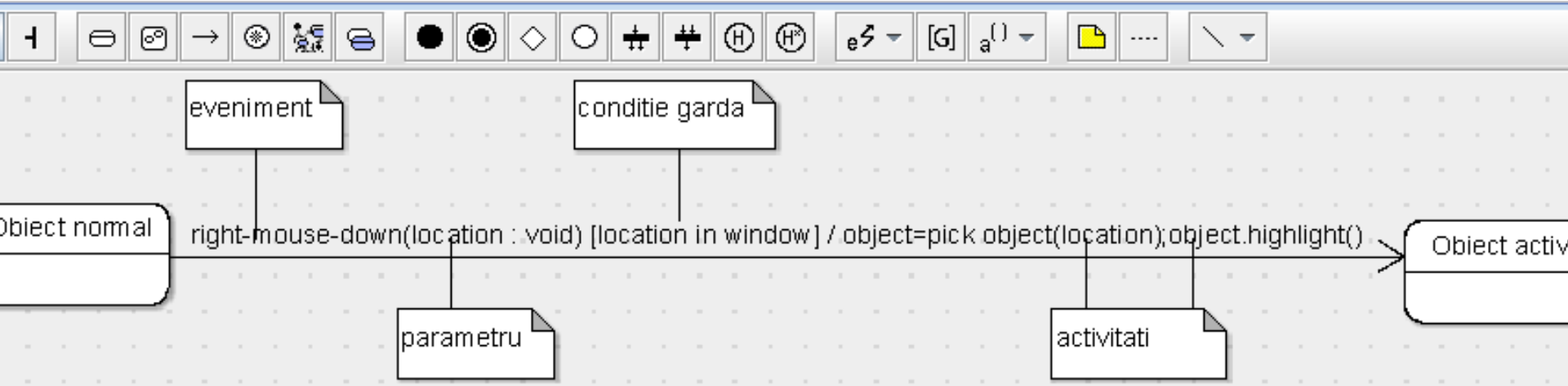
Tranziție Internă

- ▶ Forma generală a unei tranziții interne:

nume eveniment (lista parametrilor) [cond gardă] / acțiune

- ▶ **nume eveniment**
 - Identifică circumstanțele în care acțiunea specificată se execută
 - nume predefinite: *entry, exit, do, include*
- ▶ **cond gardă** este o expresie booleană care se evaluează la fiecare apariție a evenimentului specificat; acțiunea se execută doar în cazul în care rezultatul evaluării este TRUE
- ▶ **acțiunea** poate folosi attribute și legături care sunt vizibile entității modelate

Exemplu de Tranziție

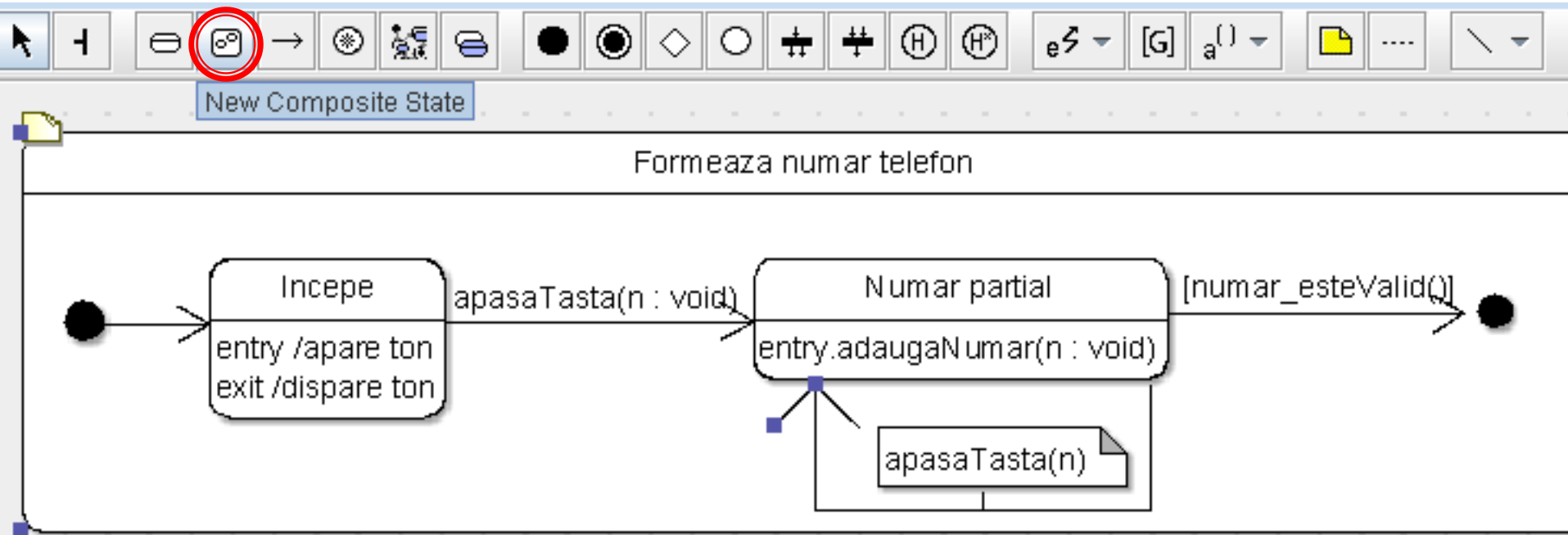


Stări Compuse

- ▶ Conțin
 - Substări – pot conține, la rândul lor, alte substări
 - Secvențial active (disjuncte)
 - Paralel active (concurente)
 - Tranziții interne

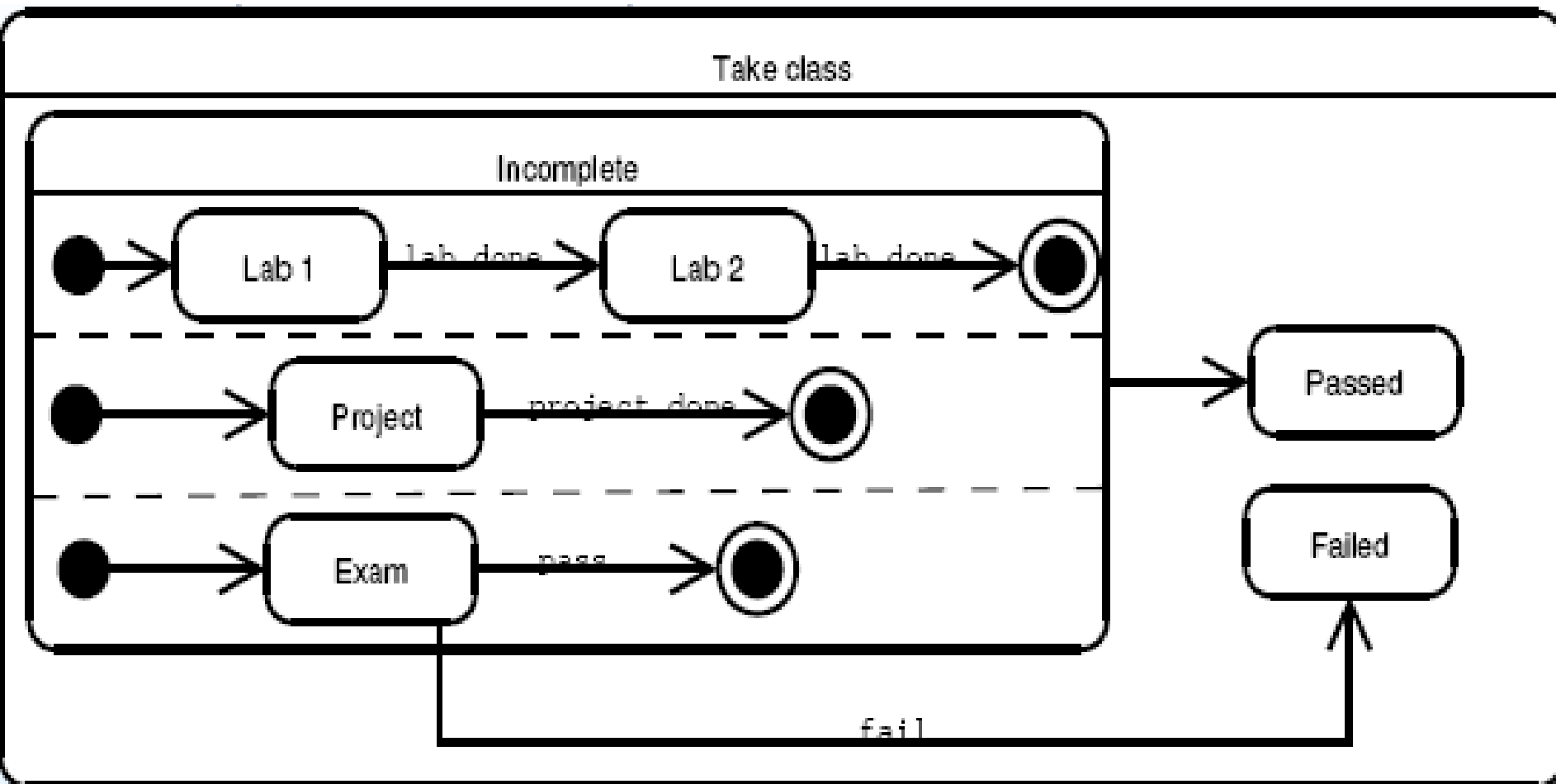
Exemplu de Stare compusă 1

- ▶ Stare compusă cu substări **secvențial** active:

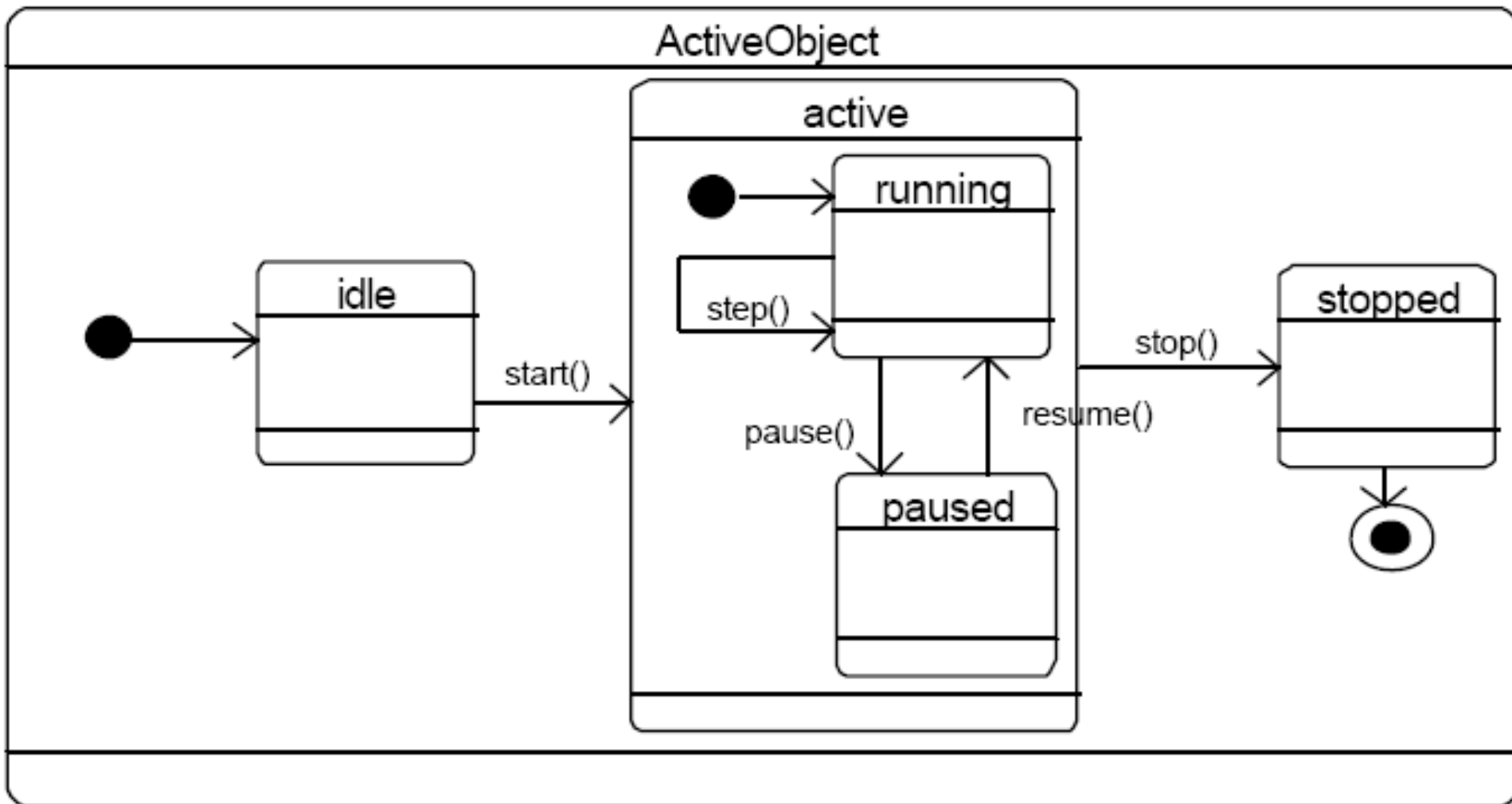


Exemplu de Stare compusă 2

- ▶ Stare compusă cu substări **paralele** active:



Exemplul de Diagramă de Stări

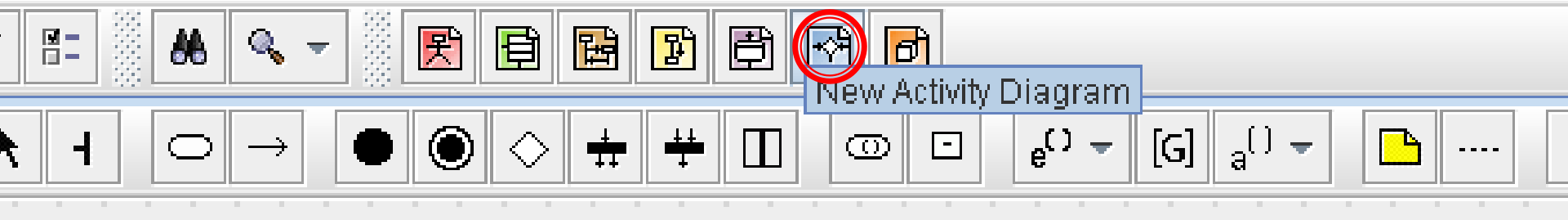


Diagramă de Activități (Activity Diagram)

- ▶ Folosită pentru a modela dinamica unui proces sau a unei operații
- ▶ Evidențiază controlul execuției de la o activitate la alta
- ▶ Se atașează:
 - Unei clase (modelează un caz de utilizare)
 - Unui pachet
 - Implementării unei operații

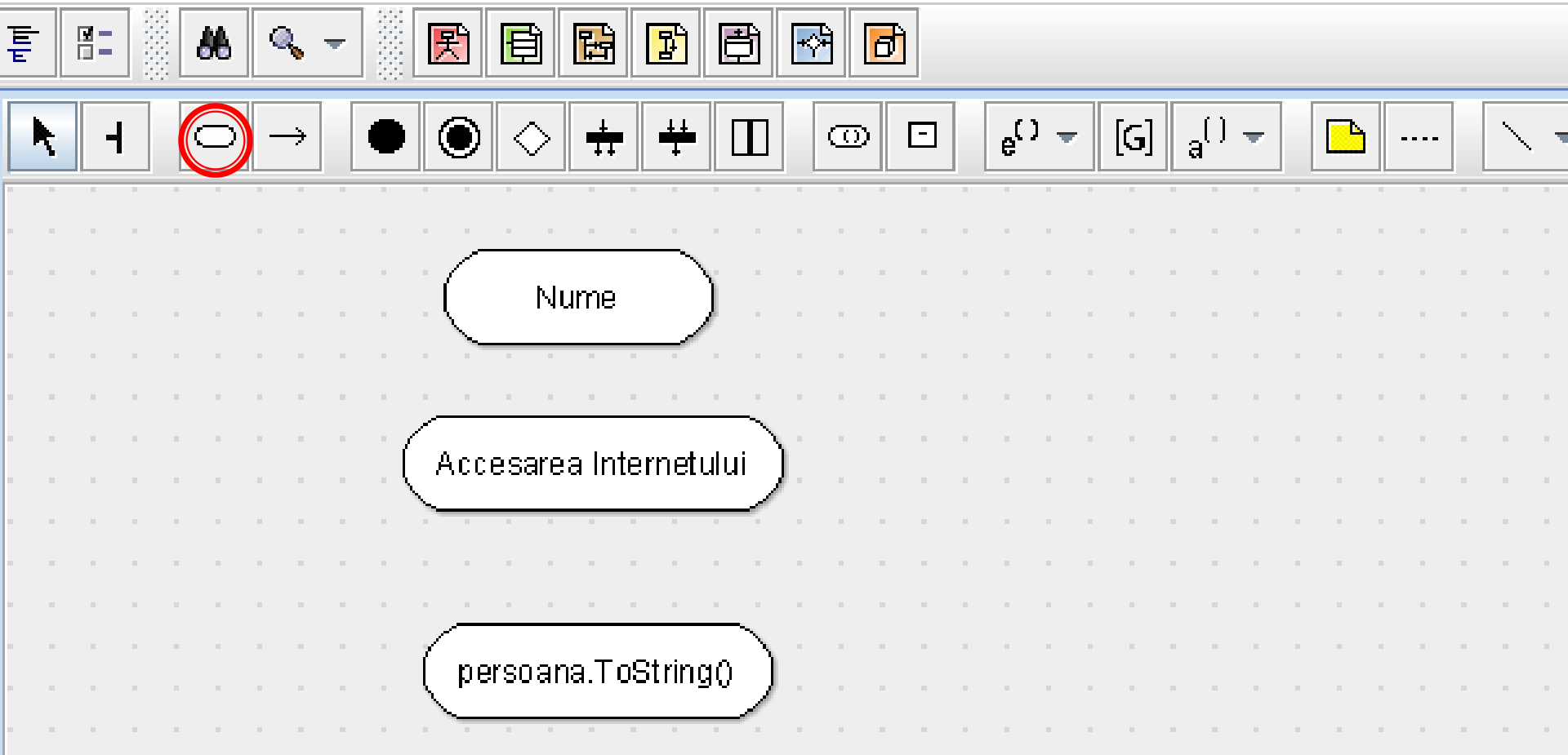
Diagrame de Activități 2

- ▶ Poate conține:
 - Stări activitate/acțiune
 - Tranziții
 - Obiecte
 - Bare de sincronizare
 - Ramificații



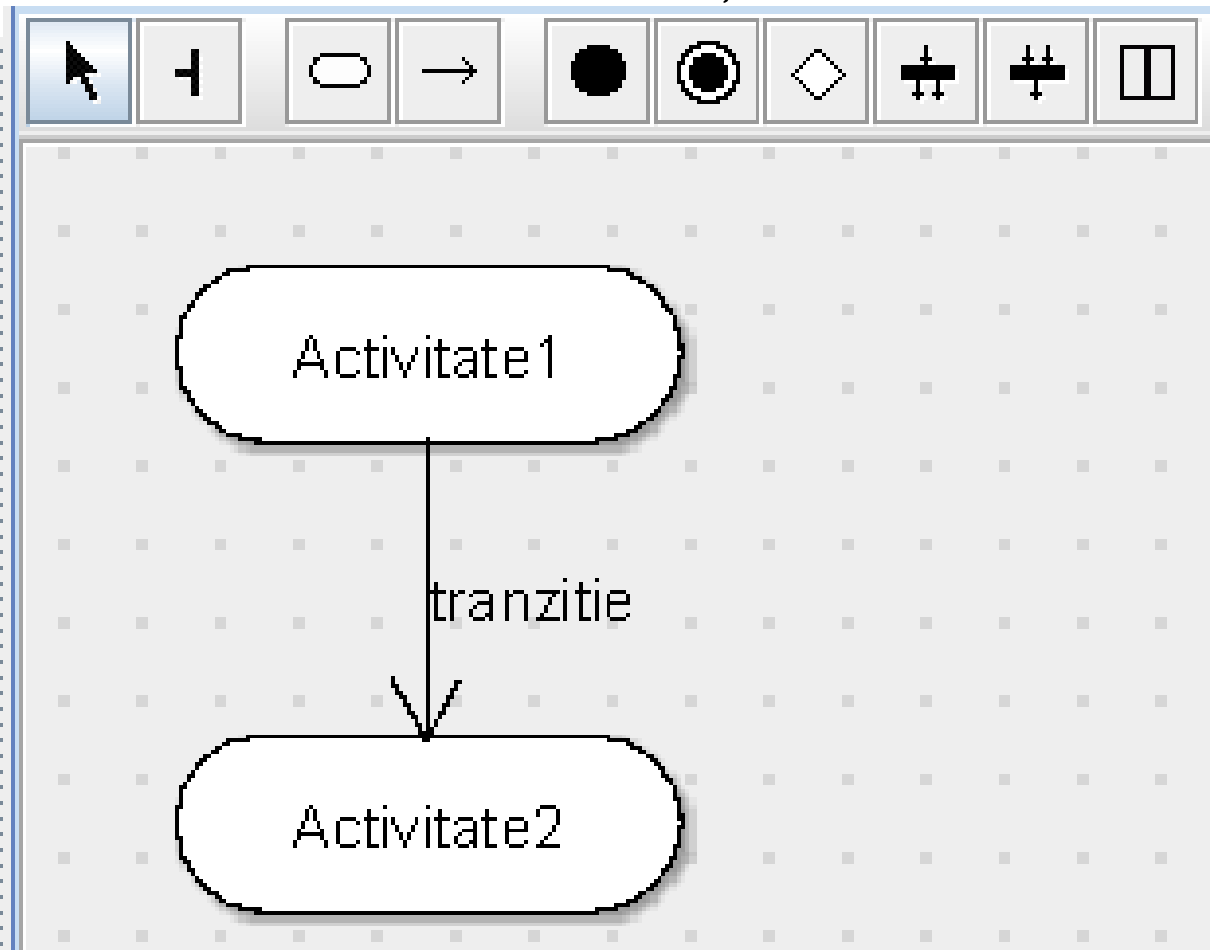
Stare activitate/acțiune

- ▶ Modelează execuția unor acțiuni sau a unor subactivități



Tranziție

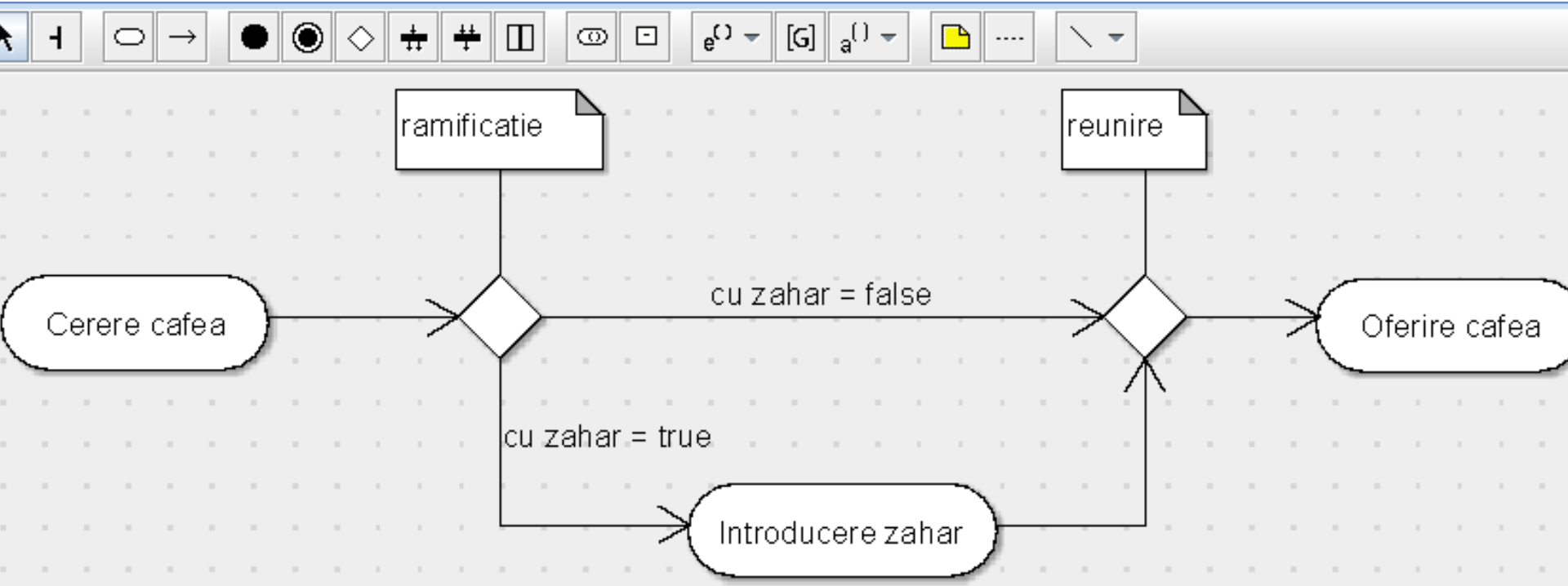
- ▶ Reprezintă o relație între două activități
- ▶ Tranziția este inițiată de terminarea primei activități și are ca efect preluarea controlului execuției de către a doua activitate



Ramificație

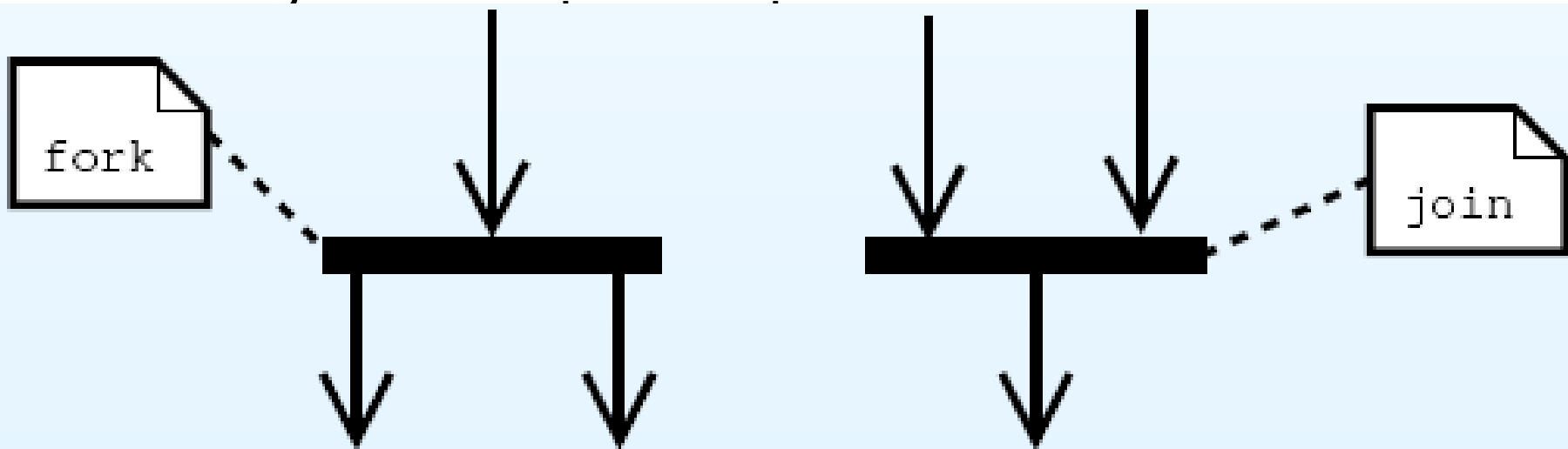
- ▶ Se folosește pentru a **modela alternative** (decizii) a căror alegere **depinde de o expresie booleană**
- ▶ Are o tranziție de intrare și două sau mai multe tranziții de ieșire
- ▶ Fiecare tranziție de ieșire trebuie să aibă o **condiție gardă**
- ▶ **Condițiile gardă trebuie să fie disjuncte** (să nu se suprapună) și să acopere toate situațiile posibile de continuare a execuției

Exemplu de Ramificație

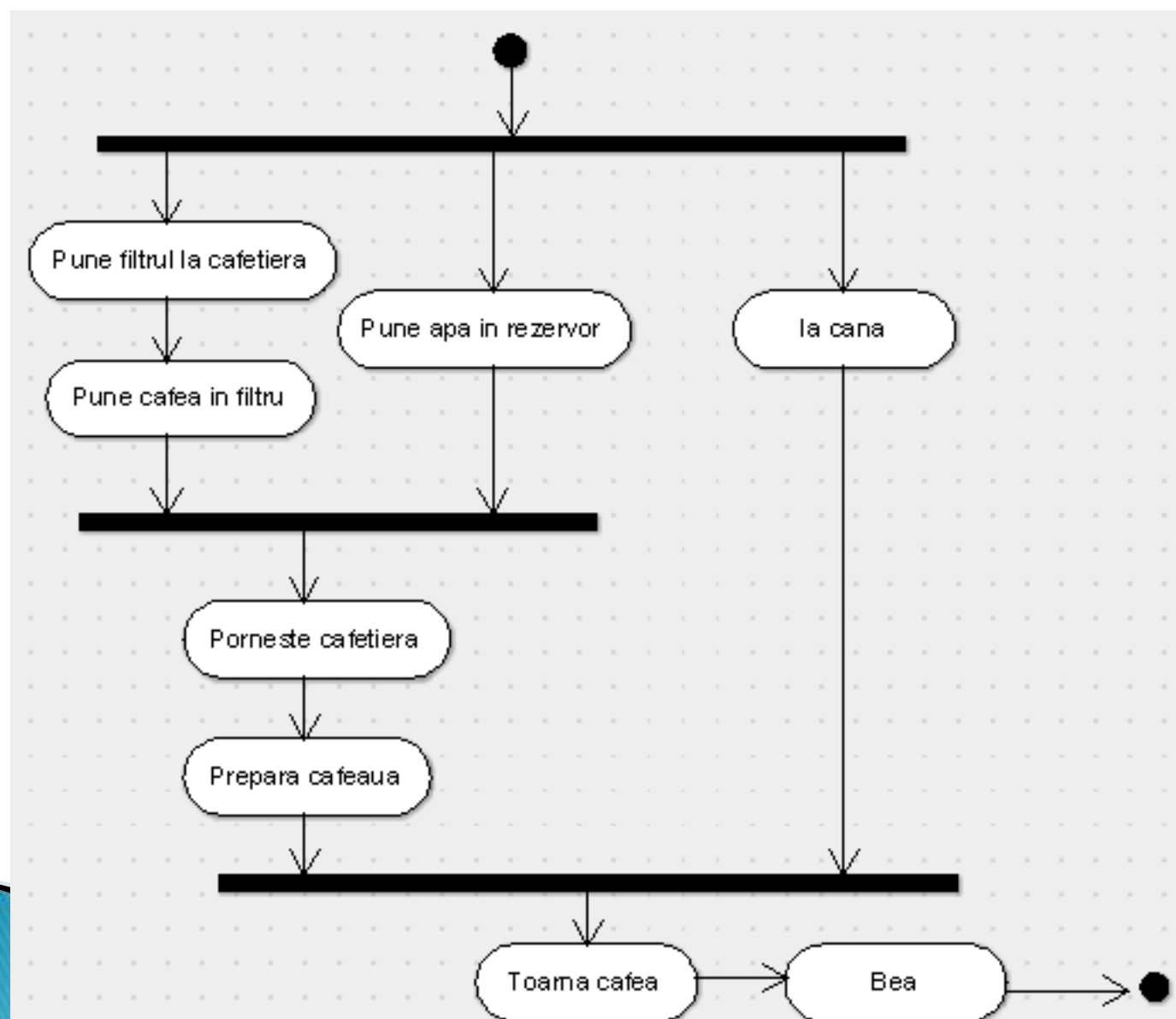


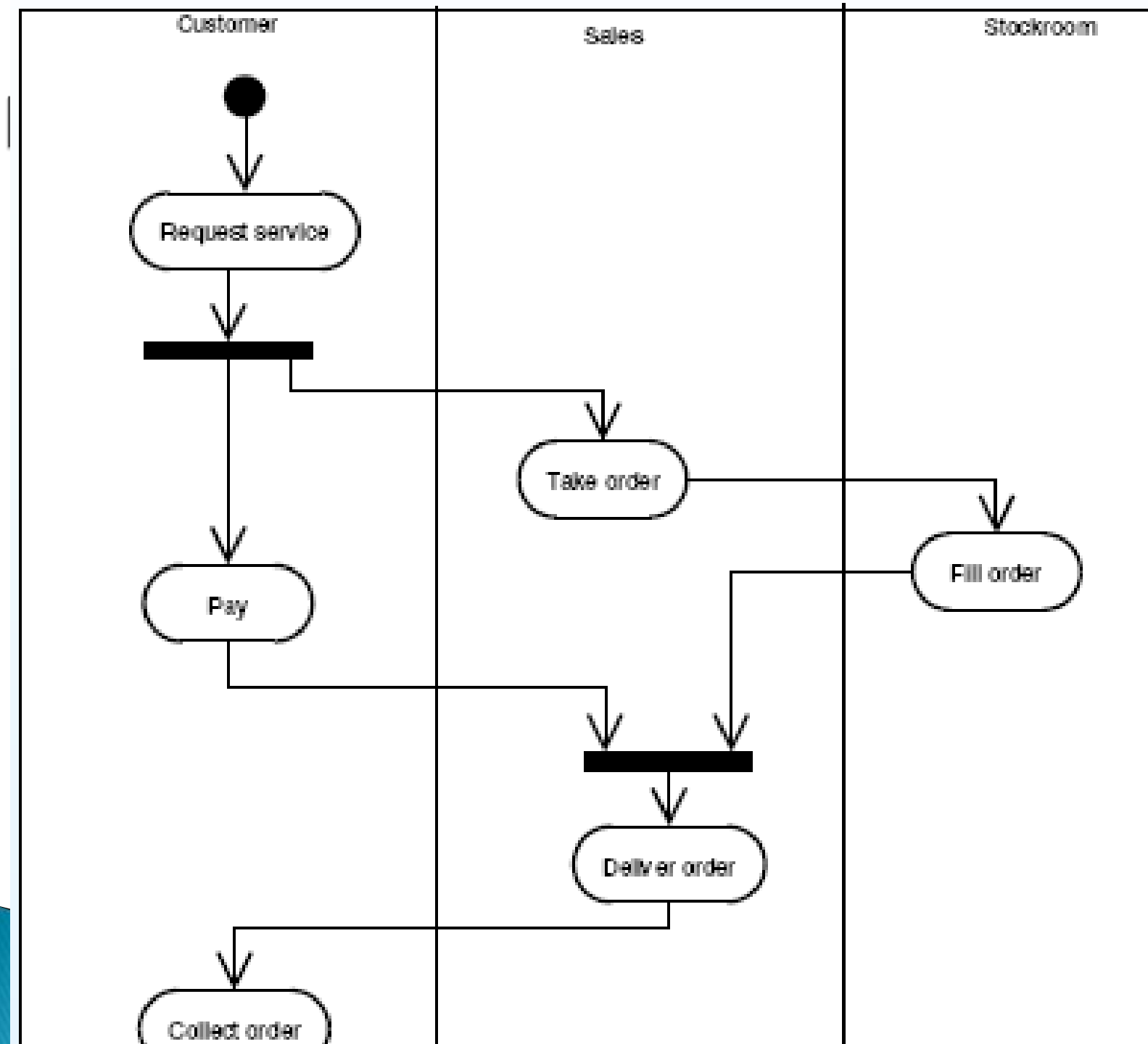
Bară de Sincronizare

- ▶ Folosită pentru a modela sincronizarea mai multor activități care se execută în paralel
- ▶ Poate fi de două tipuri:
 - **fork**: are o tranziție de intrare și două sau mai multe tranziții de ieșire
 - **join**: are două sau mai multe tranziții de intrare și o singură tranziție de ieșire



Exemplu de DA (Sincronizare)



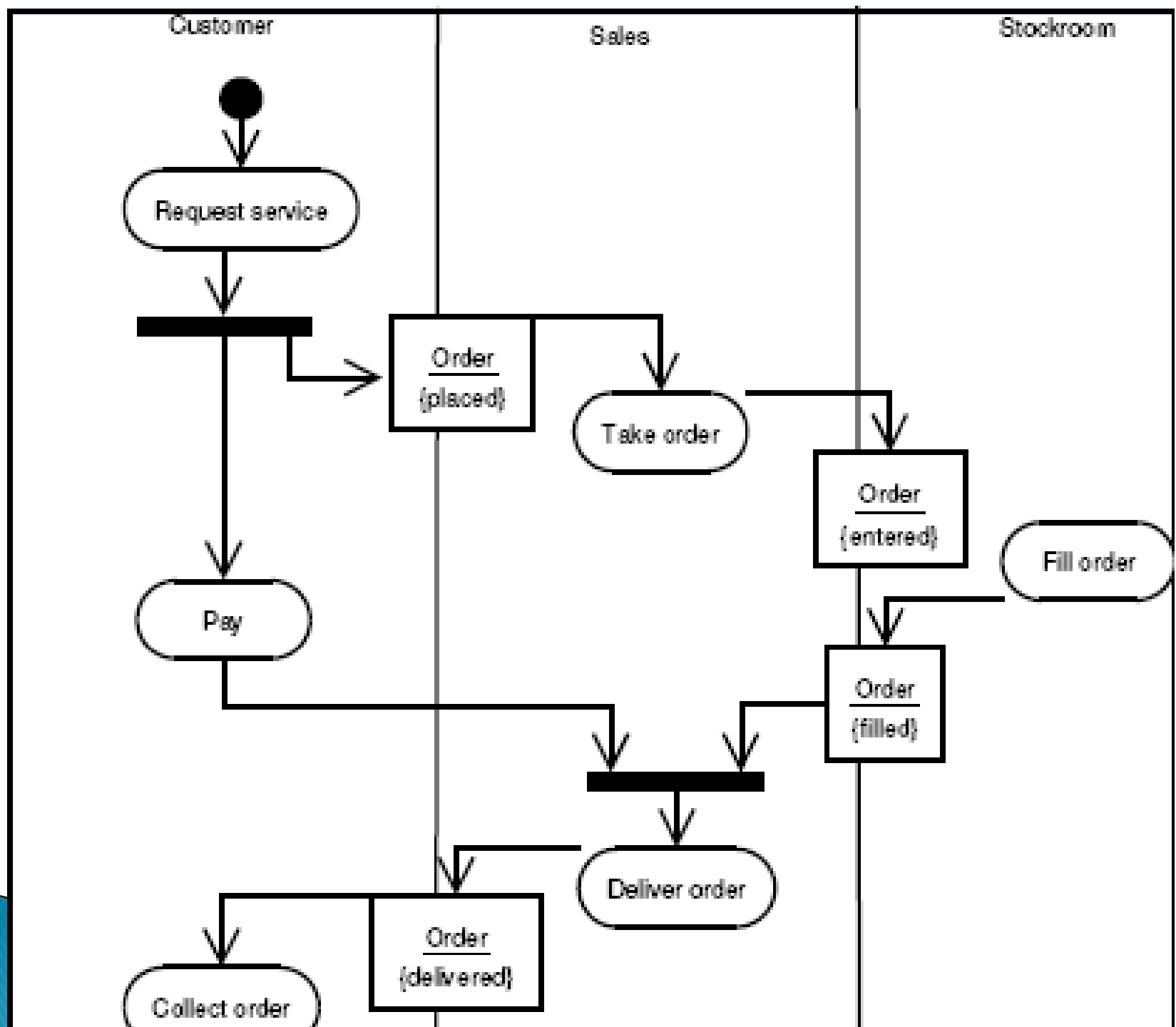


Obiecte 1

- ▶ Acțiunile sunt realizate de către obiecte sau operează asupra unor obiecte
- ▶ Obiectele pot constitui parametri de intrare/ieșire pentru acțiuni
- ▶ Obiectele pot fi conectate de acțiuni prin linii punctate cu o săgeată la unul din capete (orientarea săgeții indica tipul parametrului – intrare sau ieșire)

Obiecte 2

- ▶ Un obiect poate apărea de mai multe ori în cadrul aceleiași diagrame de activități
- ▶ Fiecare apariție indică un alt punct (stare) în viața obiectului
- ▶ Pentru a distinge aparițiile, numele stării obiectului poate fi adăugat la sfârșitul numelui obiectului

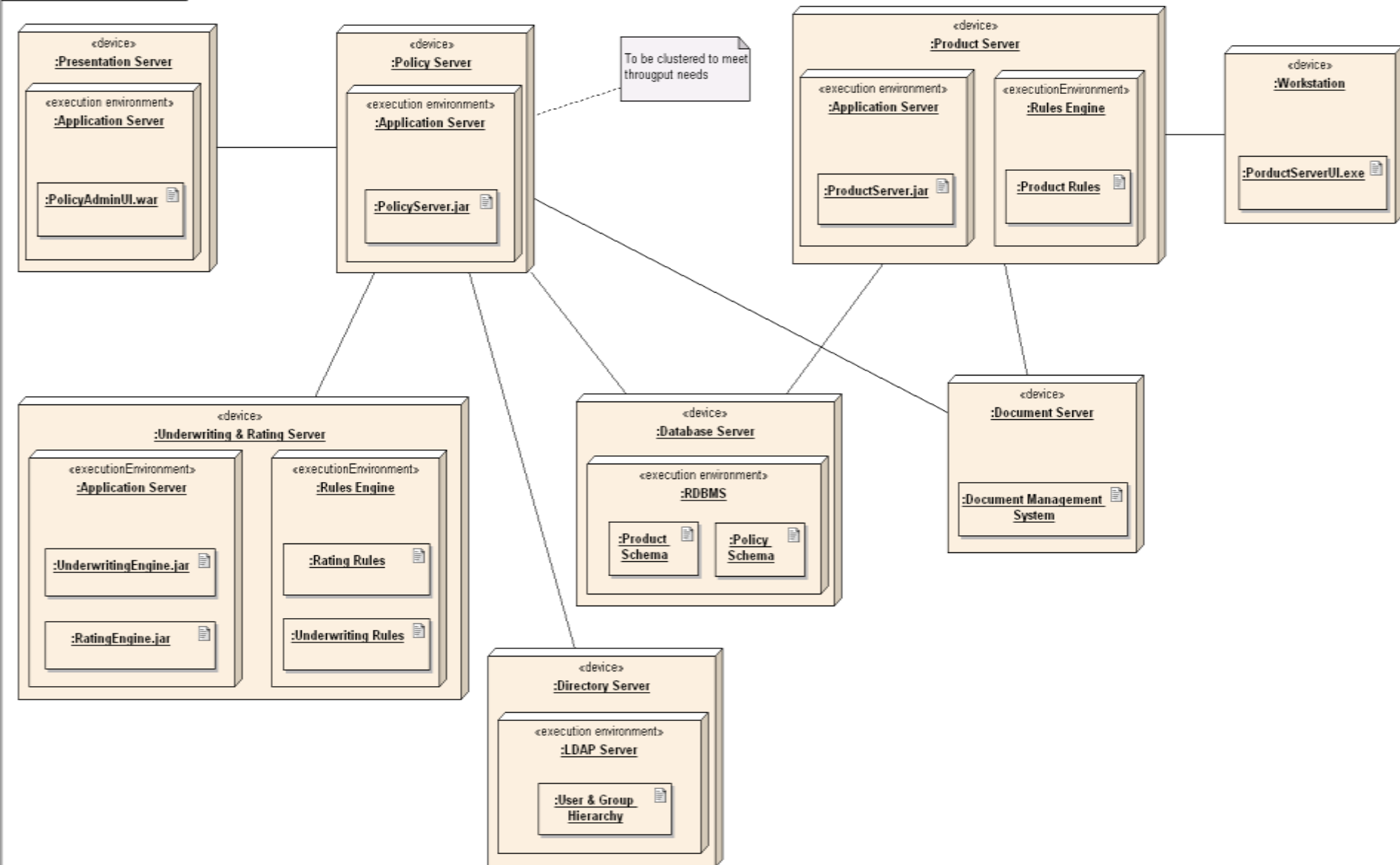


Diagrame de deployment

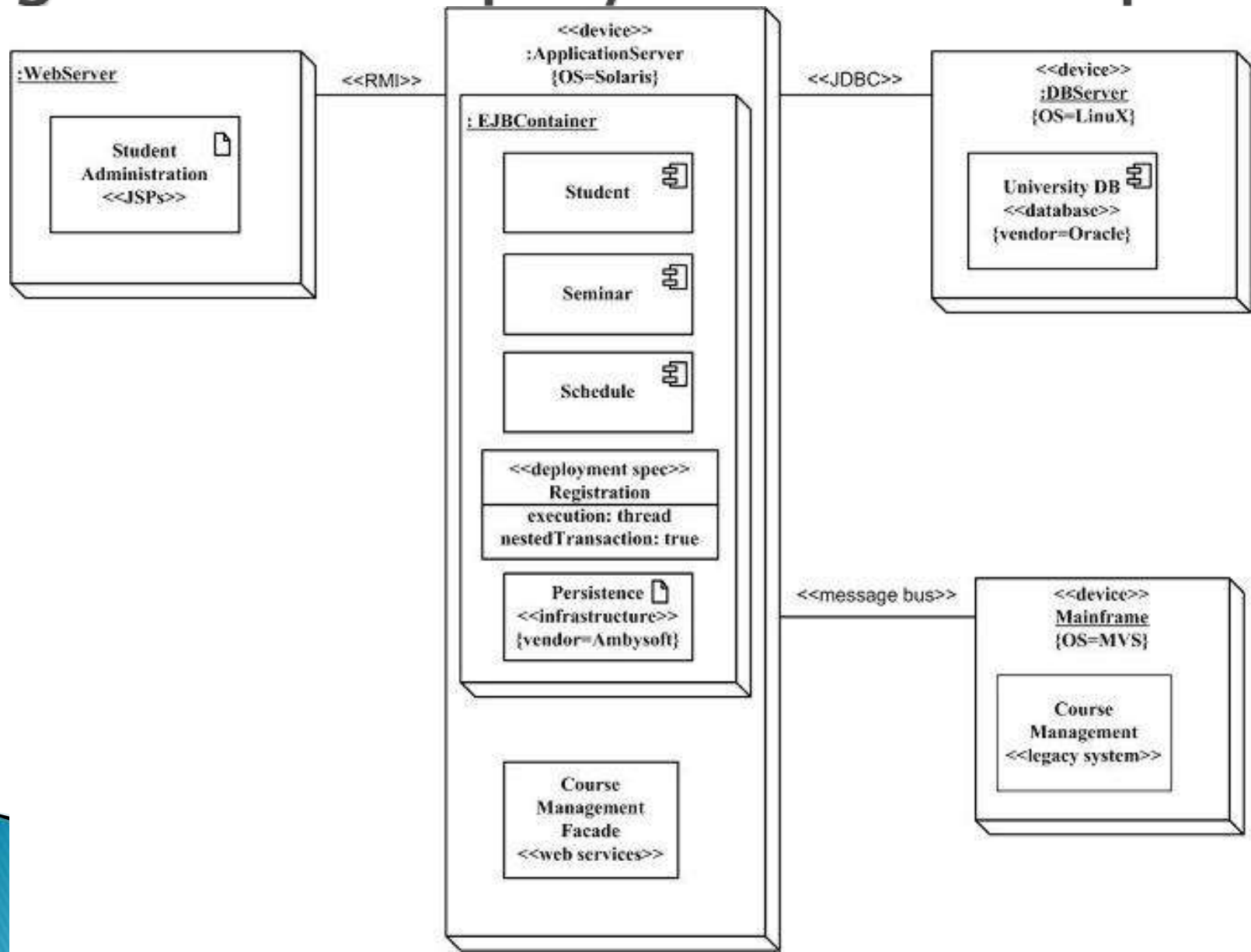
- ▶ Modelează mediul hardware în care va funcționa proiectul
- ▶ Exemplu: pentru a descrie un site web o diagramă de deployment va conține **componentele hardware**
 - server-ul web,
 - server-ul de aplicații,
 - server-ul de baze de date
- ▶ **Componentele software** de pe fiecare din acestea
 - Aplicația web
 - Baza de date
- ▶ Modul în care acestea sunt conectate:
 - JDBC, REST, RMI

Diagramă de deployment – Exemplu 1

dd Deployment of Components



Diagramă de deployment – Exemplul 2



Diagrame de Pachete (Package Diagram)

► Pachetul:

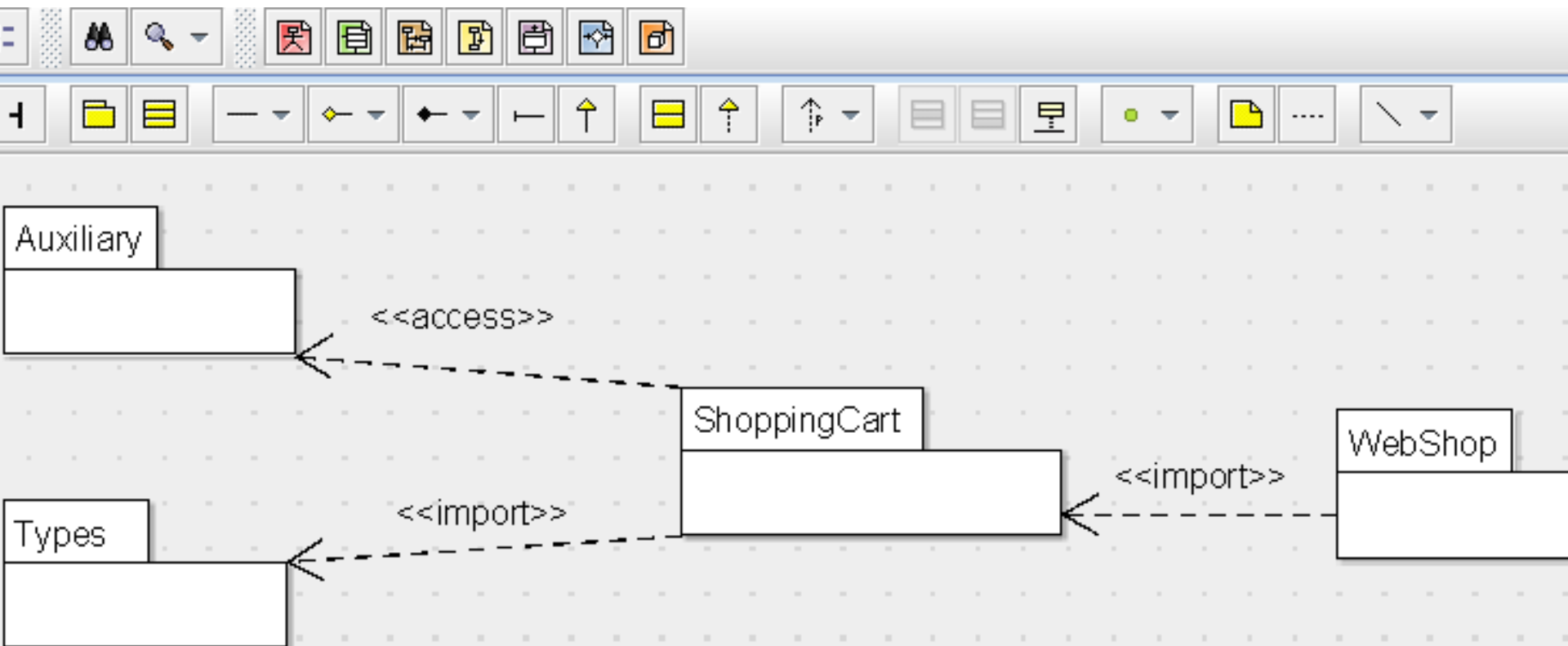
- Este un container logic pentru elemente între care se stabilesc legături
- Definește un spațiu de nume
- Toate elementele UML pot fi grupate în pachete (cel mai des pachetele sunt folosite pentru a grupa clase)
- Un pachet poate conține subpachete => se creează o structură arborescentă (similară cu organizarea fișierele/directoarelor)

Diagrame de Pachete 2

- ▶ Relații:
 - dependență `<<access>>` = import privat
 - dependență `<<import>>` = import public
- ▶ Ambele relații permit folosirea elementelor aflate în pachetul destinație de către elementele aflate în pachetul sursă fără a fi necesară calificarea numelor elementelor din pachetul destinație (similar directivei **import** din java)
- ▶ Aceste tipuri de diagrame se realizează în cadrul diagramelor de clasă

Exemplu de Diagramă de Pachete

- ▶ Elementele din Types sunt importate în ShoppingCart și apoi sunt importate mai departe de către WebShop
- ▶ Elementele din Auxiliary pot fi accesate însă doar din ShoppingCart și nu pot fi referite folosind nume necalificate din WebShop



Utilitatea diagramelor de pachete

- ▶ Împart sisteme mari în subsisteme mai mici și mai ușor de gestionat
- ▶ Permit dezvoltare paralelă iterativă
- ▶ Definirea unor interfețe clare între pachete promovează refolosirea codului (ex. pachet care oferă funcții grafice, pachet care oferă posibilitatea conectării la BD, etc...)

Recomandări în realizarea diagramelor UML

- ▶ Diagramele să nu fie nici prea complicate, dar nici prea simple: scopul este comunicarea eficientă
- ▶ Dați nume sugestive elementelor componente
- ▶ Aranjați elementele astfel încât liniile să nu se intersecteze
- ▶ Încercați să nu arătați prea multe tipuri de relații odată (evitați diagramele foarte complicate)
- ▶ Dacă este nevoie, realizați mai multe diagrame de același tip

GRASP

- ▶ GRASP = General Responsibility Assignment Software Patterns (Principles)
- ▶ Descrise de Craig Larman în cartea *Applying UML and Patterns. An Introduction to Object Oriented Analysis and Design*
- ▶ Ne ajută să alocăm responsabilități claselor și obiectelor în cel mai elegant mod posibil
- ▶ Exemple de principii folosite în GRASP: *Information Expert* (sau *Expert*), *Creator*, *High Cohesion*, *Low Coupling*, *Controller*, *Polymorphism*, *Pure Fabrication*, *Indirection*, *Protected Variations*

Ce responsabilități?

▶ Să facă:

- Să facă ceva el însuși, precum crearea unui obiect sau să facă un calcul
- Inițializarea unei acțiuni în alte obiecte
- Controlarea și coordonarea activităților altor obiecte

▶ Să cunoască:

- Atributele private
- Obiectele proprii
- Lucrurile pe care le poate face sau le poate apela

Pattern

- ▶ Traducere: șablon, model
- ▶ Este o soluție generală la o problemă comună
- ▶ Fiecare pattern are un nume sugestiv și ușor de reținut (ex. composite, observer, iterator, singleton, etc.)

Information Expert 1

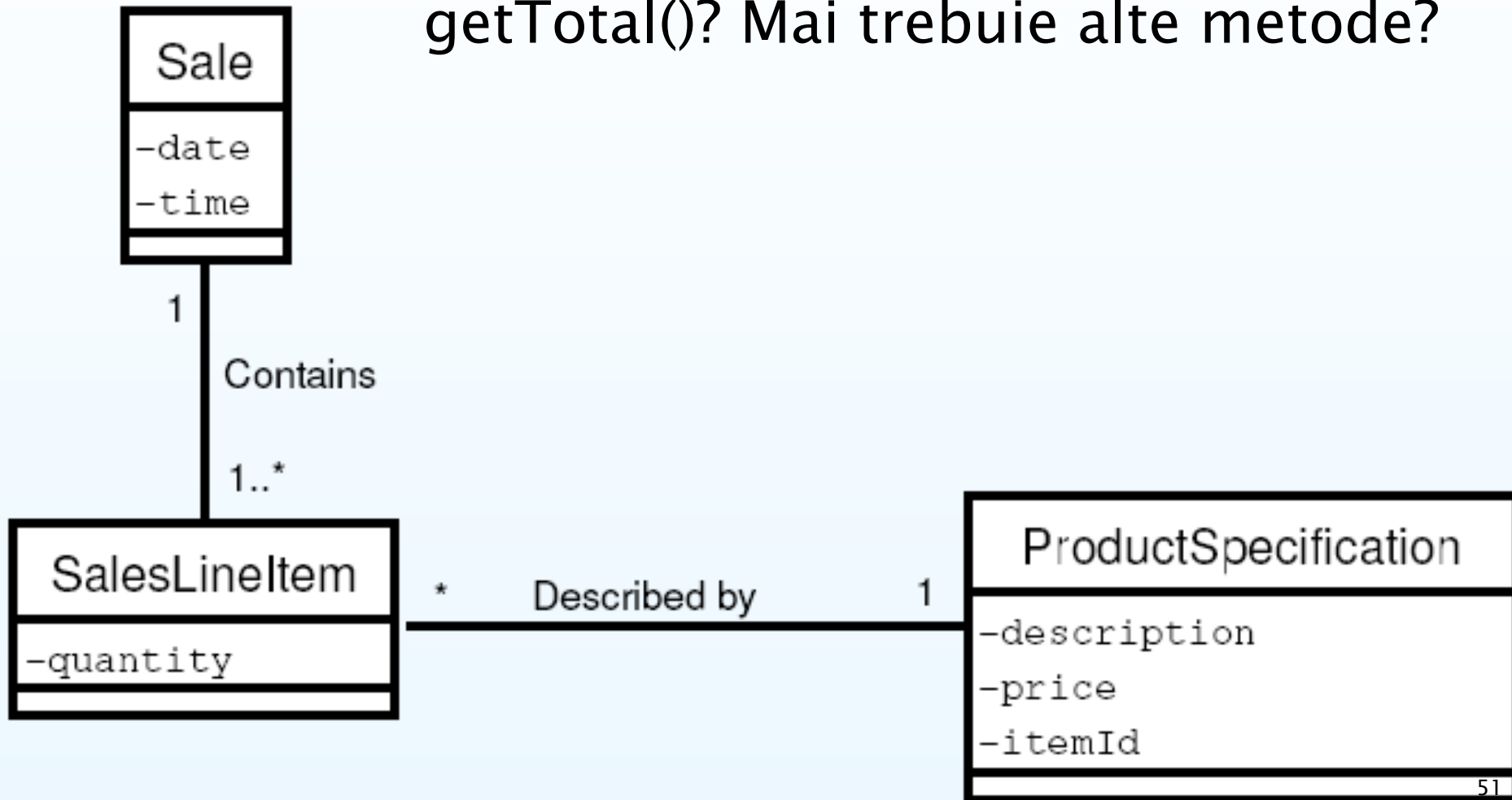
- ▶ **Problemă:** dat un anumit comportament (operație), cărei clase trebuie să-i fie atribuit?
- ▶ O alocare bună a operațiilor conduce la sisteme care sunt:
 - Ușor de înțeles
 - Mai ușor de extins
 - Refolosibile
 - Mai robuste

Information Expert 2

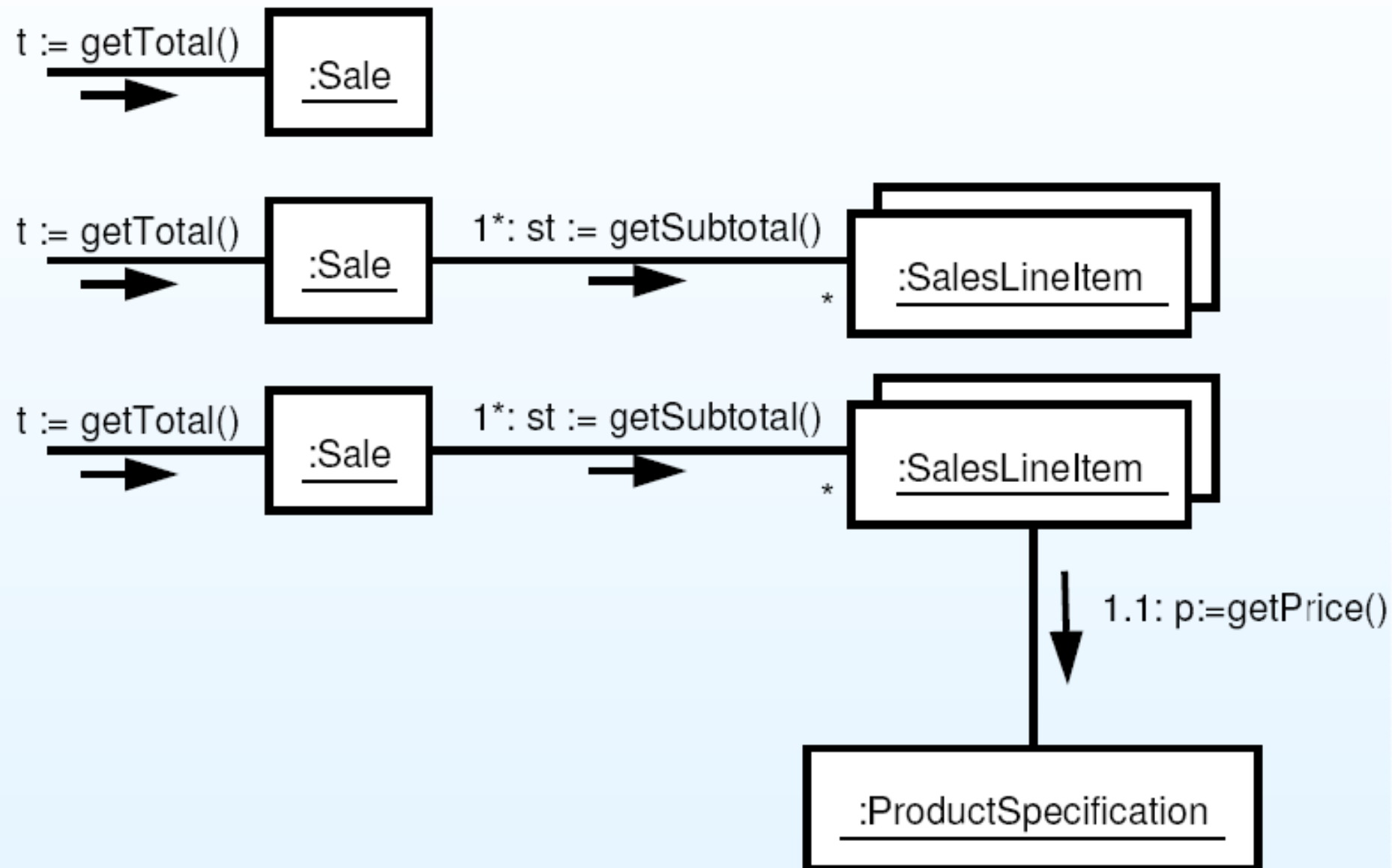
- ▶ **Soluție:**
- ▶ asignez o responsabilitate clasei care are *informațiile necesare* pentru îndeplinirea acelei responsabilități
- ▶ **Recomandare:**
- ▶ începeți asignarea responsabilităților evidențiind clar care sunt responsabilitățile

Exemplul 1

- ▶ Carei clase trebuie sa-i fie asignată metoda getTotal()? Mai trebuie alte metode?

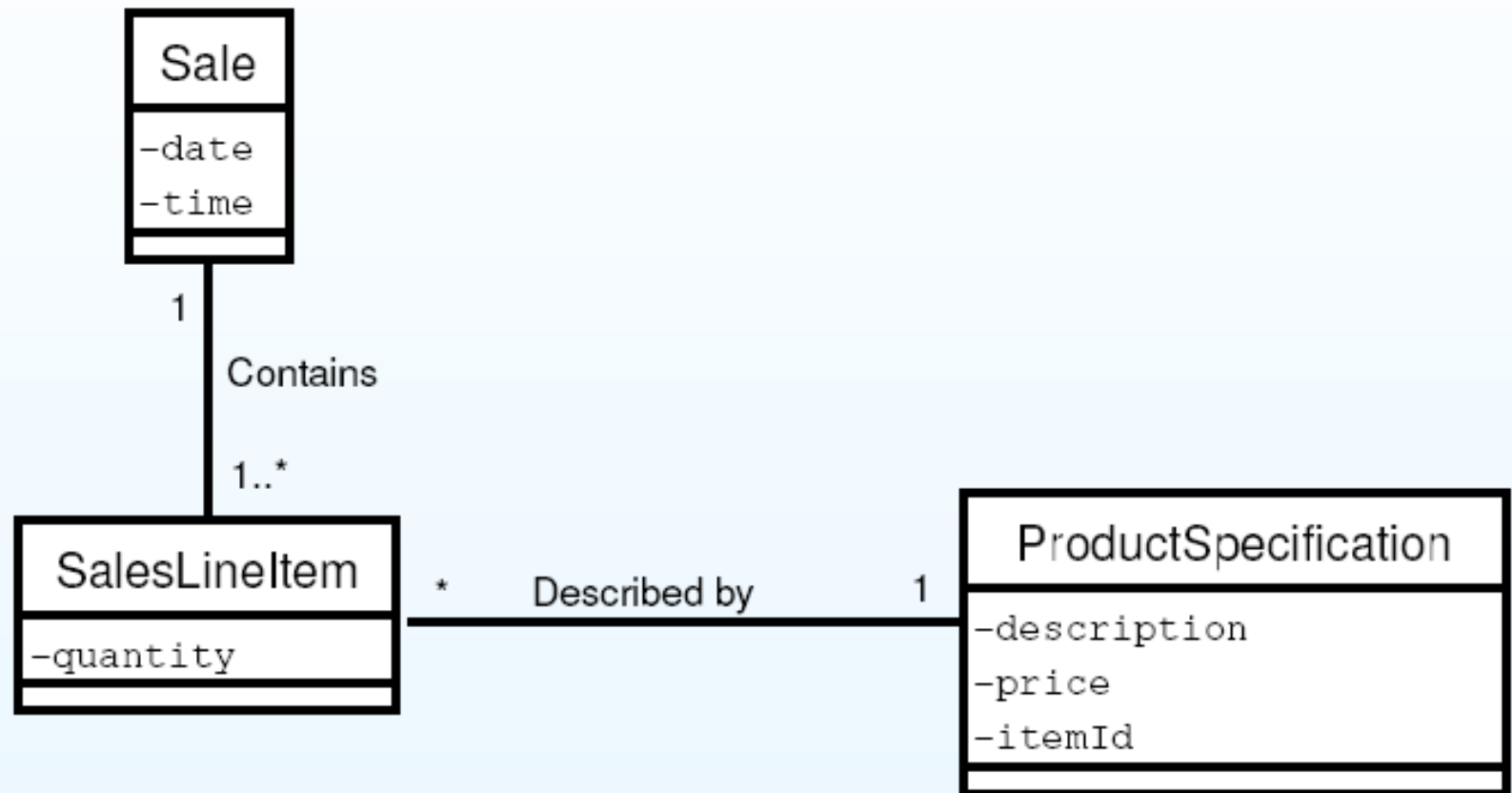


Exemplul 2

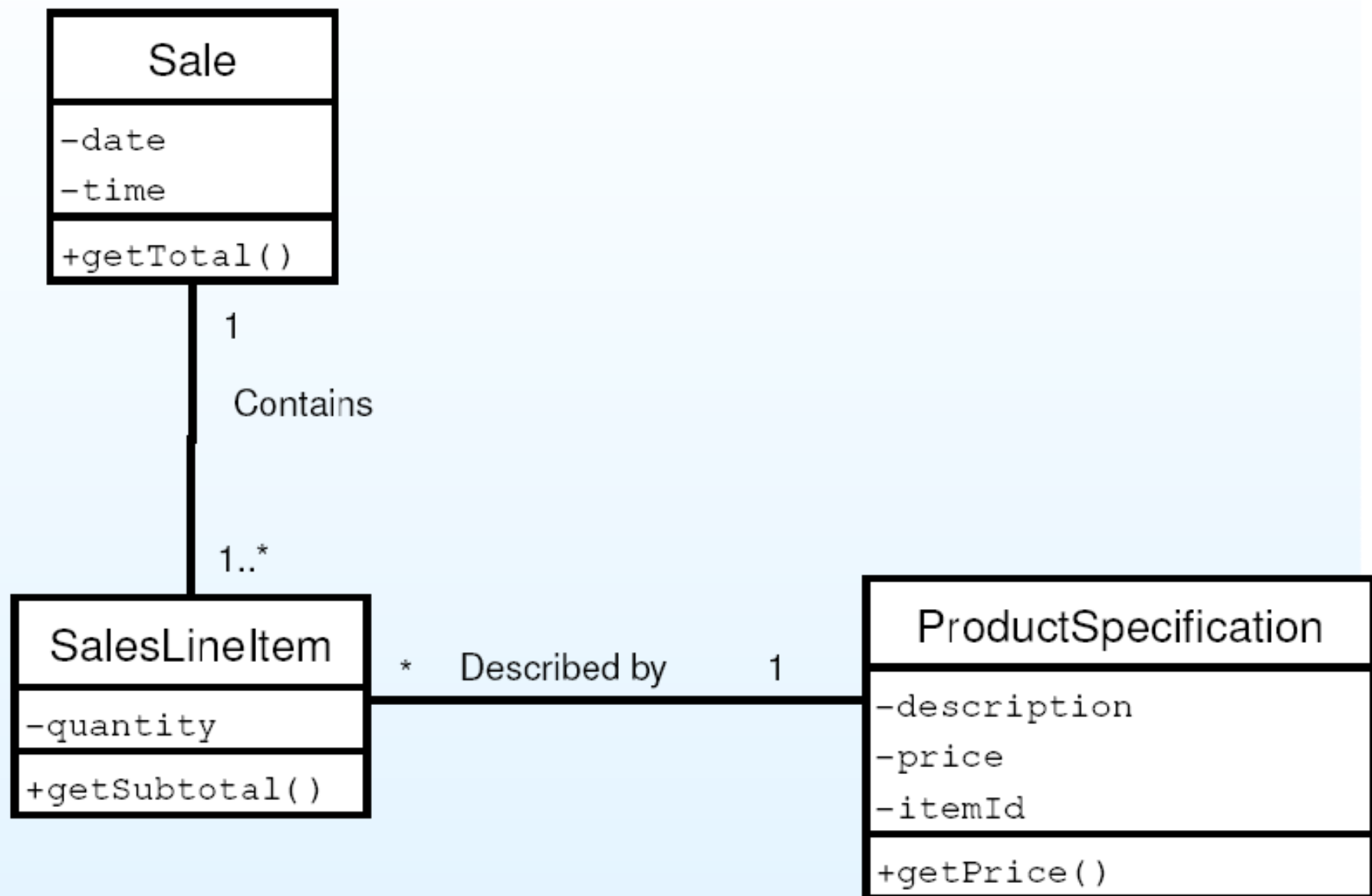


Soluție posibilă 1

Clasă	Responsabilități
Sale	să cunoască valoarea totală a cumpărăturilor
SalesLineItem	să cunoască subtotalul pentru un produs
ProductSpecification	să cunoască prețul produsului



Soluție posibilă 2

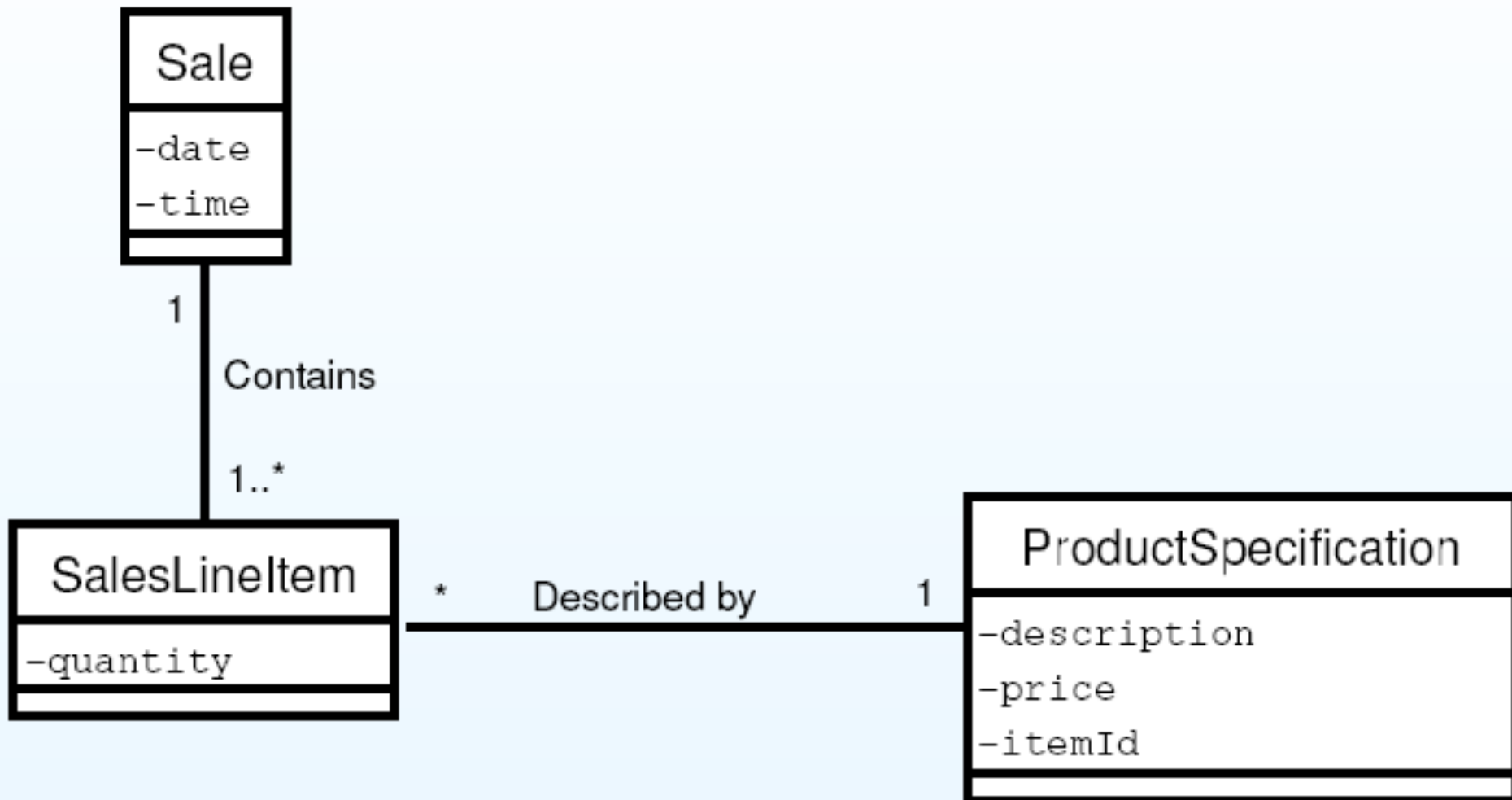


Creator 1

- ▶ **Problemă:** cine trebuie să fie responsabil cu crearea unei instanțe a unei clase?
- ▶ **Soluție:** Asignați clasei B responsabilitatea de a crea instanțe ale clasei A doar dacă cel puțin una dintre următoarele afirmații este adevărată:
 - *B agregă obiecte de tip A*
 - *B conține obiecte de tip A*
 - *B folosește obiecte de tip A*
 - *B are datele de inițializare care trebuie transmise la instanțierea unui obiect de tip A (B este deci un Expert în ceea ce privește crearea obiectelor de tip A)*
- ▶ *Factory pattern* este o variantă mai complexă

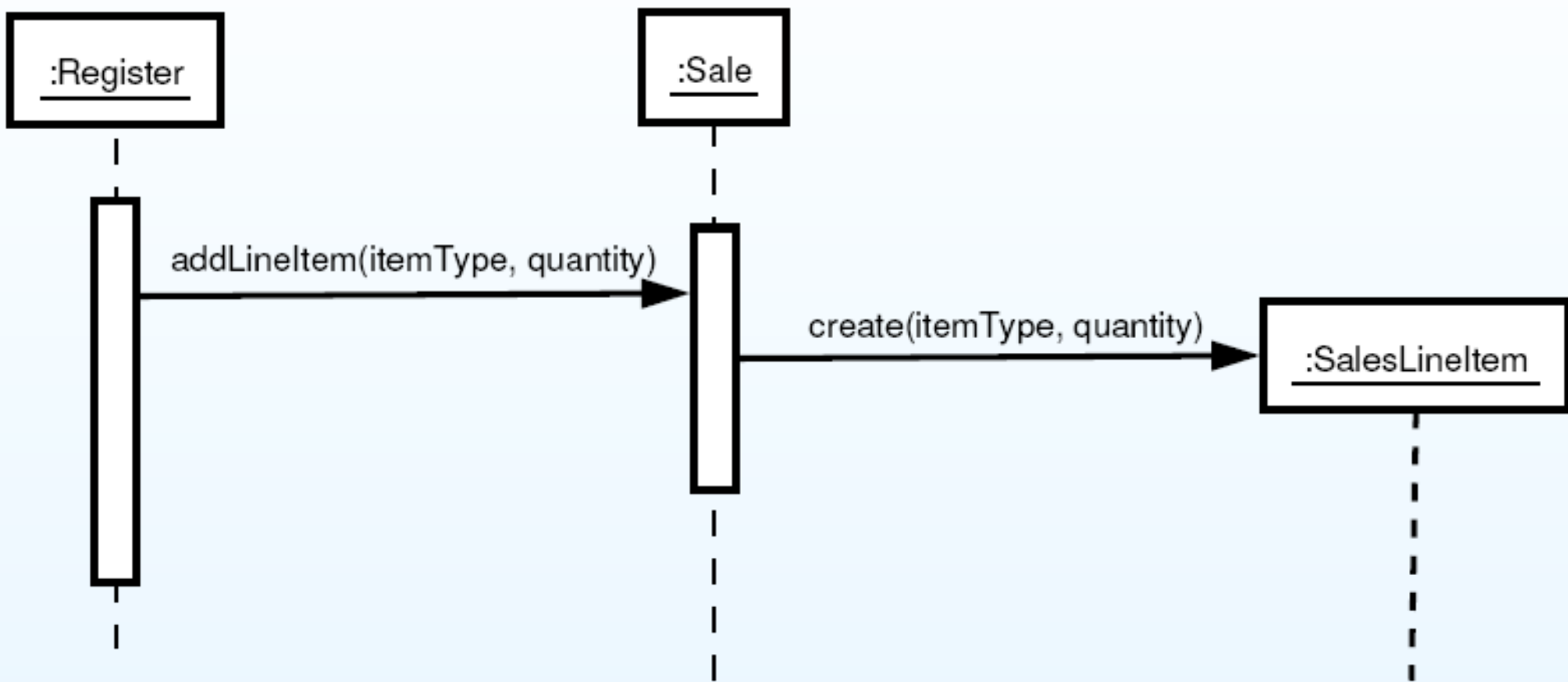
Creator 2

- ▶ Cine este responsabil cu crearea unei instanțe a clasei SalesLineItem?



Creator 3

- ▶ Deoarece Sale conține (agregă) instanțe de tip SalesLineItem, Sale este un bun candidat pentru a i se atribui responsabilitatea creării acestor instanțe



Low coupling (cuplaj redus)

- ▶ Cuplajul este o măsură a gradului de dependență a unei clase de alte clase
- ▶ Tipuri de Dependență:
 - este conectată cu
 - are cunoștințe despre
 - se bazează pe
- ▶ **O clasă care are cuplaj mic (redus) nu depinde de “multe” alte clase; unde “multe” este dependent de contex**
- ▶ O clasă care are cuplaj mare depinde de multe alte clase

Cuplaj 2

- ▶ Probleme cauzate de cuplaj:
 - schimbări în clasele relaționate forțează schimbări locale
 - clase **greu de înțeles** în izolare (scoase din context)
 - clase **greu de refolosit** deoarece folosirea lor presupune și prezența claselor de care depind

Cuplaj 3

- ▶ Forme comune de cuplaj de la clasa A la clasa B sunt:
 - A are un atribut de tip B
 - O instanță a clasei A apelează un serviciu oferit de un obiect de tip B
 - A are o metodă care referențiază B (parametru, obiect local, obiect returnat)
 - A este subclasă (direct sau indirect) a lui B
 - B este o interfață, iar A implementează această interfață

Legea lui Demeter

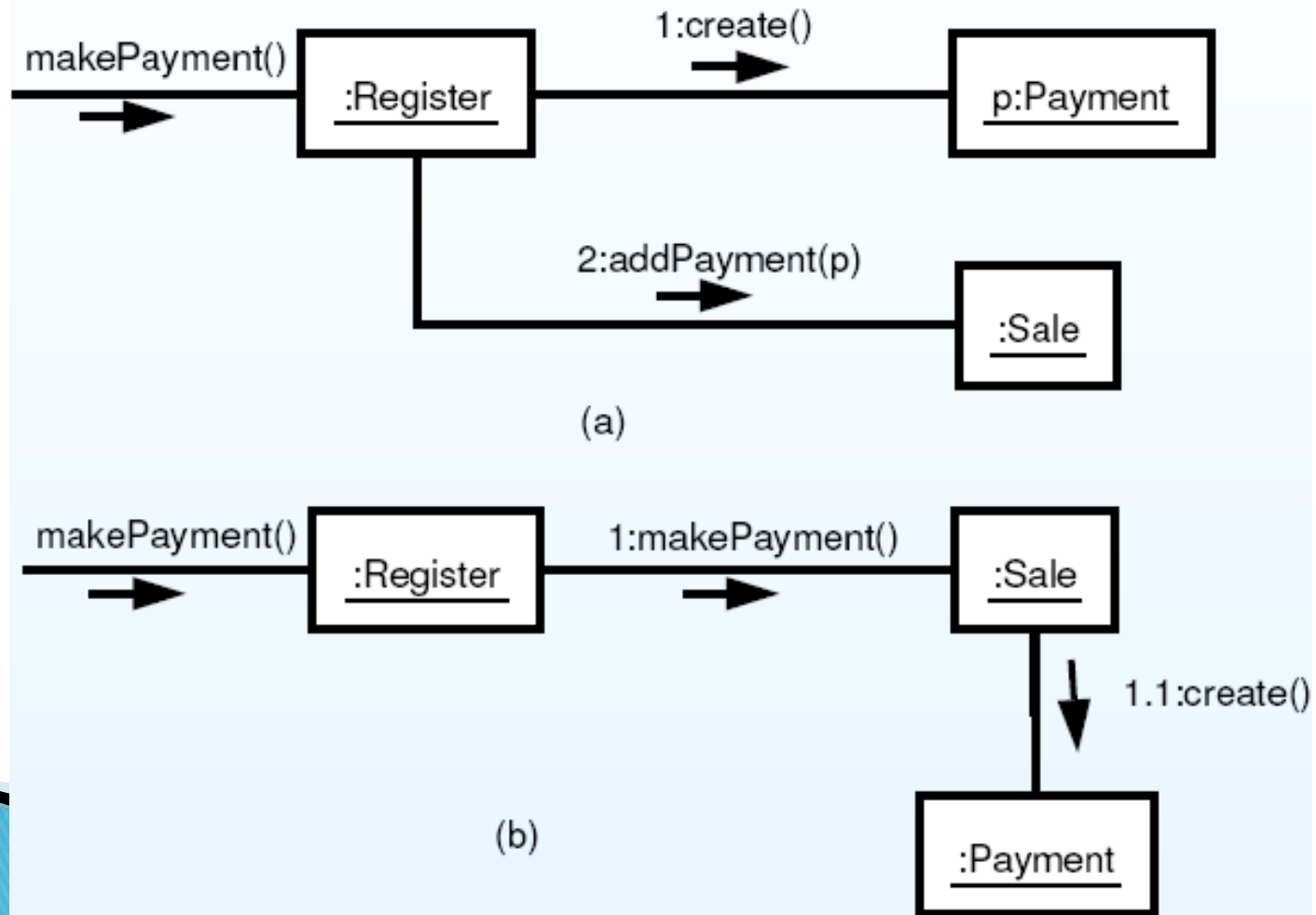
- ▶ *Don't talk to strangers*
- ▶ Orice metodă a unui obiect trebuie să apeleze doar metode aparținând
 - lui însuși
 - oricărui parametru al metodei
 - oricărui obiect pe care l-a creat
 - oricăror obiecte pe care le conține

Vizualizarea Cuplajelor

- ▶ Diagrama de clase
- ▶ Diagrama de colaborare

Exemplul 1

- ▶ Exista legături între toate clasele
- ▶ Elimină cuplajul dintre Register și Payment



High Cohesion

- ▶ **Coeziunea** este o măsură a cât de puternic sunt focalizate responsabilitățile unei clase
- ▶ O clasă ale cărei responsabilități sunt foarte strâns legate și care nu face foarte multe lucruri are o **coeziune mare**
- ▶ O clasă care face multe lucruri care nu sunt relaționate sau face prea multe lucruri are o **coeziune mică (slabă)**

Coeziune

- ▶ Probleme cauzate de o slabă coeziune:
 - greu de înțeles
 - greu de refolosit
 - greu de menținut
 - delicate; astfel de clase sunt mereu supuse la schimbări

Coeziune și Cuplaj

- ▶ Sunt principii vechi în design-ul software
- ▶ Promovează un design modular
- ▶ Modularitatea este proprietatea unui sistem care a fost descompus într-o mulțime de module coezive și slab cuplate

Controller 1

- ▶ **Problemă:** Cine este responsabil cu tratarea unui eveniment generat de un actor?
- ▶ Aceste evenimente sunt asociate cu operații ale sistemului
- ▶ Un **Controller** este un obiect care nu ține de interfața grafică și care este responsabil cu recepționarea sau gestionarea unui eveniment
- ▶ Un Controller definește o metodă corespunzătoare operației sistemului

Controller 2

- ▶ **Soluție:** asignează responsabilitatea pentru recepționarea sau gestionarea unui eveniment unei clase care reprezintă una dintre următoarele alegeri:
 - Reprezintă întregul sistem sau subsistem (fațadă controller)
 - Reprezintă un scenariu de utilizare în care apare evenimentul;

Controller 3

- ▶ În mod normal, un controller ar trebui să delege altor obiecte munca care trebuie făcută;
- ▶ **Controller-ul coordonează sau controlează activitatea, însă nu face prea multe lucruri el însuși**
- ▶ O greșeală comună în design-ul unui controller este să i se atribuie prea multe responsabilități (fațade controller)

Concluzii

- ▶ **Diagrame UML**
 - Diagrame de Stări, Diagrame de Activități
 - Diagrame de Deployment, Diagrame de Pachete
- ▶ **GRASP**
 - Information Expert
 - Creator
 - Low coupling
 - High cohesion
 - Controller

Bibliografie

- ▶ Craig Larman. *Applying UML and Patterns. An Introduction to Object Oriented Analysis and Design*
- ▶ Ovidiu Gheorghieș, Curs 6 IP

Links

- ▶ WebProjectManager: <http://profs.info.uaic.ro/~adrianaa/uml/>
- ▶ Diagrame de Stare și de Activitate:
http://software.ucv.ro/~soimu_anca/itpm/Diagrame%20de%20Stare%20si%20Activitate.doc
- ▶ Deployment Diagram:
http://en.wikipedia.org/wiki/Deployment_diagram
<http://www.agilemodeling.com/artifacts/deploymentDiagram.htm>
- ▶ GRASP:
[http://en.wikipedia.org/wiki/GRASP_\(Object_Oriented_Design\)](http://en.wikipedia.org/wiki/GRASP_(Object_Oriented_Design))
- ▶ <http://web.cs.wpi.edu/~gpollice/cs4233-a05/CourseNotes/maps/class4/GRASPpatterns.html>
- ▶ Introduction to GRASP Patterns:
[http://faculty.inverhills.edu/dlevitt/CS%202000%20\(FP\)/GRASP%20Patterns.pdf](http://faculty.inverhills.edu/dlevitt/CS%202000%20(FP)/GRASP%20Patterns.pdf)