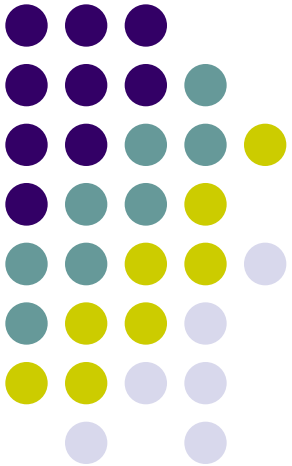
Ingineria programării

2. Limbajul unificat de modelare, UML



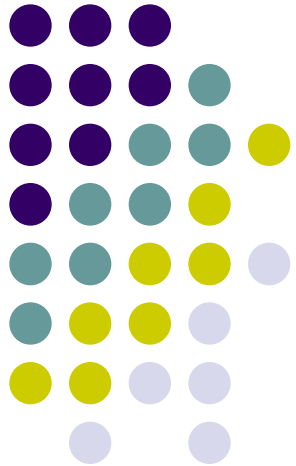
Florin Leon

Universitatea Tehnică „Gheorghe Asachi” din Iași
Facultatea de Automatică și Calculatoare

http://florinleon.byethost24.com/curs_ip.htm

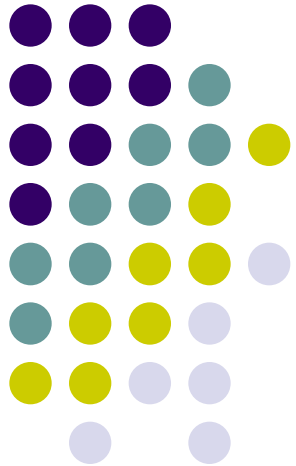
Limbajul unificat de modelare, UML

1. Modelarea
2. Limbajul unificat de modelare
3. Clasificarea diagramelor UML 2.0
4. Diagramele UML 2.0
5. Concluzii



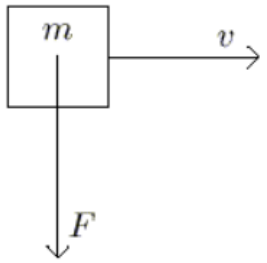
Limbajul unificat de modelare, UML

1. Modelarea
2. Limbajul unificat de modelare
3. Clasificarea diagramelor UML 2.0
4. Diagramele UML 2.0
5. Concluzii



Modelarea

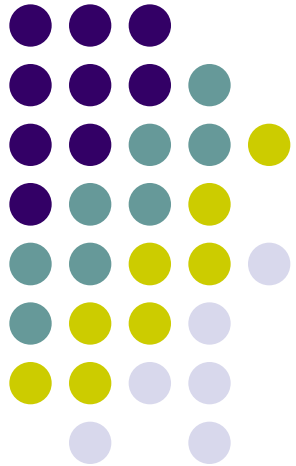
- Un model este o simplificare a unui anumit sistem, care permite analizarea unora dintre proprietățile acestuia
  - Reține caracteristicile necesare
- Folosirea de modele facilitează abordarea problemelor complexe, comunicarea și înțelegerea
  - Divide et impera
- Exemple:
  - Formalismul matematic
  - Reprezentările din fizică



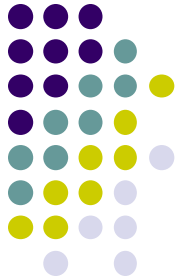
- Orice limbaj „intern” poate fi folosit pentru modelare, însă într-un context formal este nevoie de standardizare

Limbajul unificat de modelare, UML

1. Modelarea
2. Limbajul unificat de modelare
3. Clasificarea diagramelor UML 2.0
4. Diagramele UML 2.0
5. Concluzii



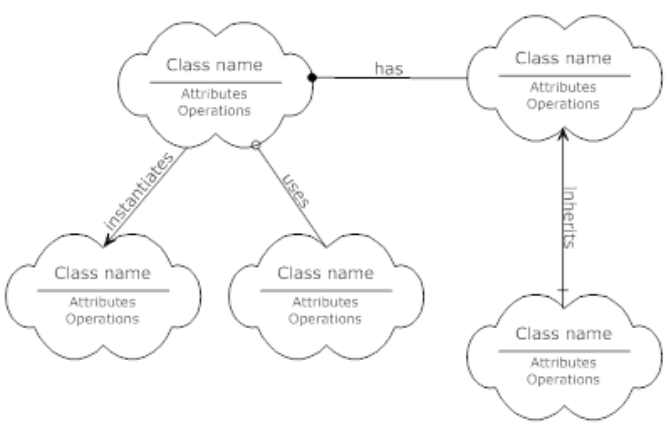
Scurt istoric



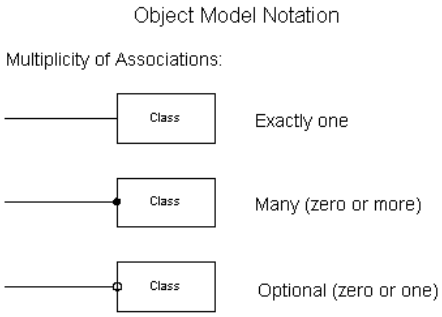
- Între 1989 și 1994 erau folosite mai mult de 50 de limbaje de modelare software, fiecare cu propriile notații
- Utilizatorii doreau un limbaj standardizat, o *lingua franca* a modelării
- La mijlocul anilor ’90 trei metode s-au dovedit mai eficiente:
  - Booch (Grady Booch): potrivită mai ales pentru proiectare și implementare, cu dezavantajul unor notații complicate
  - OMT, *Object Modeling Technique* (Jim Rumbaugh): potrivită pentru analiză și sisteme informatice cu multe date
  - OOSE, *Object Oriented Software Engineering* (Ivar Jacobson): această metodă a propus așa-numitele cazuri de utilizare, care ajută la înțelegerea comportamentului sistemului în ansamblu

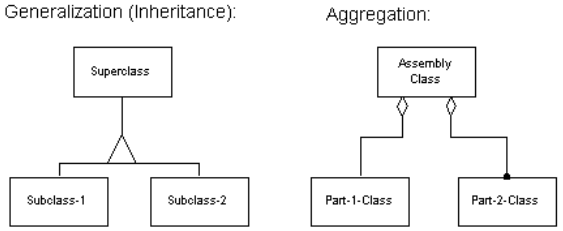
Precursorii UML

- Booch

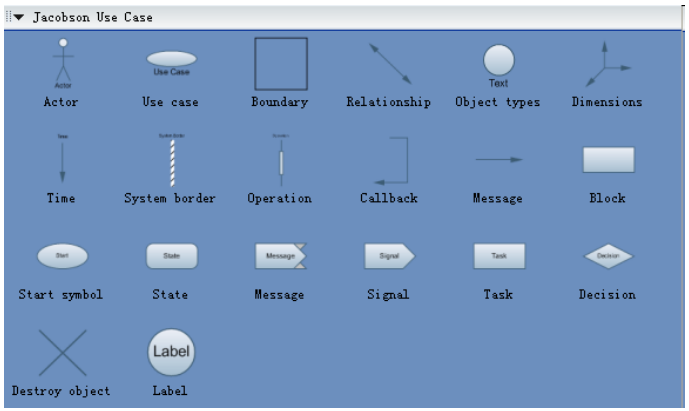


- OMT

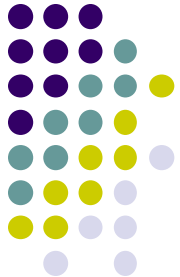




- OOSE



Scurt istoric

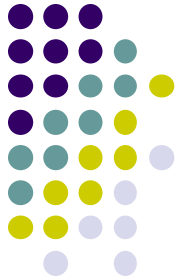


- 1994: Jim Rumbaugh, creatorul OMT, a părăsit General Electric, alăturându-se lui Grady Booch la Rational Corp.
- 1995: Ivar Jacobson, creatorul OOSE, a venit la Rational iar ideile lui, în special conceptul de cazuri de utilizare, au fost adăugate „Metodei unificate”
- Metoda rezultată a fost numită „Limbajul unificat de modelare”, UML
- 1996: Formarea de către Rational a consorțiului „Partenerilor UML” din care făceau parte giganți precum Hewlett-Packard, Microsoft și Oracle

UML

- Limbaj pentru specificarea, vizualizarea, construirea și documentarea elementelor sistemelor software
  - Un limbaj grafic care ne permite să reproducem „pe hârtie” ceea ce este produs în procesul de dezvoltare a unui sistem software
  - Poate fi folosit și pentru alte sisteme, cum ar fi procesele comerciale (engl. “business processes”)

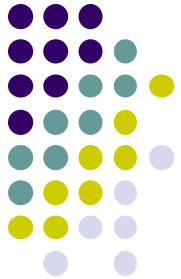
Versiuni și standardizare



- Ianuarie 1997: a fost propus spre standardizare UML 1.0 în cadrul OMG (Object Management Group)
- Noiembrie 1997: a fost adoptată versiunea UML 1.1 ca standard de către OMG
- Martie 2003: a fost lansată versiunea UML 1.5
- Octombrie 2004: versiunea UML 2.0
- Iunie 2015: versiunea UML 2.5

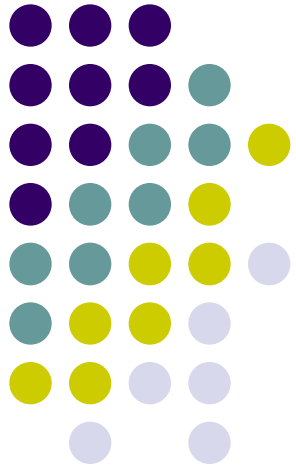
- UML este standardul ISO/IEC 19501:2005

UML

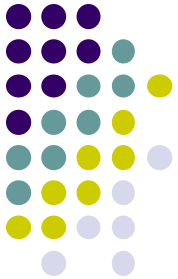


- Ca orice limbaj, UML are:
  - Notații (alfabetul de simboluri)
  - Sintaxă și gramatică (reguli pentru combinarea simbolurilor)
- UML este un instrument de comunicare
- UML nu este o metodologie de dezvoltare
  - Dar este determinat de cele mai bune practici în domeniu

Limbajul unificat de modelare, UML

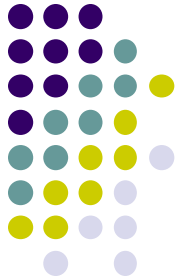


Clase de diagrame



- **Diagrame de structură**
  - Prezintă elementele unei specificații independent de timp
  - Includ: diagramele de clase, componente, desfășurare (deployment), obiecte, pachete și structuri compuse
- **Diagrame de comportament**
  - Prezintă trăsăturile comportamentale ale sistemului
  - Includ: diagramele de activități, mașini de stare și cazuri de utilizare, precum și cele 4 diagrame de interacțiune
  - **Diagrame de interacțiune**
    - Scot în evidență interacțiunile dintre obiecte
    - Includ: diagramele de secvențe, comunicare, interacțiuni generale (interaction overview) și cronometrare (timing)
  - *Legendă: utilitate practică mare, medie, mică*

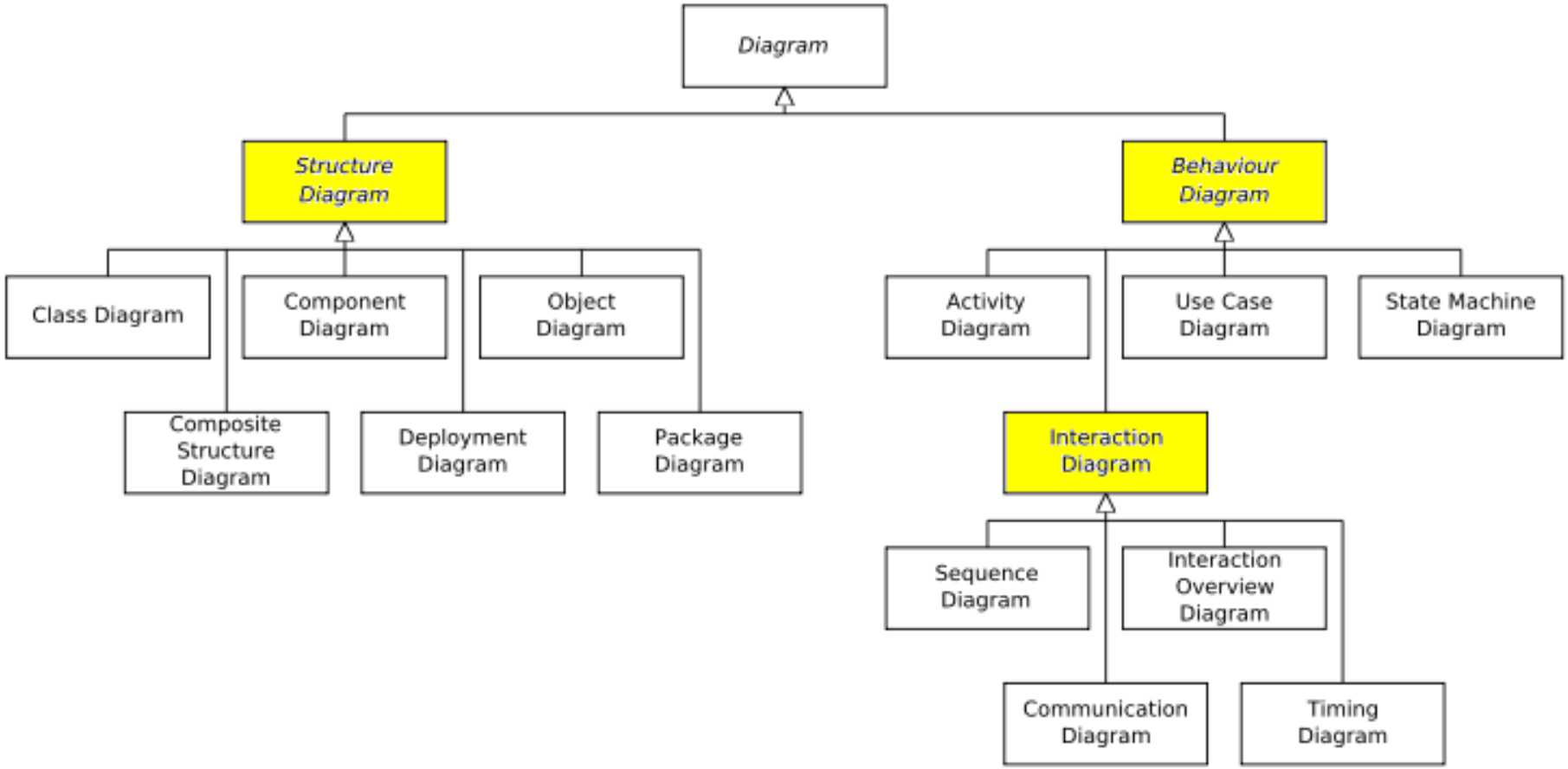
Diagramele UML 2.0



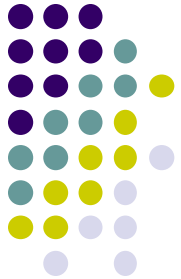
- Diagrame de structură:
  ce conține sistemul
  - Clase
  - Componente
  - Desfășurare
  - Obiecte
  - *Pachete*
  - *Structuri compuse*
- Diagrame de comportament:
  ce se întâmplă în sistem
  - Activități
  - Mașini de stare
  - Cazuri de utilizare
- Diagrame de interacțiune:
  fluxurile de control și date dintre componentele sistemului
  - Secvențe
  - Comunicare
  - *Interacțiuni generale*
  - *Cronometrare*

Diagrame introduse în UML 2.0

Diagramele UML 2.0

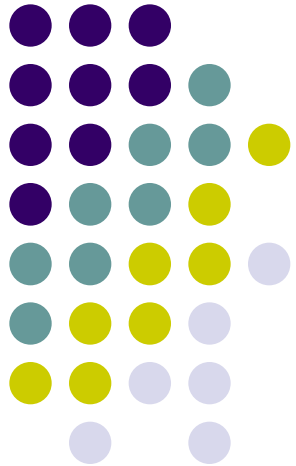


OCL și XMI



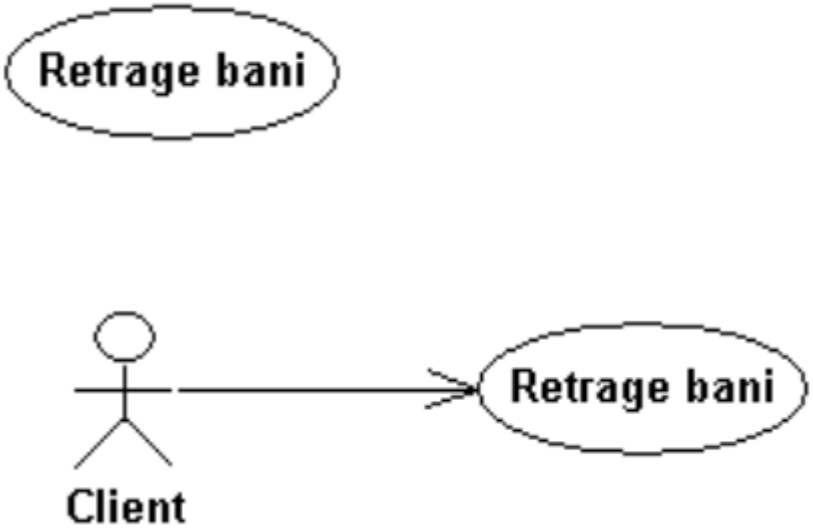
- Object Constraint Language (OCL)
  - Pe lângă diagrame, UML mai dispune de un limbaj suplimentar pentru descrierea unor reguli care se aplică elementelor din model, de exemplu constrângeri
- XML Metadata Interchange (XMI)
  - Standard pentru transferul de metadate între diferite instrumente de modelare UML
  - Mai simplu, un limbaj în care se pot salva modele UML, independent de instrumentele de modelare folosite

Limbajul unificat de modelare, UML

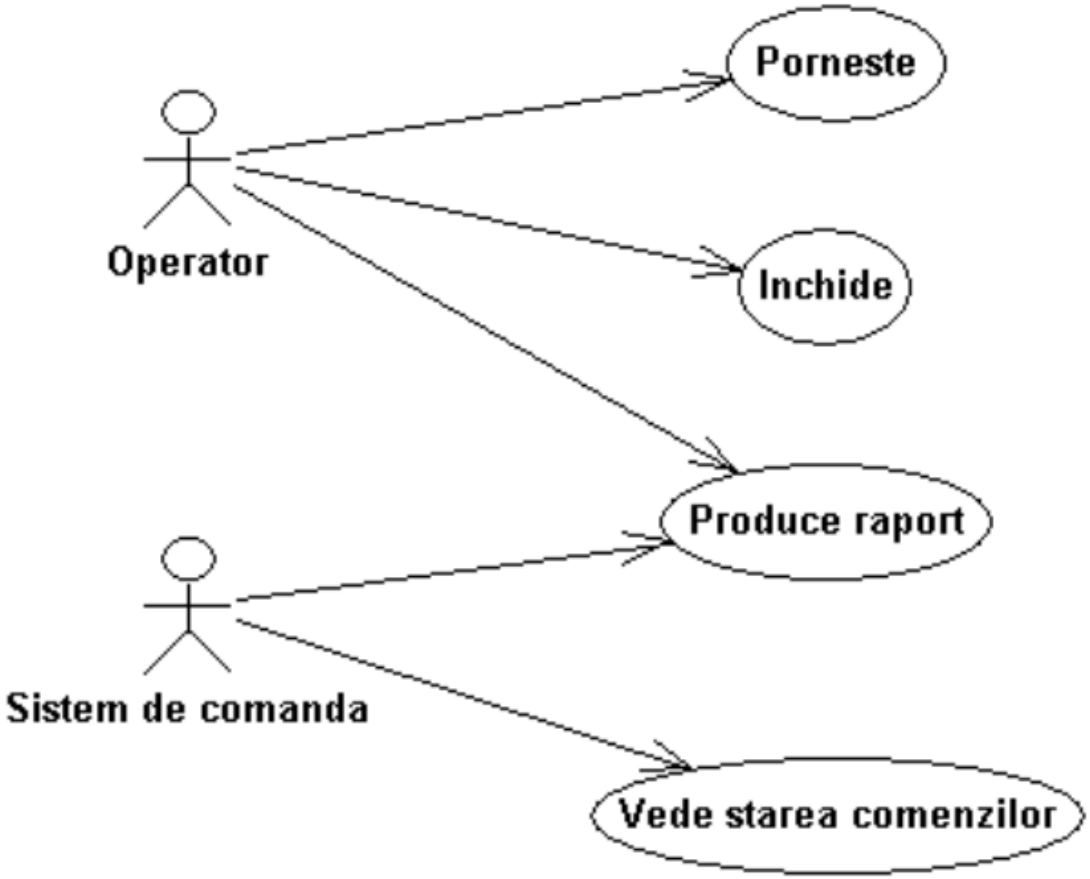


1. Diagrama cazurilor de utilizare

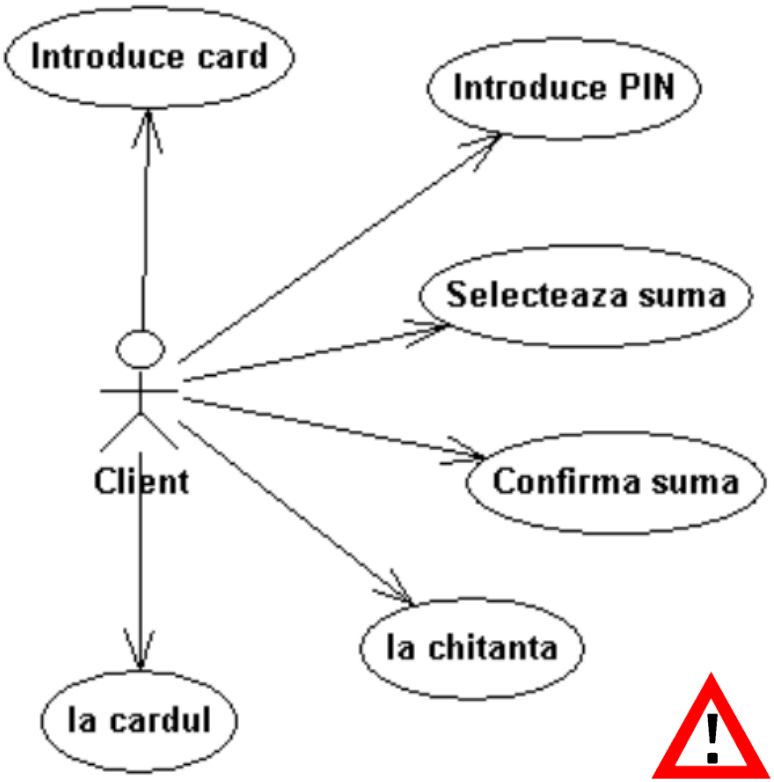
- Descrie interacțiunile dintre utilizatori și sistem

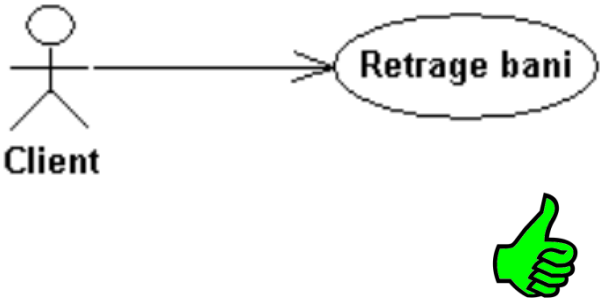


Actori și cazuri multiple



Granularitatea



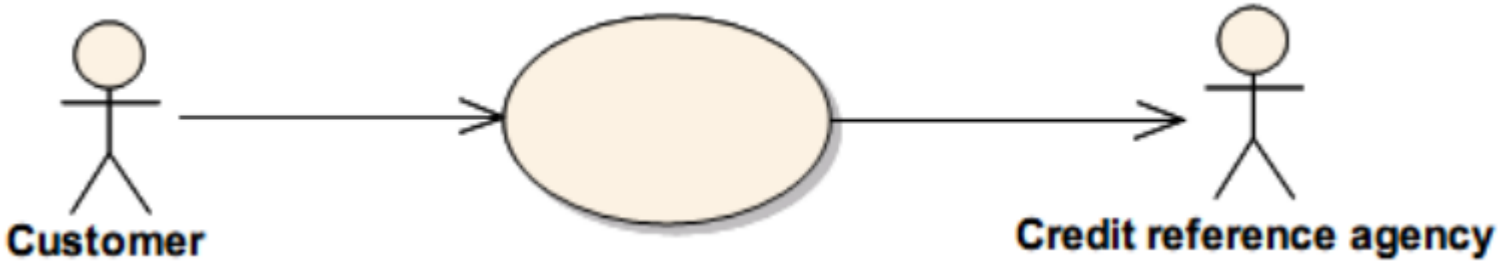


Un caz de utilizare trebuie să satisfacă un scop pentru actor

Avantaje

- Suma cazurilor de utilizare este întregul sistem
- Permit comunicarea cu persoane fără cunoștințe tehnice IT
- Partiționează funcționalitatea, ghidează dezvoltarea iterativă
- Ajută planificarea și testarea
- Ajută la crearea manualelor de utilizare

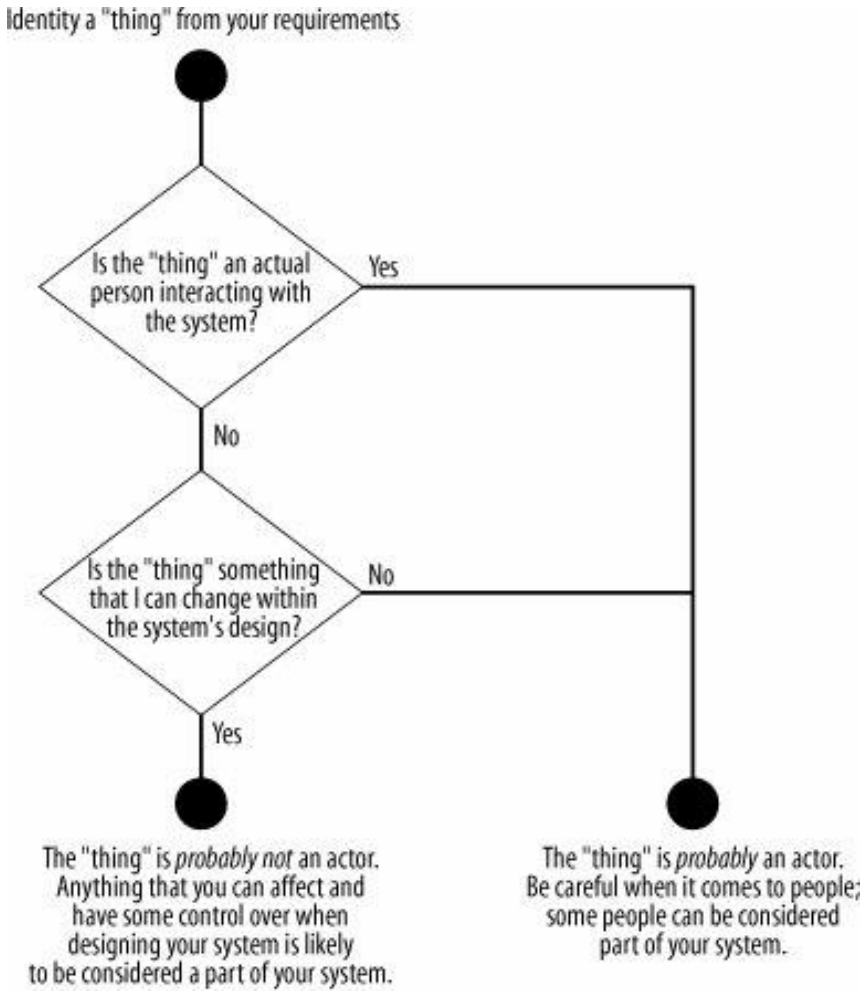
Actori primari și secundari



beneficiază

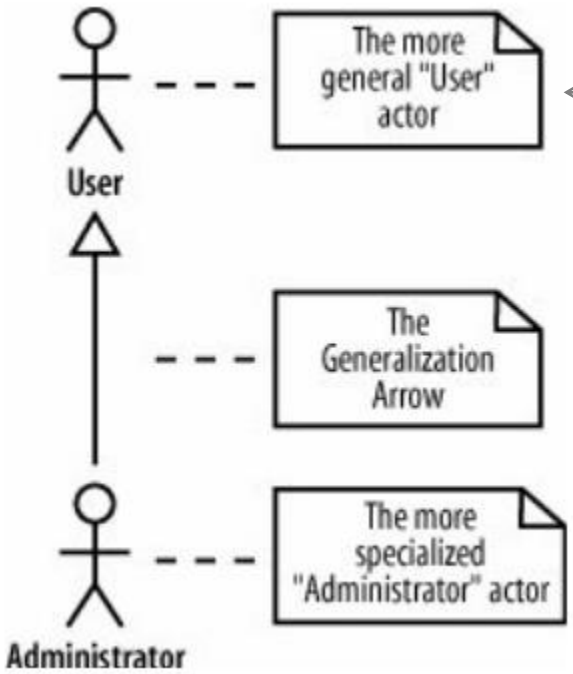
ia parte

Identificarea actorilor



Exemplu: timpul poate fi un actor

Generalizarea actorilor

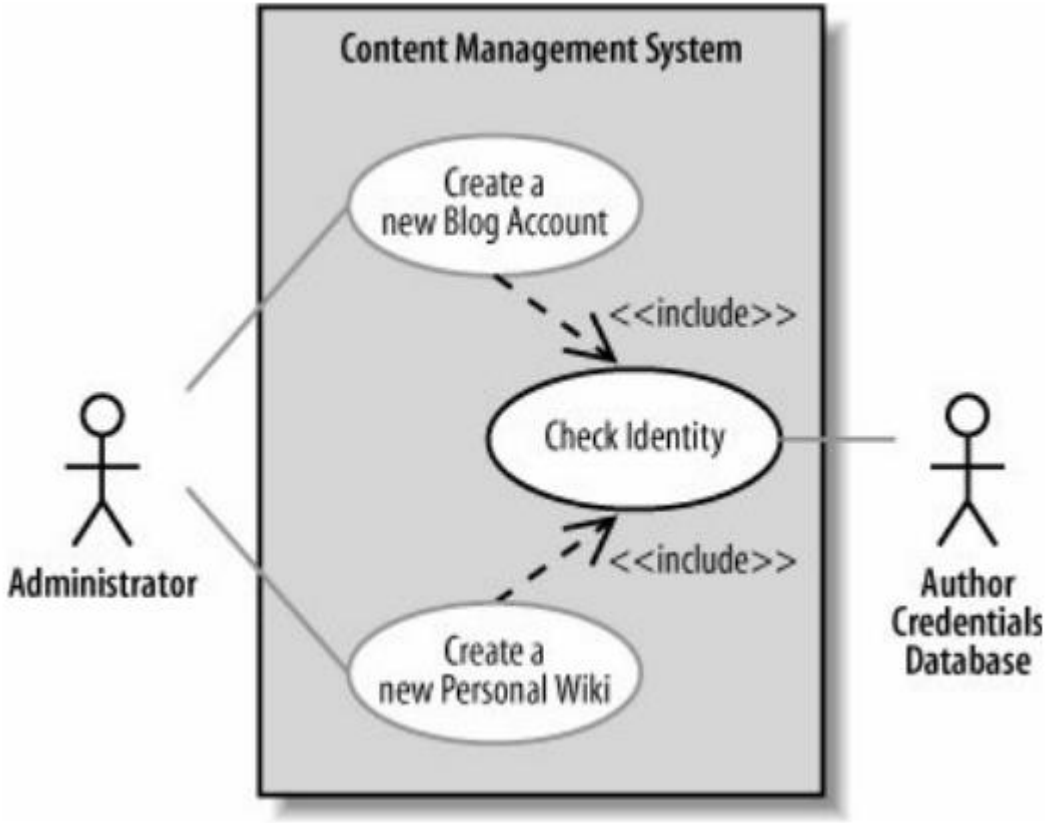


„note” (comentarii)

Administratorul poate face tot ce face și utilizatorul, dar și ceva în plus

Relațiile dintre cazurile de utilizare

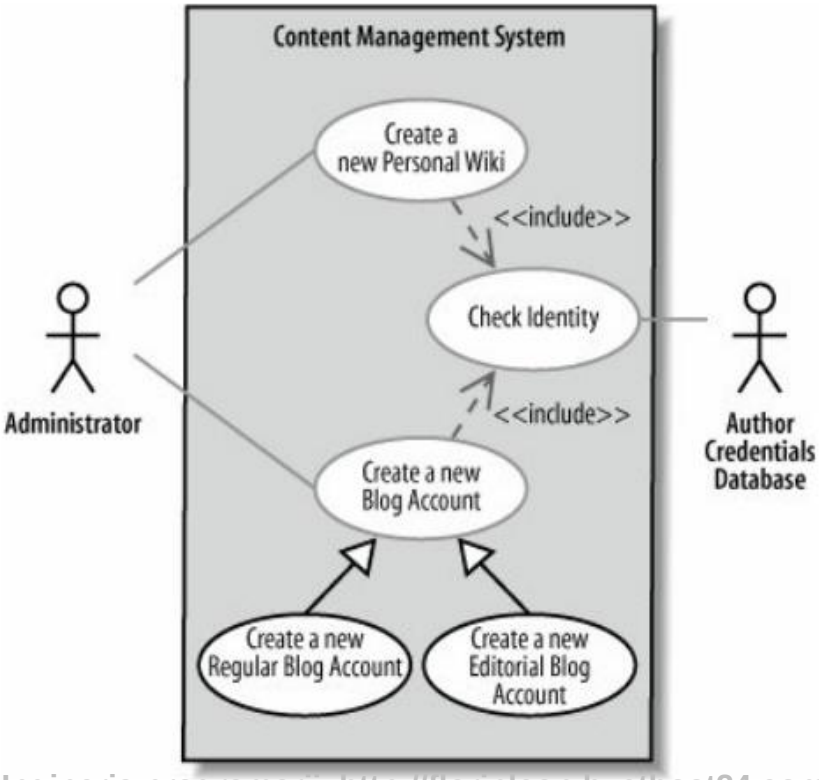
- Reutilizare: *include*



Doar *Check Identity* este conectat cu *Author Credentials Database*

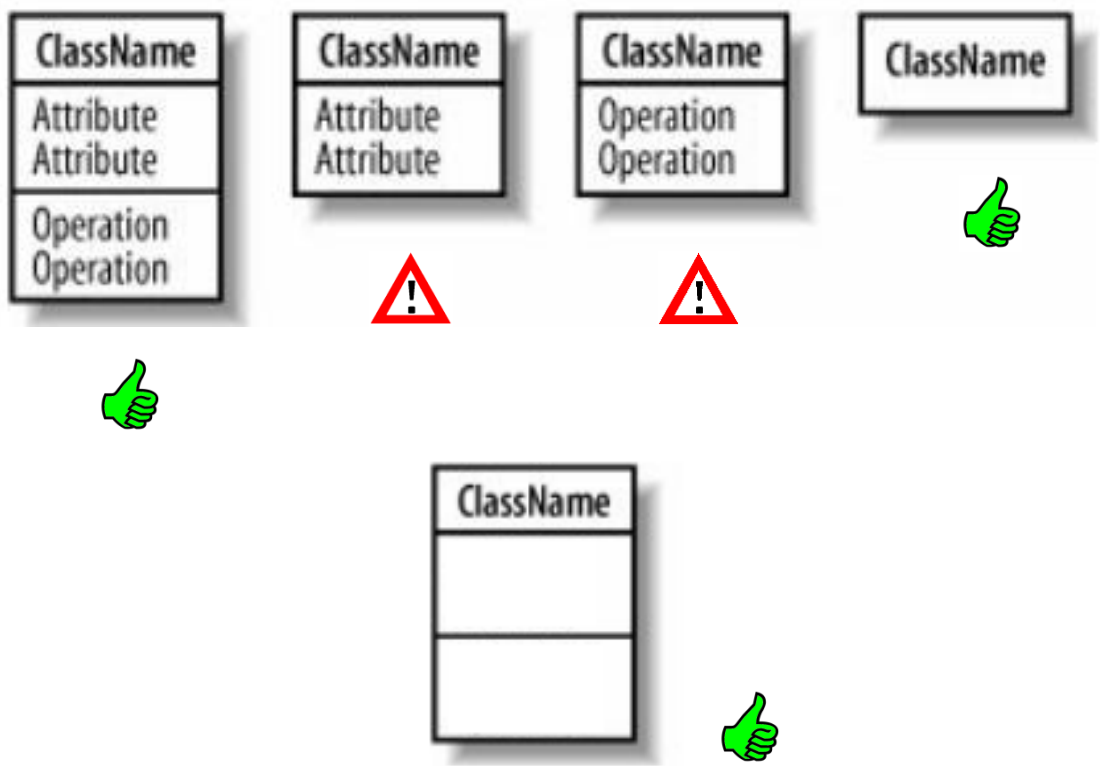
Relațiile dintre cazurile de utilizare

- Generalizarea cazurilor de utilizare: reutilizare cu modificări



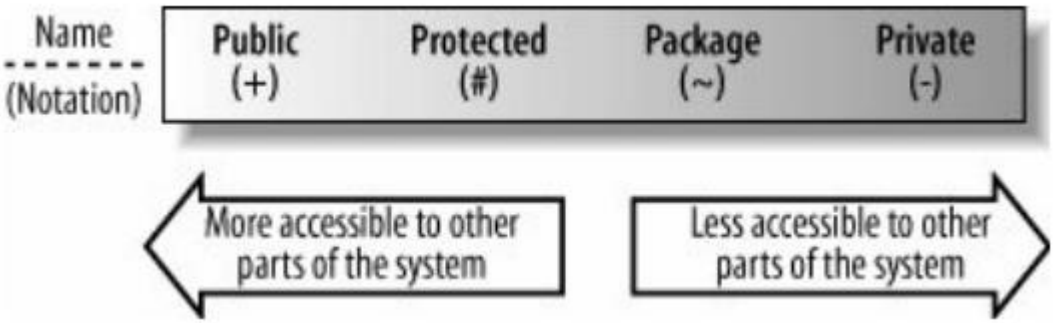
2. Diagrama de clase

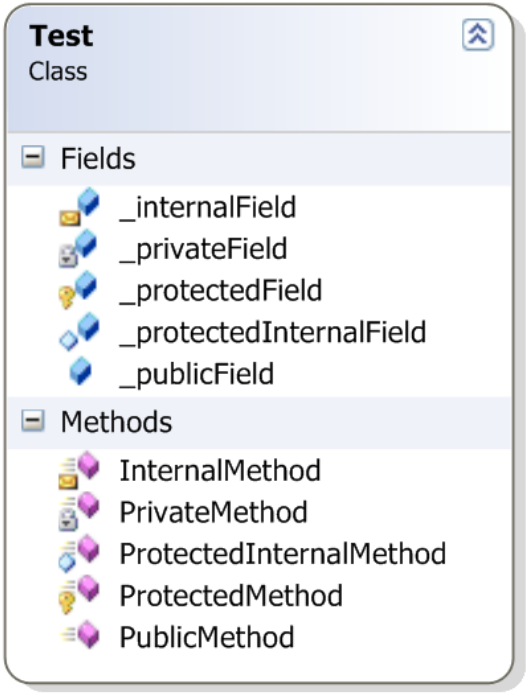
- Clasa



Vizibilitatea trăsăturilor

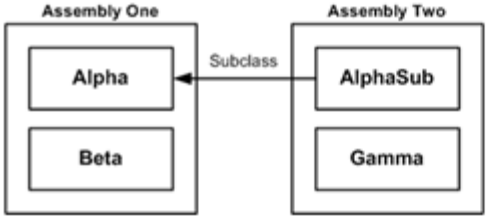
- Public
- Protejat
- Pachet
- Privat





Vizibilitatea în C# și Java

C#:

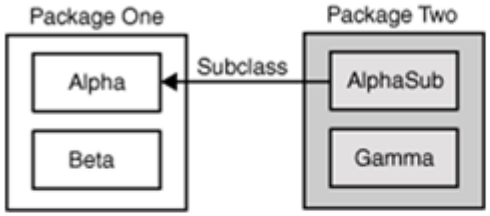


Visibility of the members of **Alpha**:

Modifier	Alpha	Beta	AlphaSub	Gamma
public	Y	Y	Y	Y
protected internal	Y	Y	Y	N
internal	Y	Y	N	N
protected	Y	N	Y	N
private *	Y	N	N	N

(* *private* is the default access-level for members if nothing is specified)

Java:



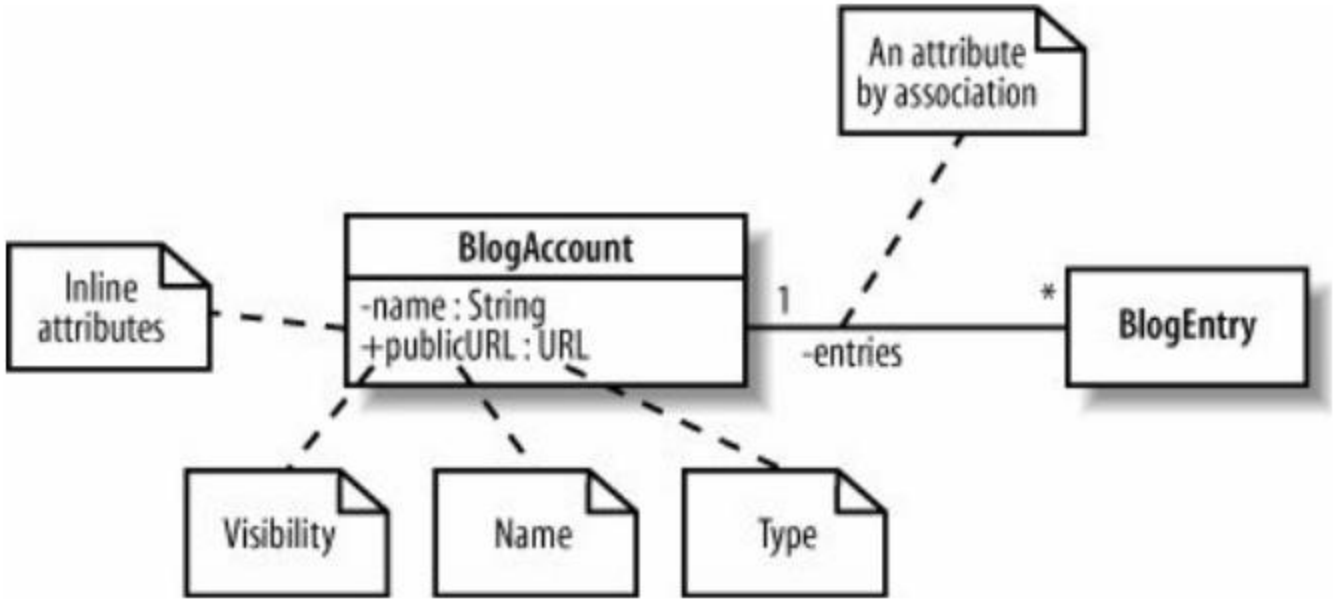
Visibility of the members of **Alpha**:

Modifier	Alpha	Beta	AlphaSub	Gamma
public	Y	Y	Y	Y
protected	Y	Y	Y	N
*no modifier**	Y	Y	N	N
private	Y	N	N	N

(* also called *package private*)

Reprezentarea atributelor

- *Inline* (în interiorul clasei)
- Prin asociere

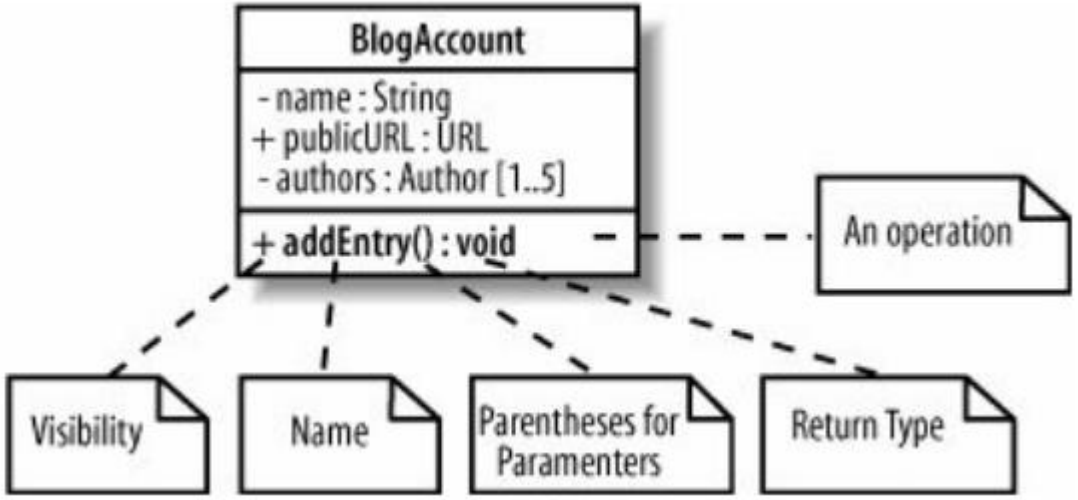


Vizibilitatea atributelor

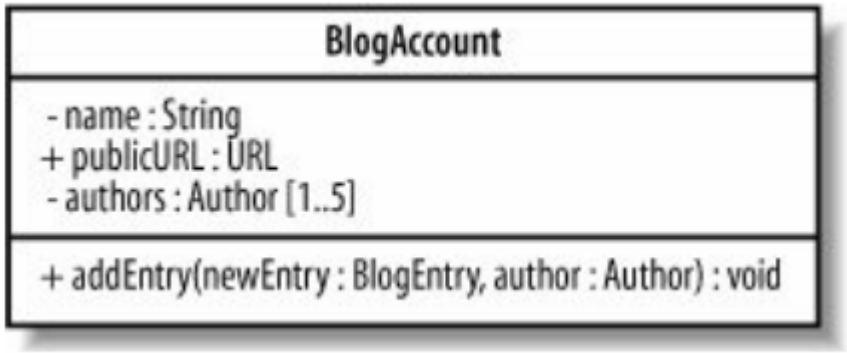
- De obicei private sau protejate
- Nu se recomandă vizibilitatea publică, decât:
  - Pentru constante
  - Pentru atribute *read-only*

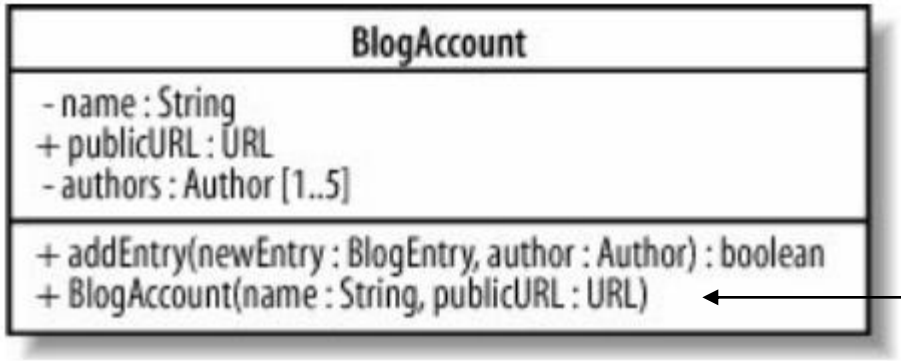

```
public class ContentManagementSystem
{
    private readonly string createdBy = "Adam Cook Software Corp.";
}
```

Operațiile


BlogAccount
- name : String
+ publicURL : URL
- authors : Author [1..5]
+ addEntry() : void
An operation
Visibility
Name
Parentheses for Paramenters
Return Type

Parametrii și tipurile de return

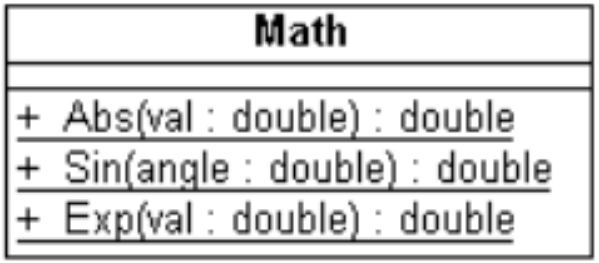




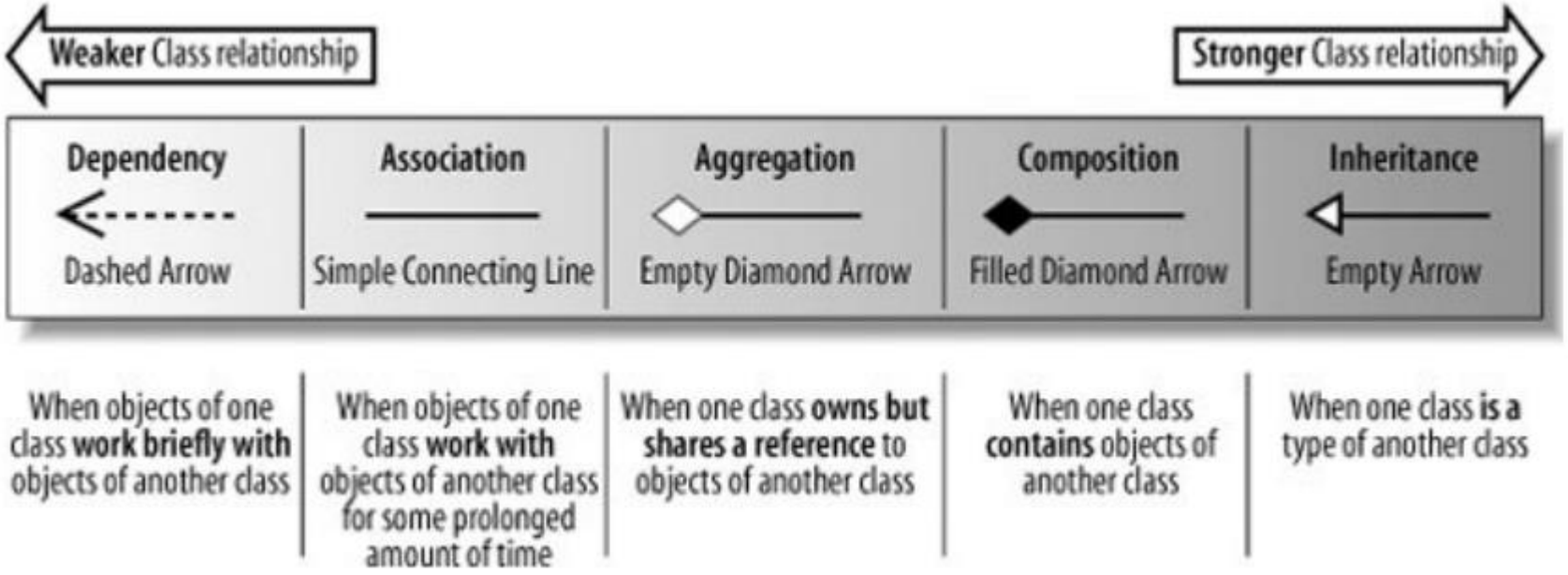
Constructor

Trăsături statice

- Trăsături (*features*) = atribute și operații
- Trăsăturile statice se subliniază

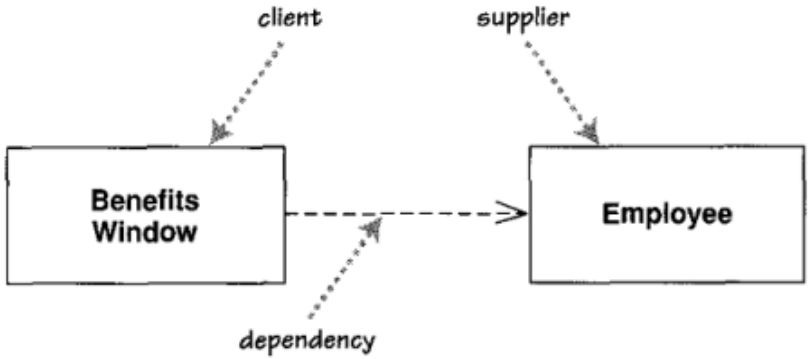


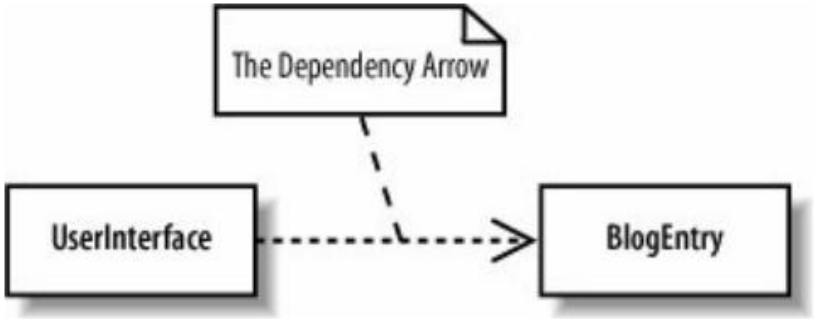
Relații între clase



Dependența

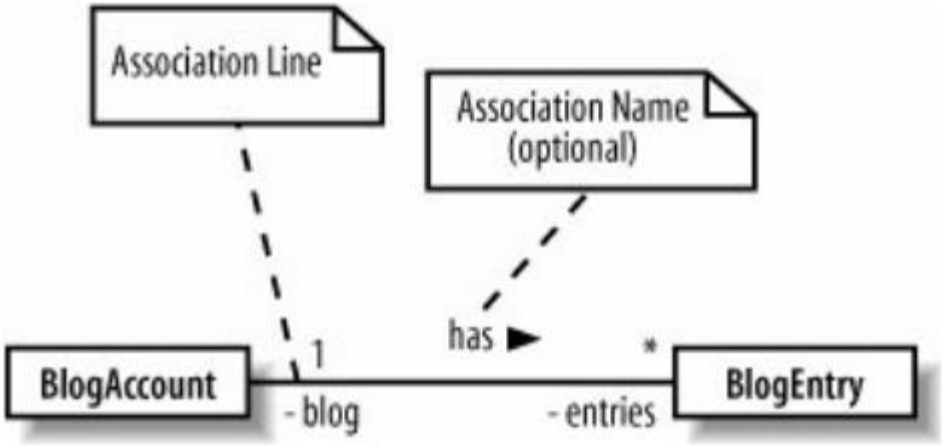
- O clasă folosește pentru scurt timp o altă clasă
  - Trimiterea unui mesaj, de exemplu apelul unei metode din clasa *Math*
  - Instanțierea unei clase într-o metodă
  - Primirea unui obiect ca parametru într-o metodă
  - Crearea și returnarea unui obiect dintr-o metodă





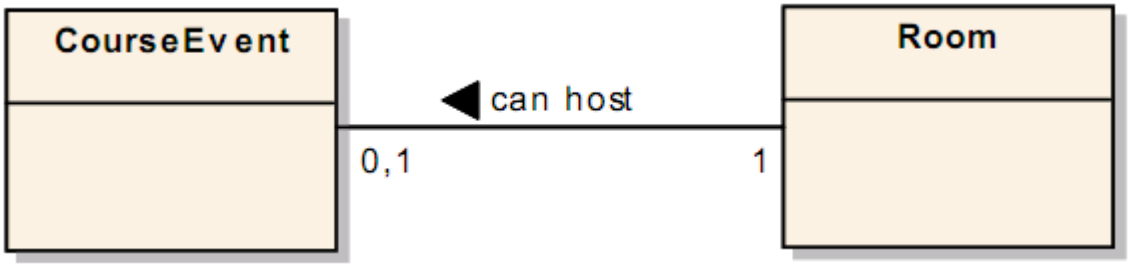
Asocierea

- O clasă folosește un timp îndelungat o altă clasă
  - De obicei, o clasă are un câmp instanțiat din cealaltă clasă

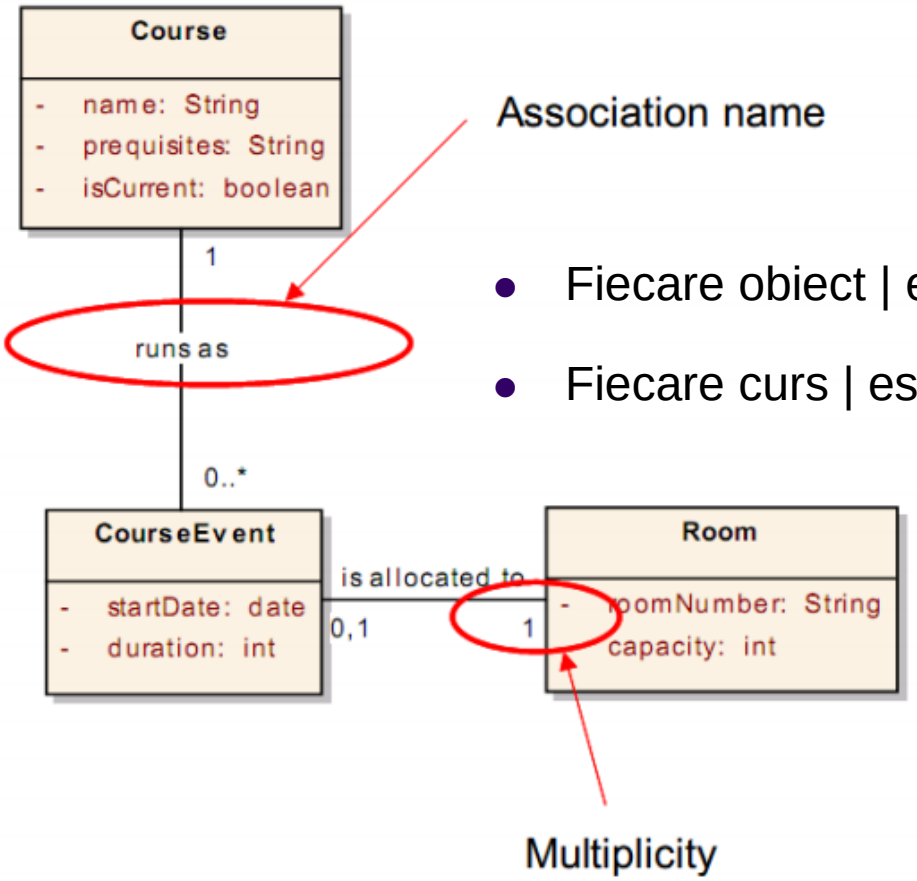


Direcția de citire

- Direcția de citire este de obicei de la stânga la dreapta și de sus în jos
- Direcția de citire se poate indica explicit
  - Săgeata care indică direcția de citire nu trebuie pusă pe linia de asociere!



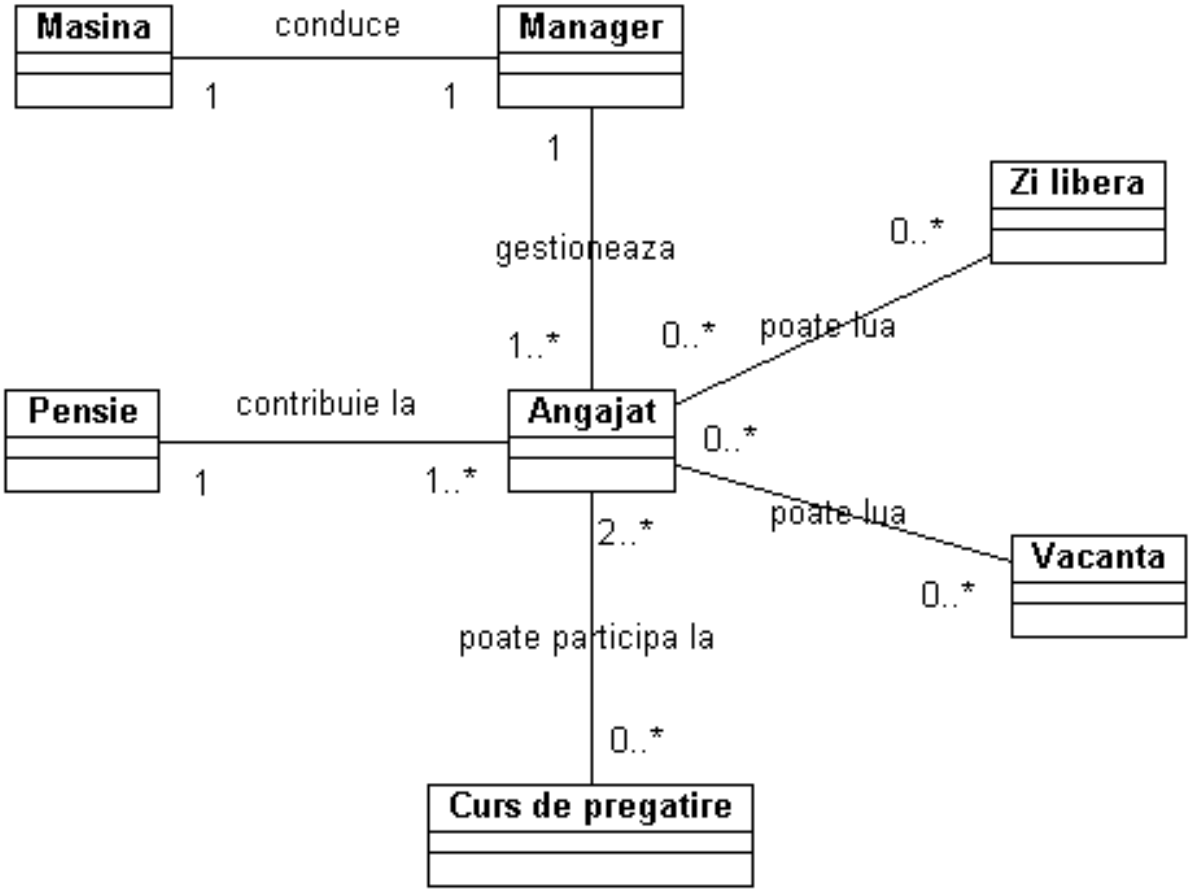
Validarea asocierilor



- Fiecare obiect | este predat ca | 0 sau mai multe | cursuri
- Fiecare curs | este pentru | un singur | obiect

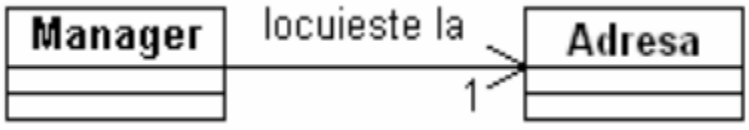
Course = obiect
Course event = curs

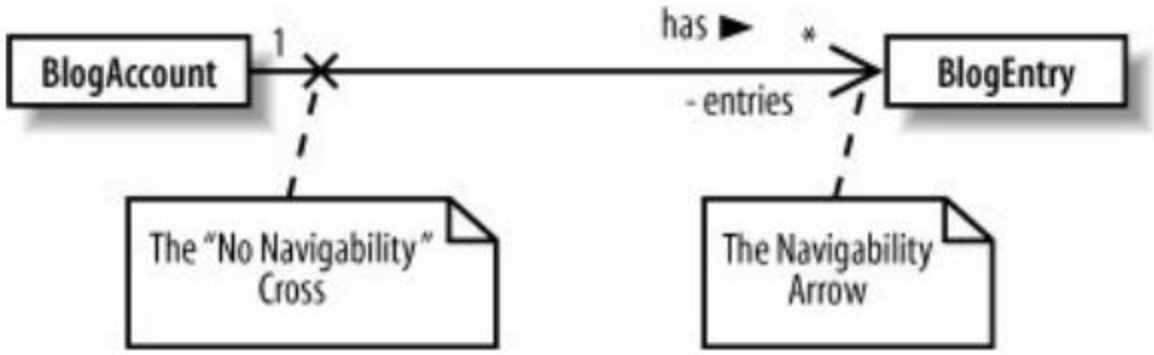
Asociere complexă



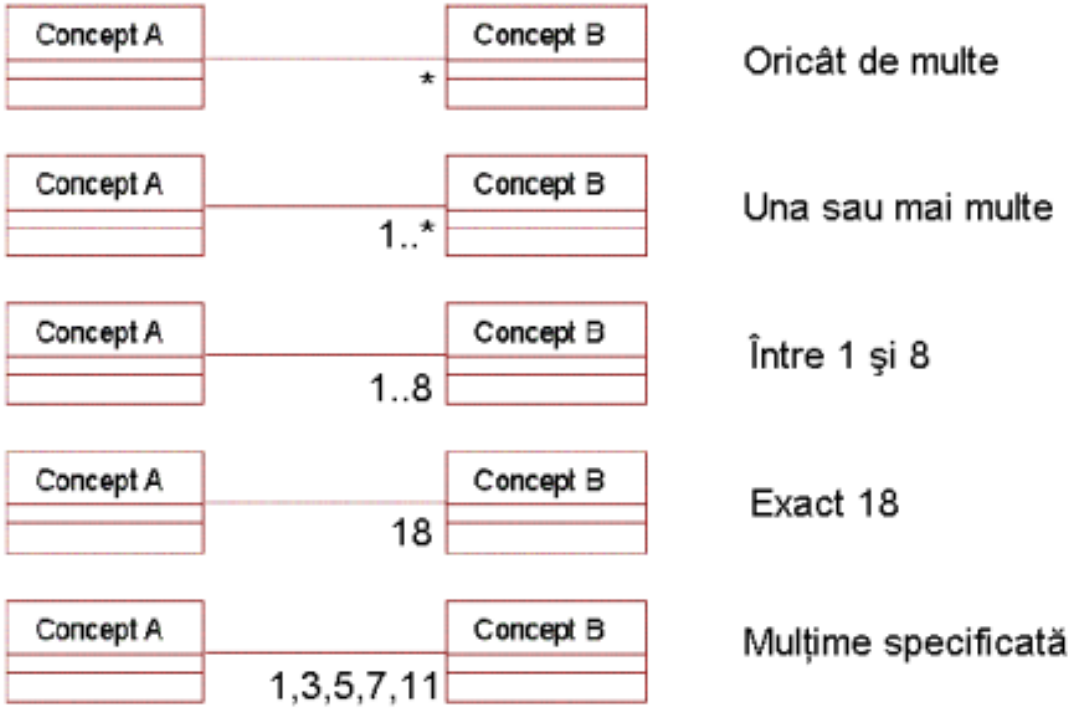
Asociere unidirecțională

- Numai o clasă „știe” de cealaltă
- *Manager* va avea un câmp de tip *Adresă* și nu invers



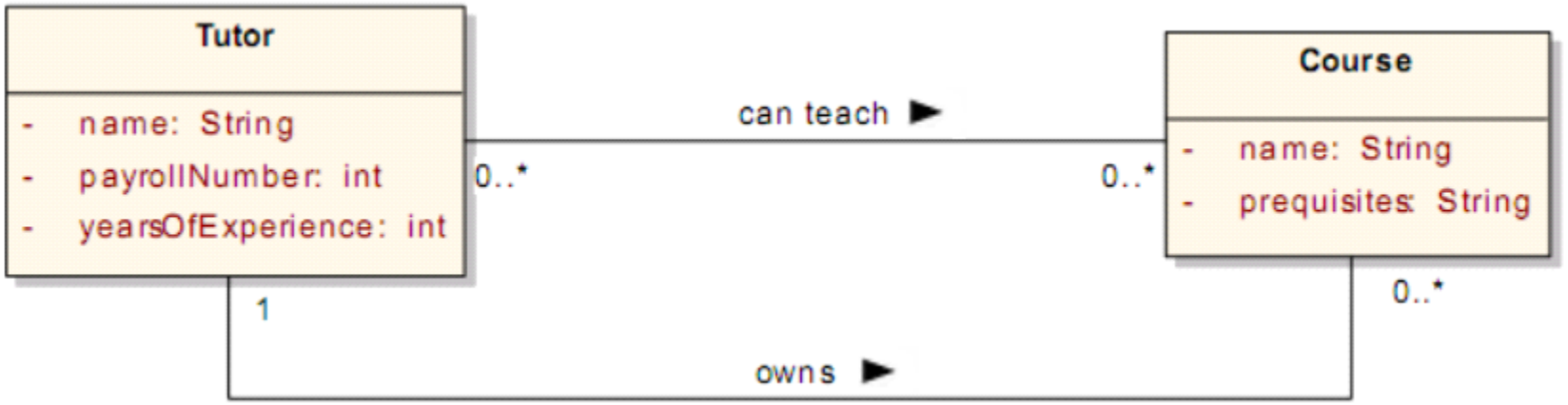


Cardinalitatea (multiplicitatea) asocierii

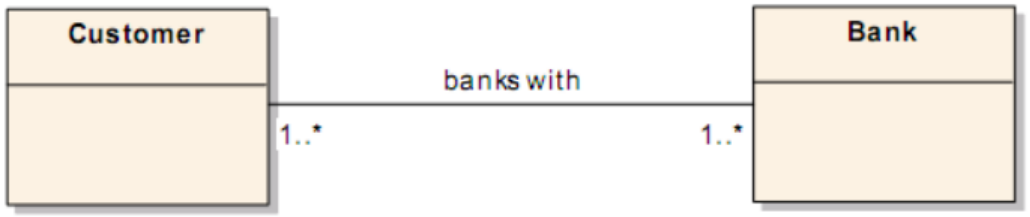


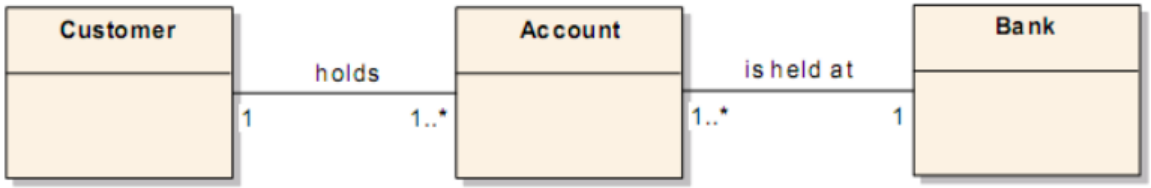
Cardinalitatea * conține mai puține informații

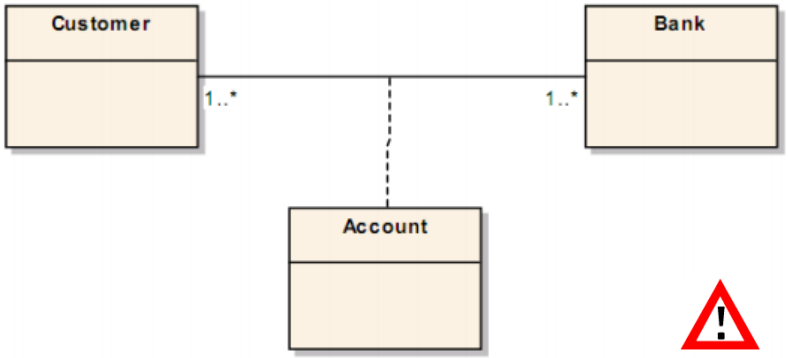
Asocieri multiple



Asocieri multiple

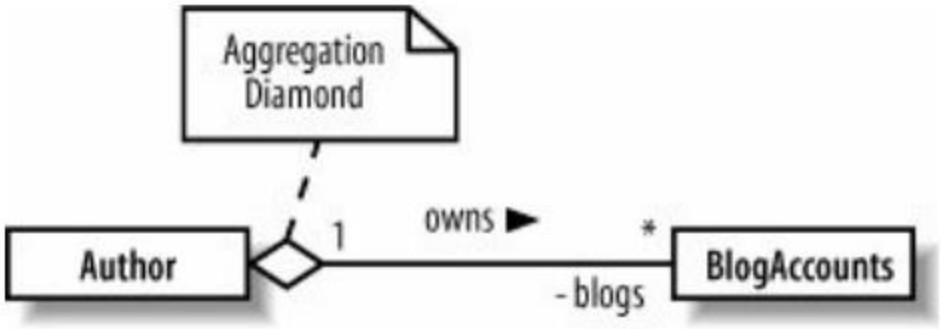


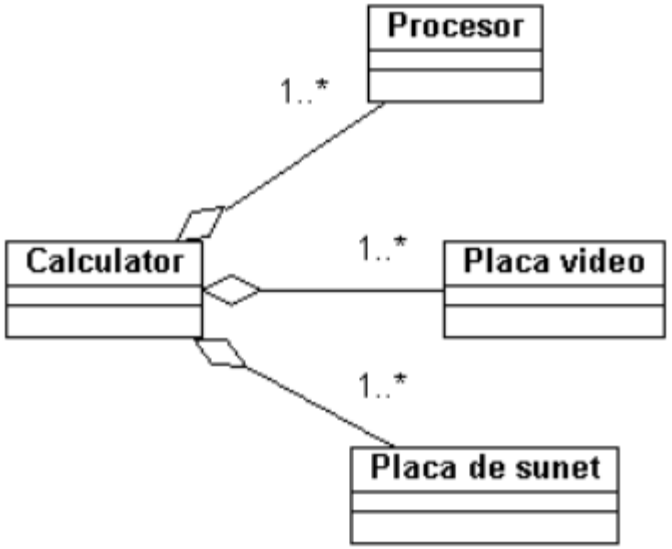




Agregarea

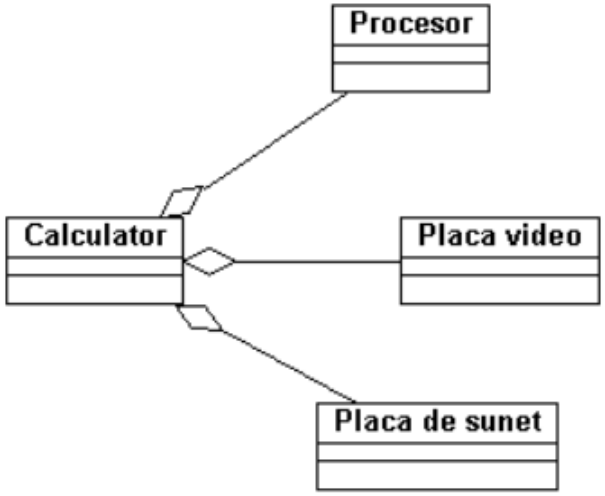
- O clasă are dar *partajează* obiecte din cealaltă clasă





Este o relație de tip **ARE-UN / ARE-O**

Agregarea

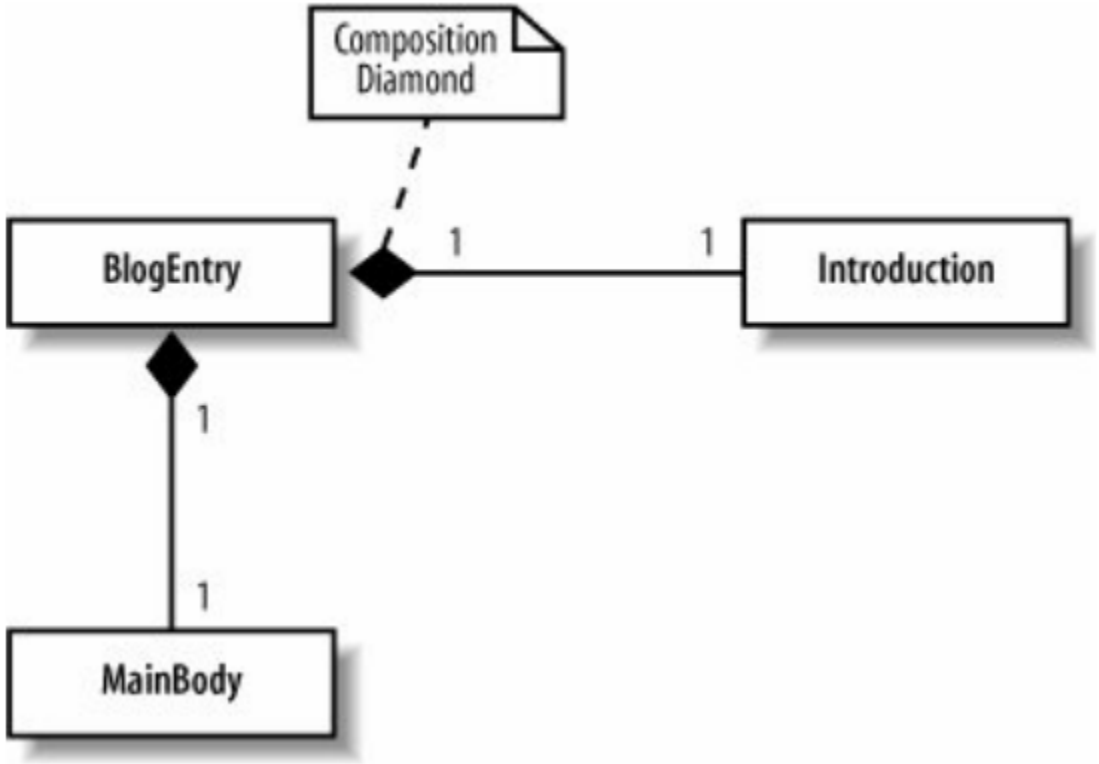


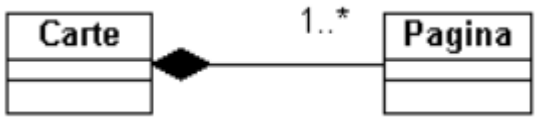

```
class Program
{
    static void Main(string[] args)
    {
        Procesor procesor = new Procesor();
        PlacaVideo placaVideo = new PlacaVideo();
        PlacaDeSunet placaDeSunet = new PlacaDeSunet();
        Calculator calculator1 = new Calculator(procesor, placaVideo, placaDeSunet);
        Calculator calculator2 = new Calculator(procesor, placaVideo, placaDeSunet);
    }
}

public class Calculator
{
    private Procesor _procesor;
    private PlacaVideo _placaVideo;
    private PlacaDeSunet _placaDeSunet;

    public Calculator(Procesor procesor, PlacaVideo placaVideo, PlacaDeSunet placaDeSunet)
    {
        _procesor = procesor;
        _placaVideo = placaVideo;
        _placaDeSunet = placaDeSunet;
    }
}
```

```
public class Procesor
{

}

public class PlacaVideo
{

}

public class PlacaDeSunet
{

}
```

Compunerea

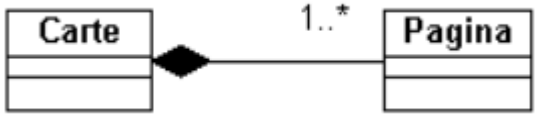
- Atributele *compun* clasa





Compunerea

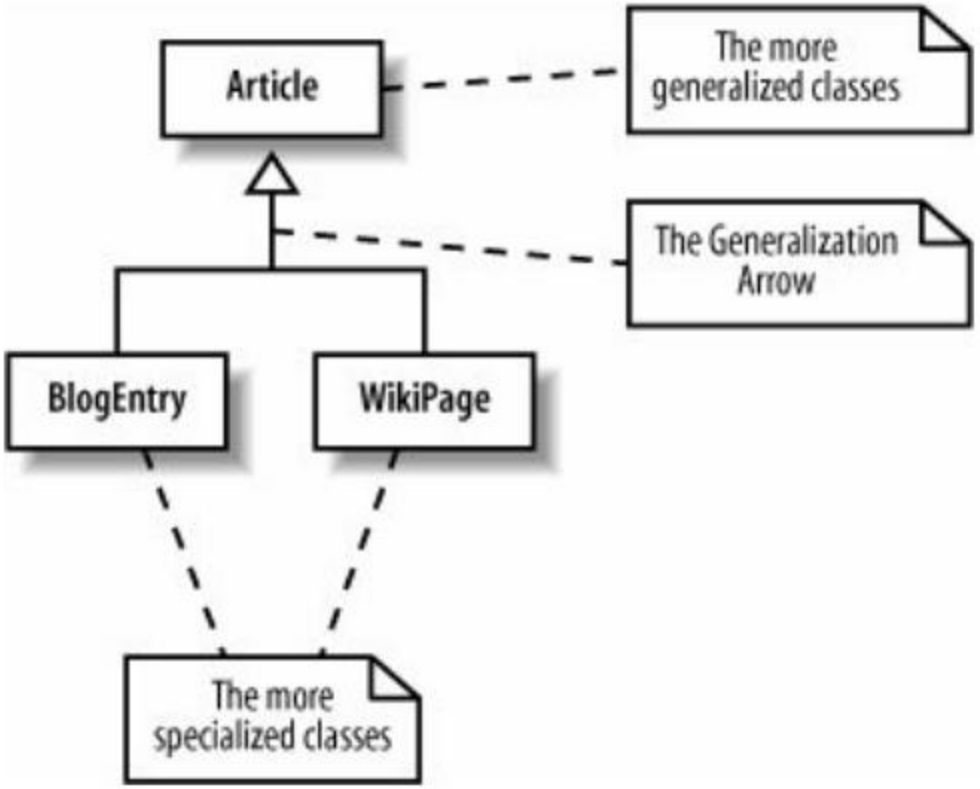
```
public class Carte
{
    private List<Pagina> _pagini;

    public Carte()
    {
        _pagini = new List<Pagina>();
    }
}

public class Pagina
{

}
```



Implementarea

- Din punct de vedere al implementării, asocierea, agregarea și compunerea presupun introducerea unui atribut (câmp)
- Dacă nu sunt evidente sau foarte importante distincțiile privind agregarea sau compunerea, este mai simplu să se folosească numai relațiile de asociere

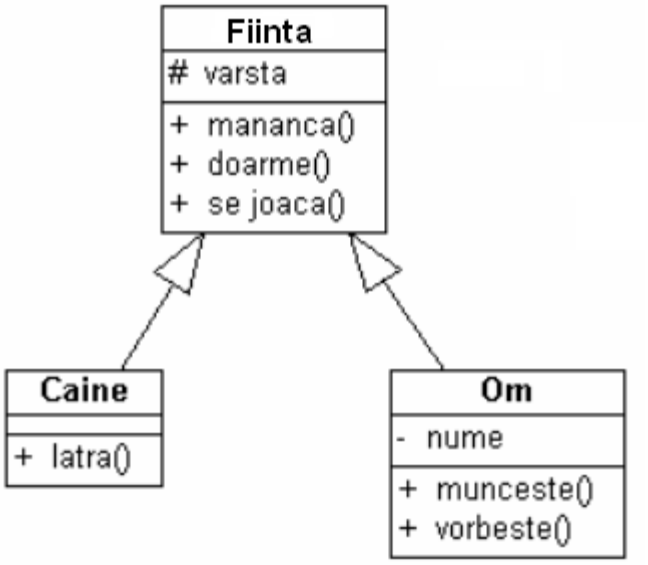
Generalizarea (moștenirea)

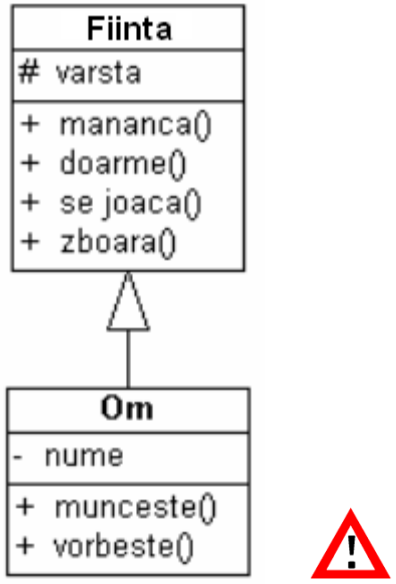


Este o relație de tip **ESTE-UN / ESTE-O**

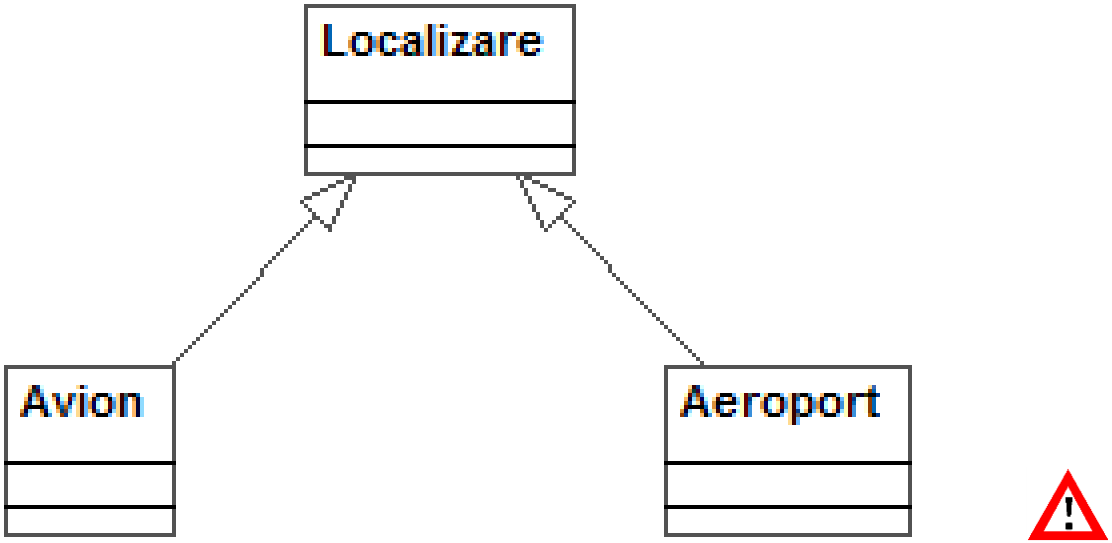
Regula 100%

- Toate definițiile clasei de bază trebuie să se aplice tuturor claselor derivate

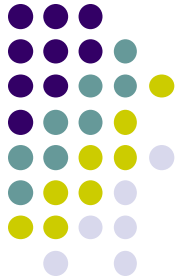




Corectitudinea generalizărilor

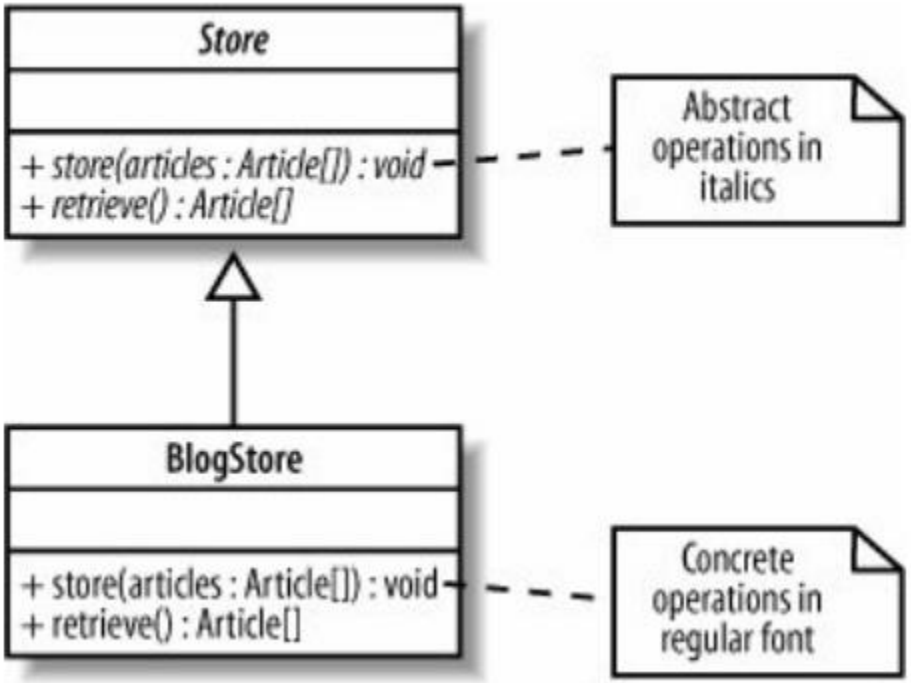


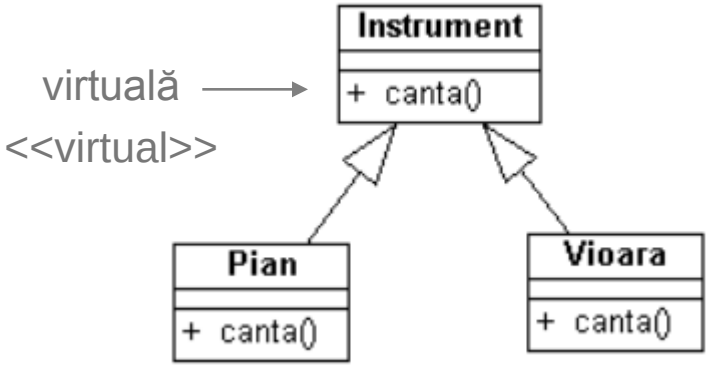
Recomandare

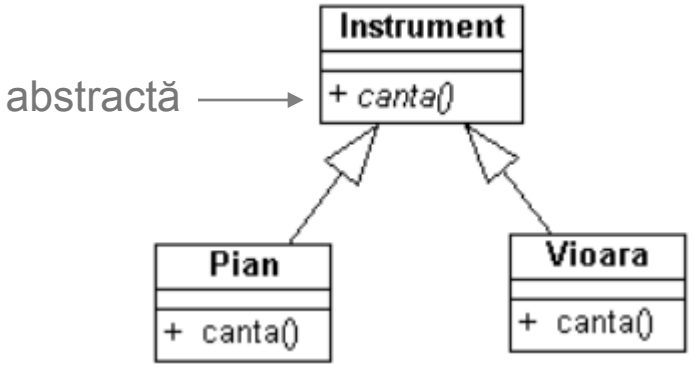


- Compunerea ar trebui preferată moștenirii
  - Moștenirea este cea mai puternică formă de cuplare
  - În general, compunerea este mai ușor de gestionat

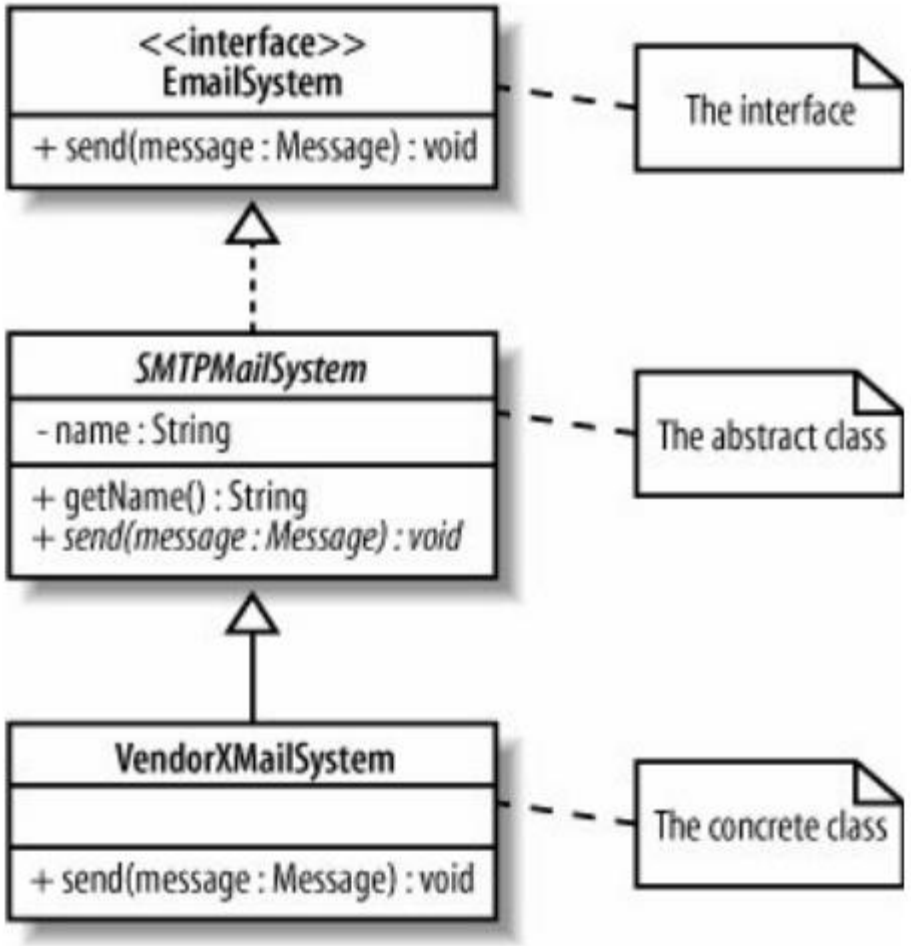
Clase și operații abstracte

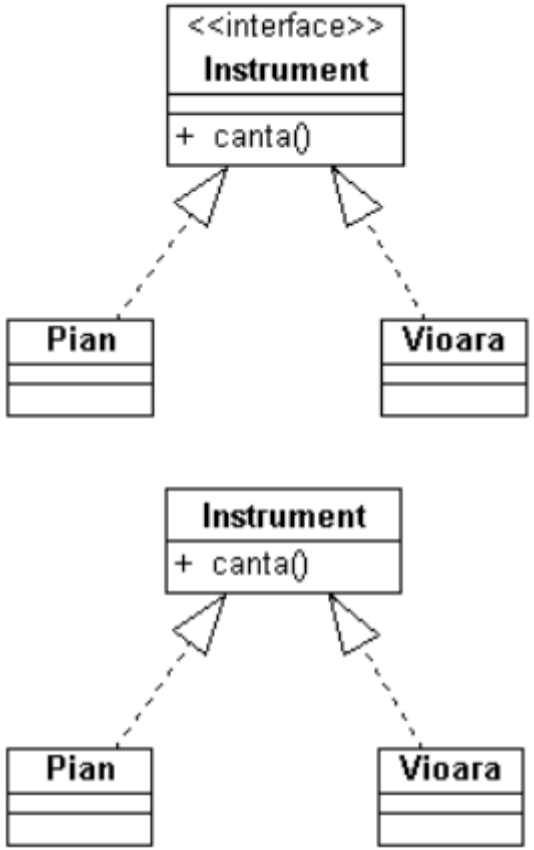




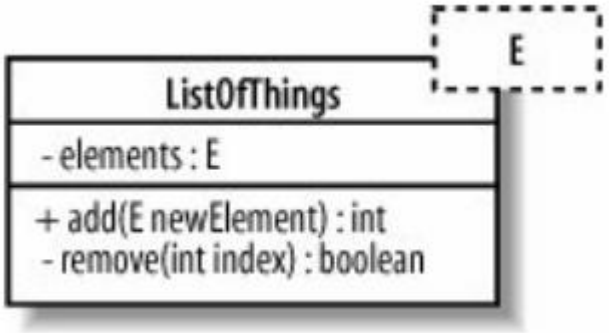


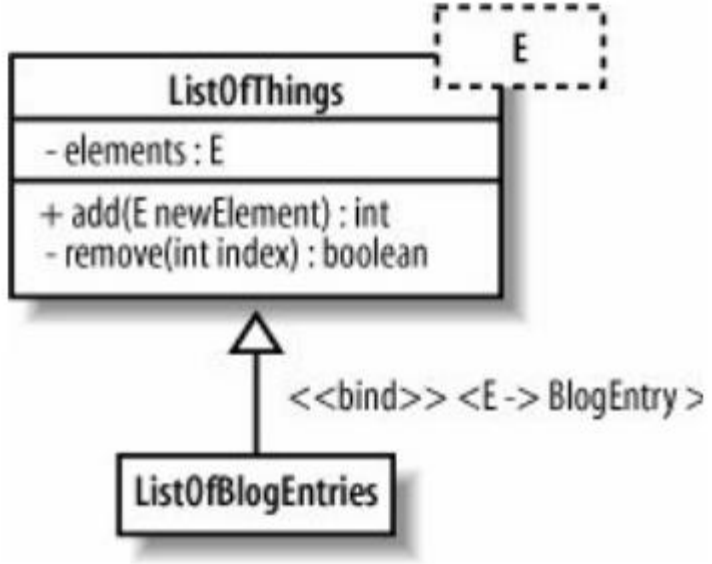
Interfețe


<<interface>>
EmailSystem
+ send(message : Message) : void
The interface
SMTPMailSystem
- name : String
+ getName() : String
+ send(message : Message) : void
The abstract class
VendorXMailSystem
+ send(message : Message) : void
The concrete class


<<interface>>
Instrument
+ canta()
Pian
Vioara
Instrument
+ canta()
Pian
Vioara

Template-uri

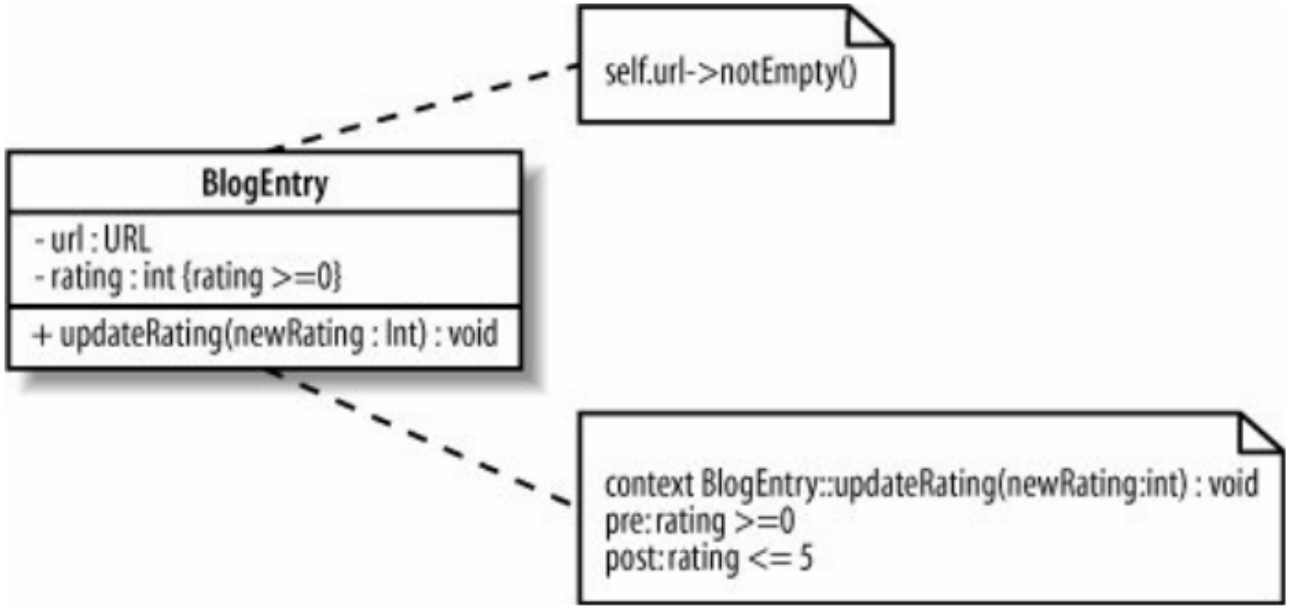




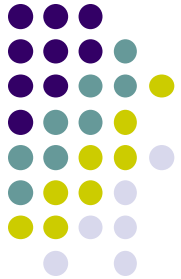
De exemplu: List<int> – tipul este specificat în implementare

Constrângeri (în OCL)

- Tipuri de constrângeri:
  - Invariante
  - Pre-condiții
  - Post-condiții

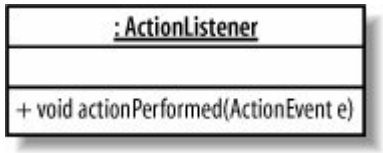


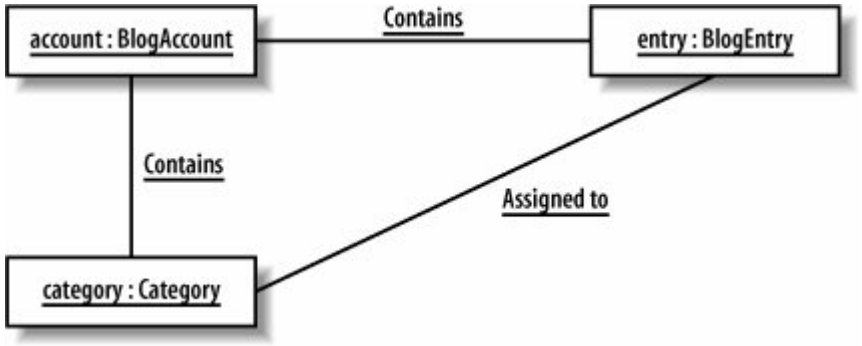
Recomandări

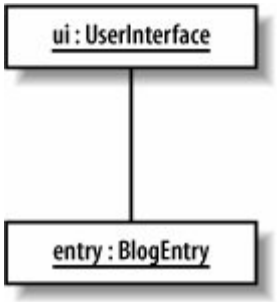


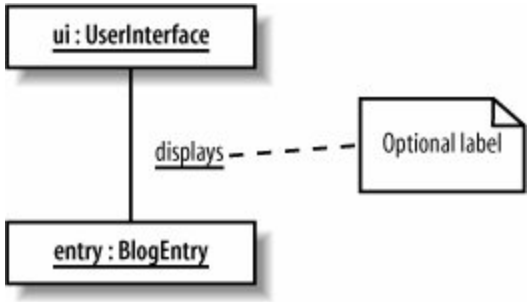
- Nu introduceți prea multe informații în diagramă
- Ignorați atributele și operațiile necritice
- Arătați într-o diagramă numai clasele relevante pentru un caz de utilizare
- Nu includeți clasele sistem (*string*, *Dictionary* etc.)
- Nu includeți prea devreme informații despre implementare (navigabilitate, vizibilitate)

3. Diagrama de obiecte





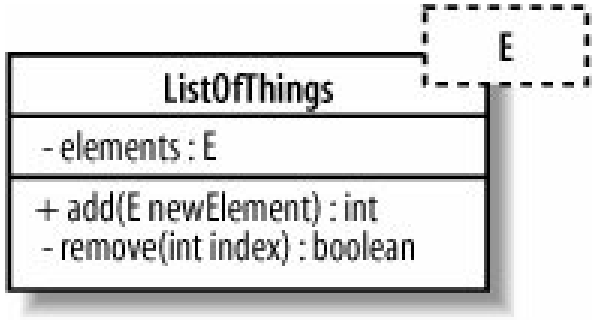




Arată cum se comportă obiectele într-un anumit scenariu, cum interacționează

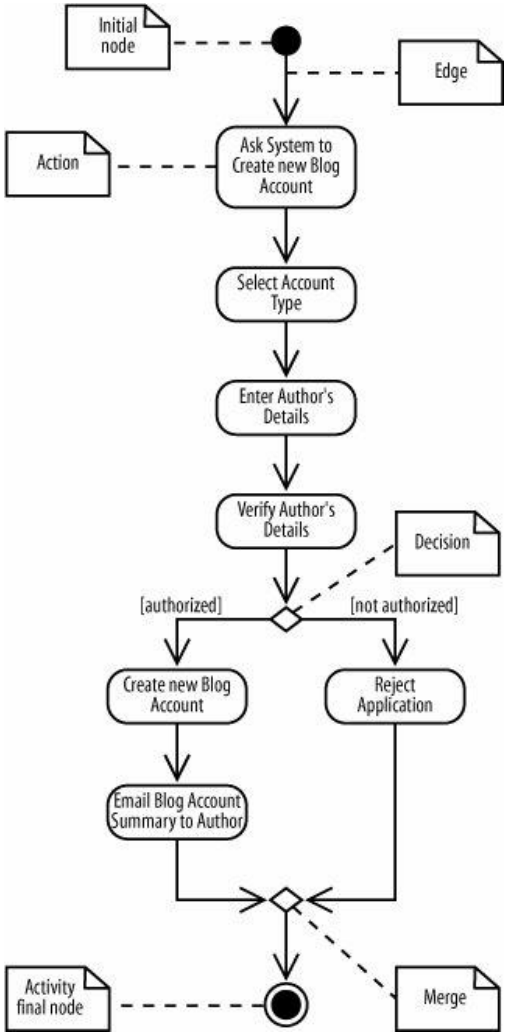
Asocierile corespund asocierilor dintre clase

Template-uri



De exemplu: List<ClasaDeBaza> – tipul derivat din ClasaDeBaza este specificat în momentul execuției

4. Diagrama de activități



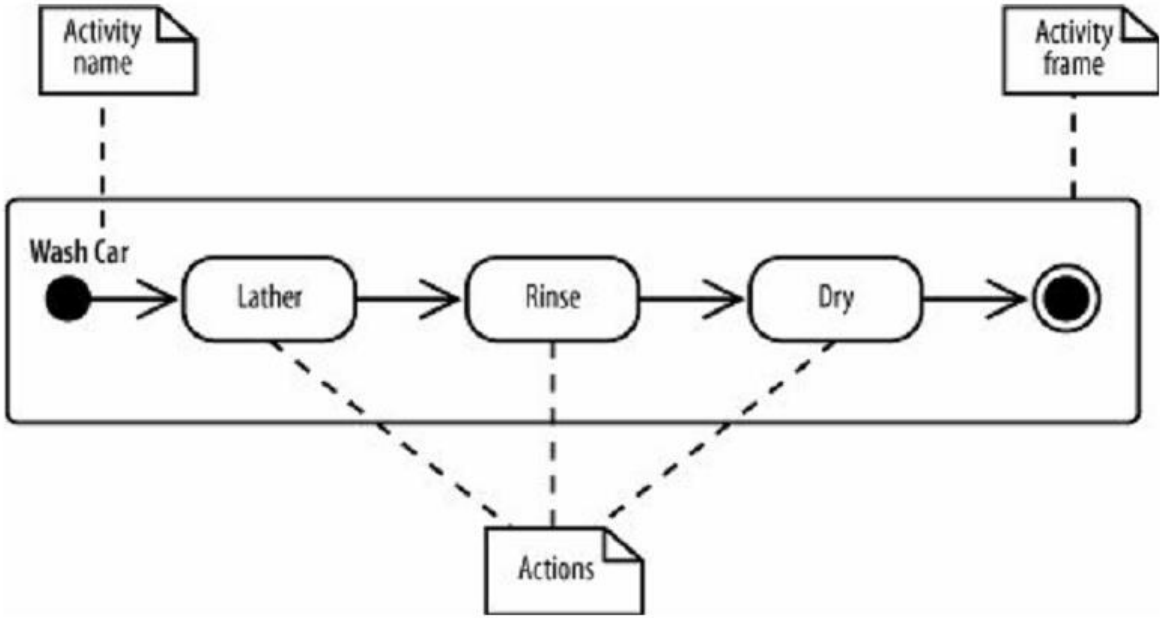
Aceste diagrame descriu logica procedurală, procesele

Sunt asemănătoare schemelor logice, dar suportă și paralelismul

"merge" = îmbinare

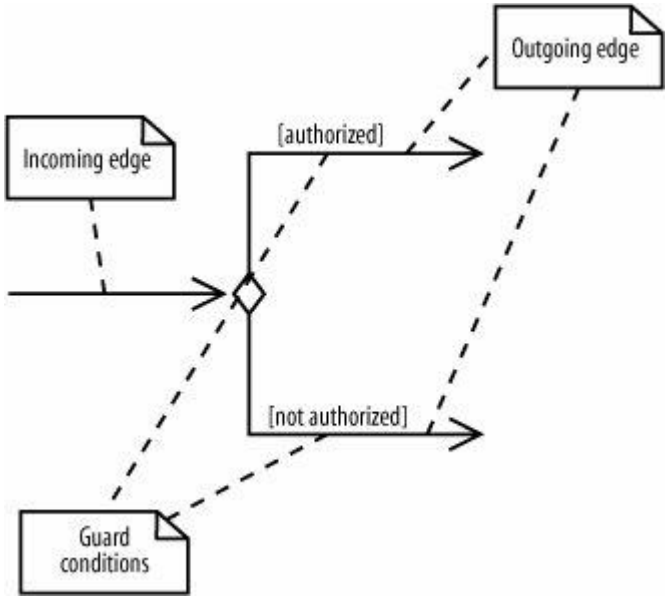
Activități și acțiuni

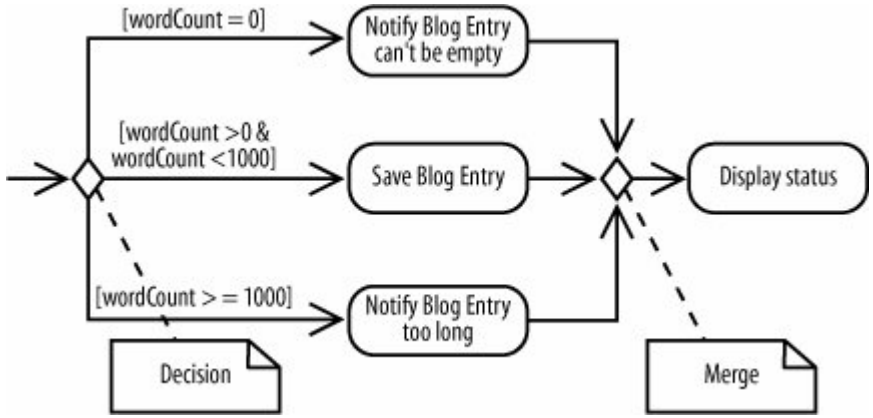
- Activitatea este procesul modelat
- O acțiune este un pas din activitate



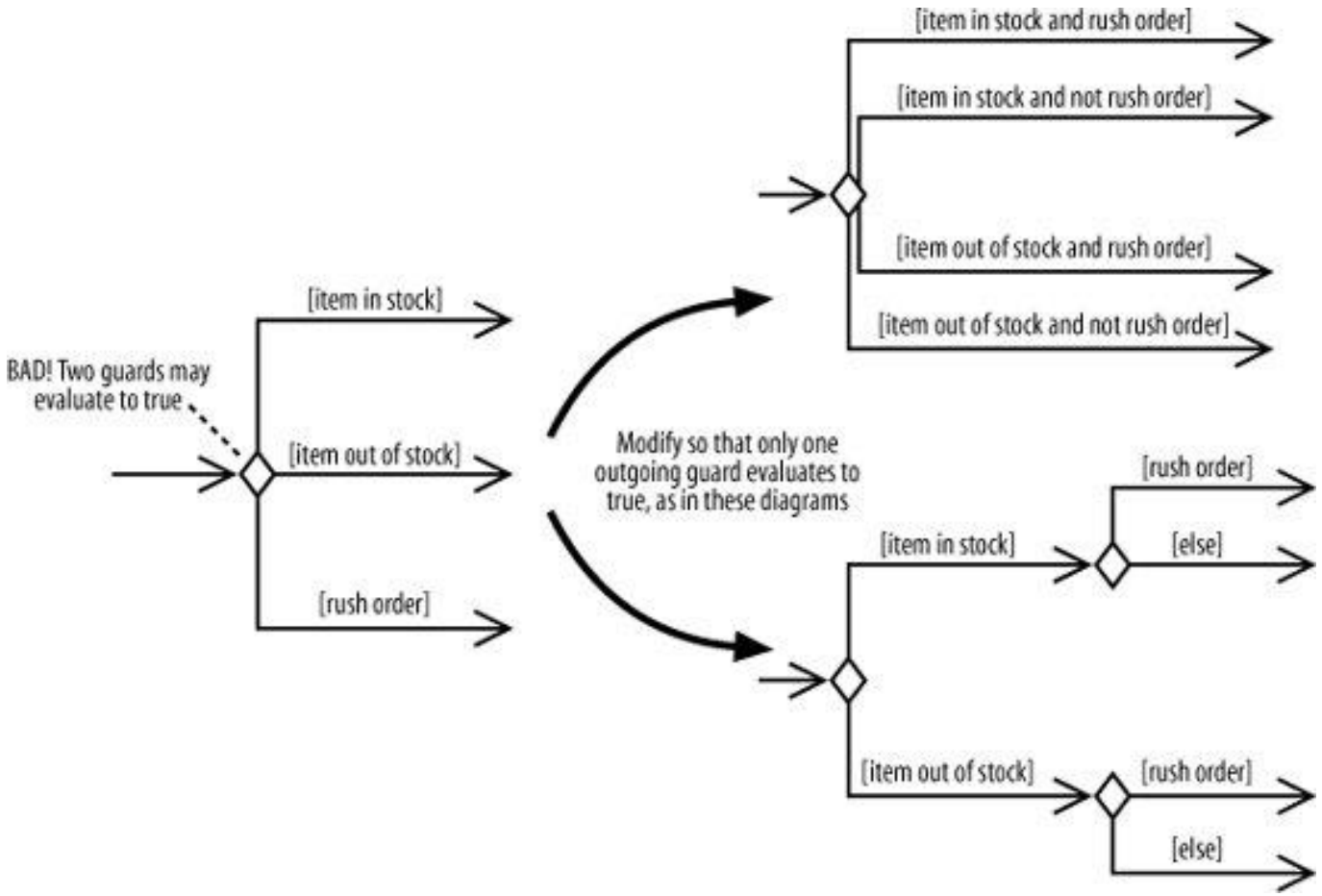
Decizii și îmbinări

- Condițiile trebuie să fie complete și mutual exclusive
- Pasul următor trebuie să fie unic determinat

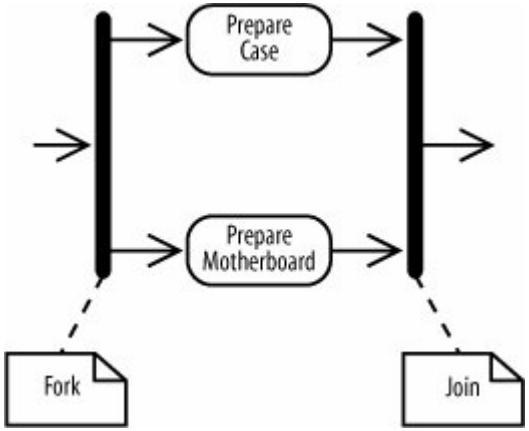




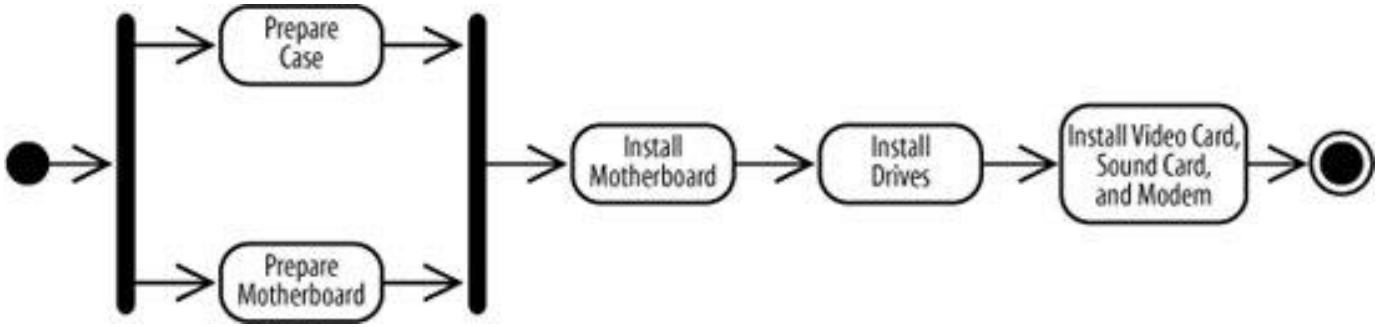
Teste incomplete



Procese paralele

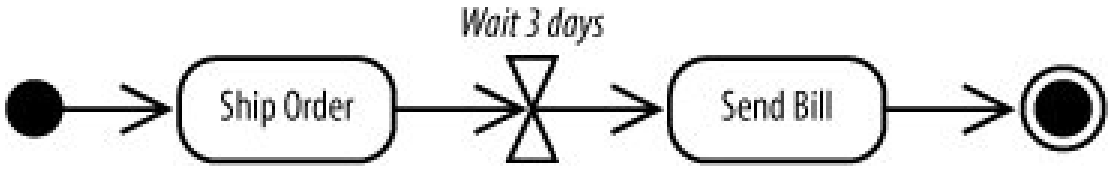


"fork" = ramificație
"join" = reunire

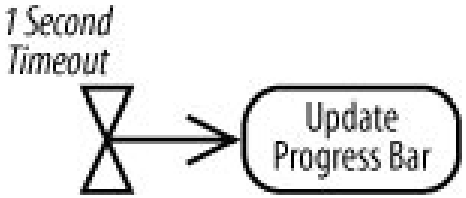


Se folosesc pentru procese sau fire de execuție multiple
La reunire, se așteaptă terminarea tuturor acțiunilor incidente

Evenimente de timp

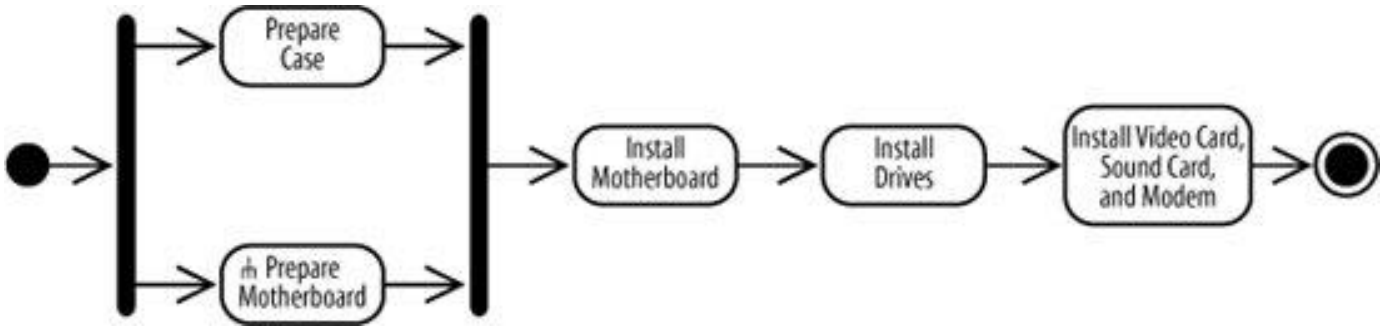


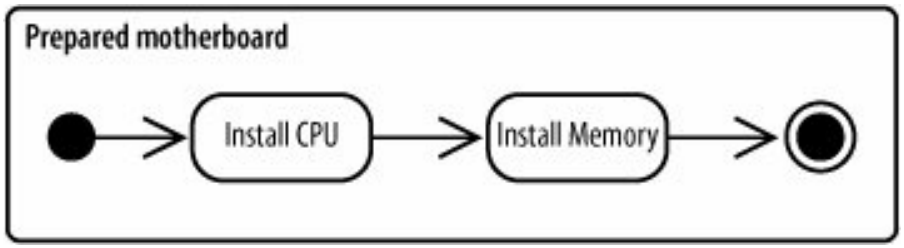
Perioadă de așteptare



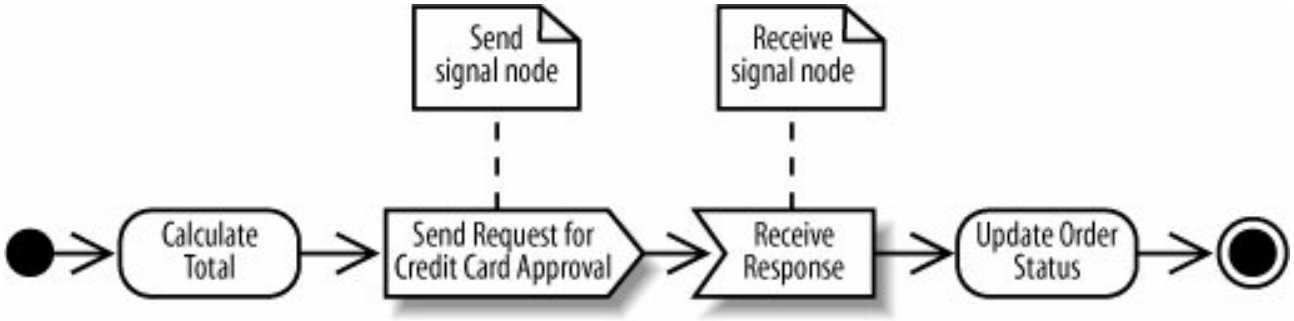
Eveniment de timp recurent

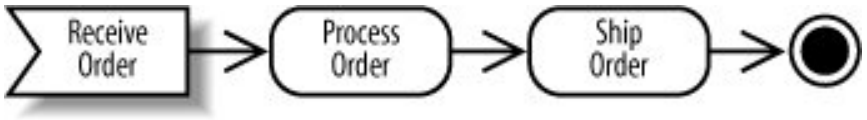
Apelarea altor activități





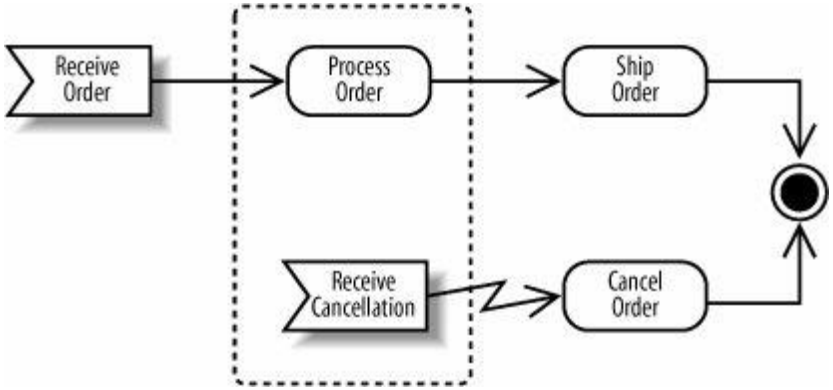
Semnale

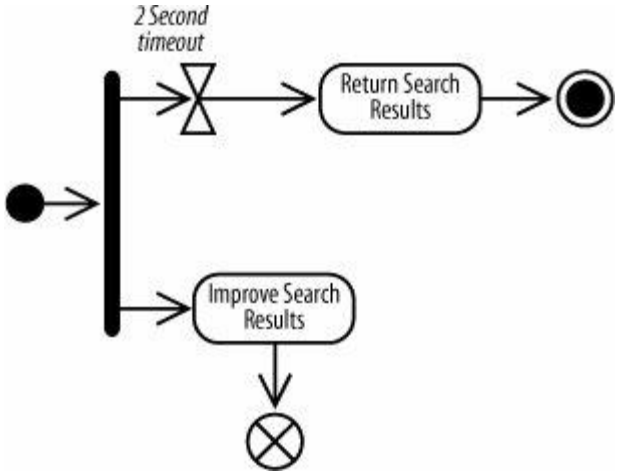




Așteaptă tot timpul

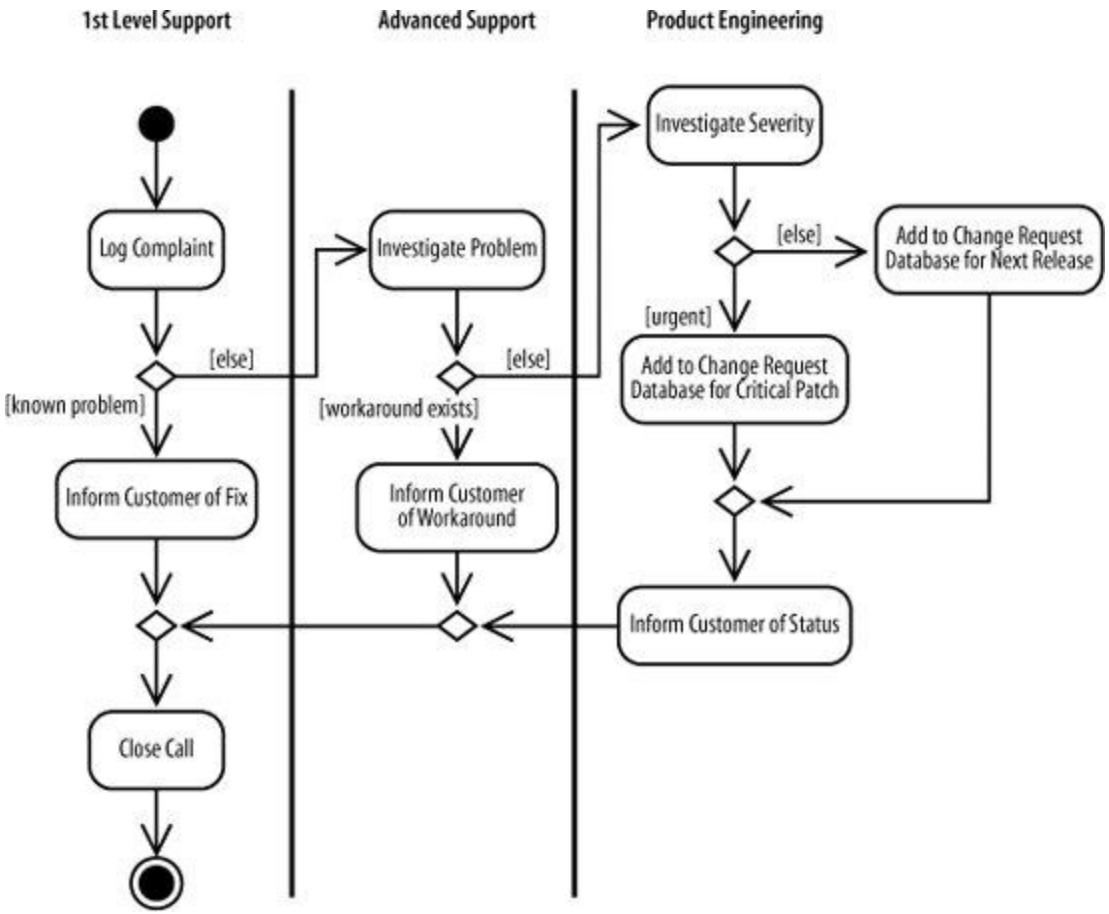
Întreruperi și terminări de flux



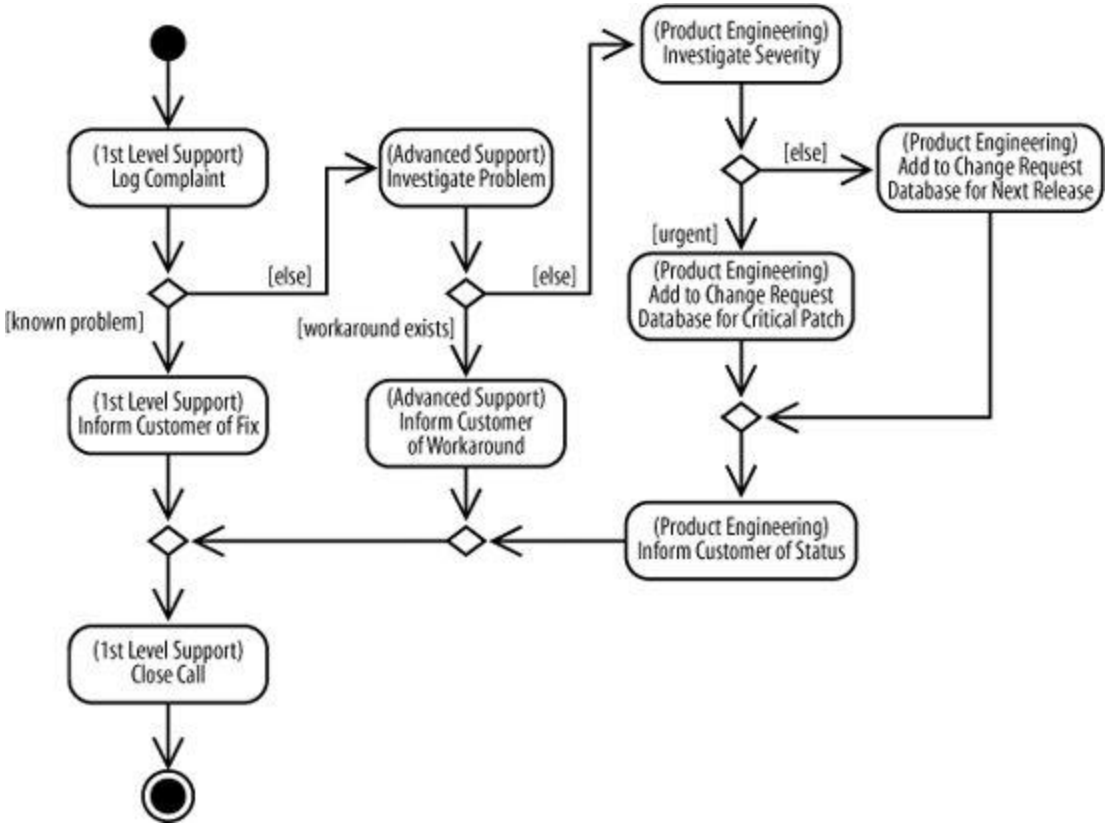


Un flux se poate termina fără a termina întreaga activitate
Dacă se mai poate îmbunătăți soluția în 2 secunde, este foarte bine, dacă nu, se returnează rezultatul existent
După 2 secunde, se returnează orice rezultat disponibil, îmbunătățit sau nu

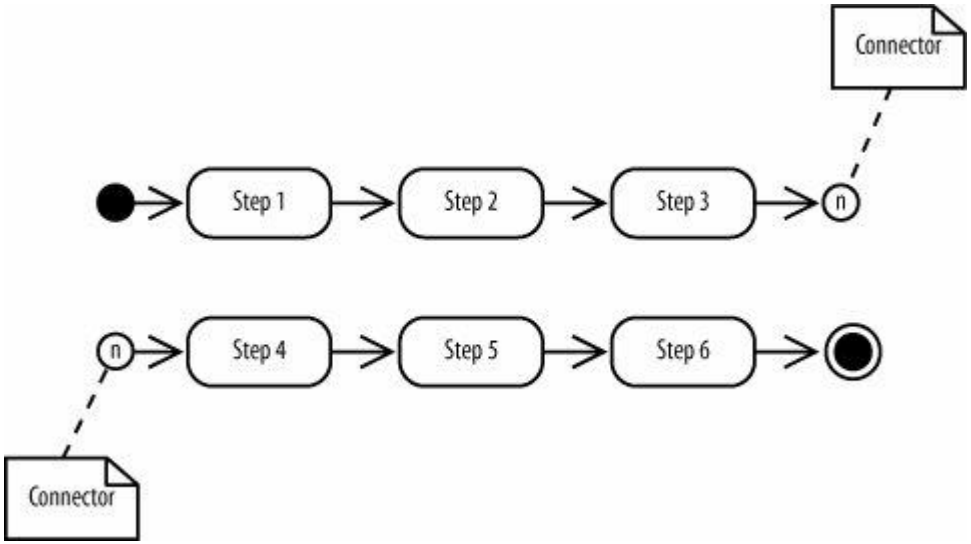
Partiții (culoare)



Adnotări

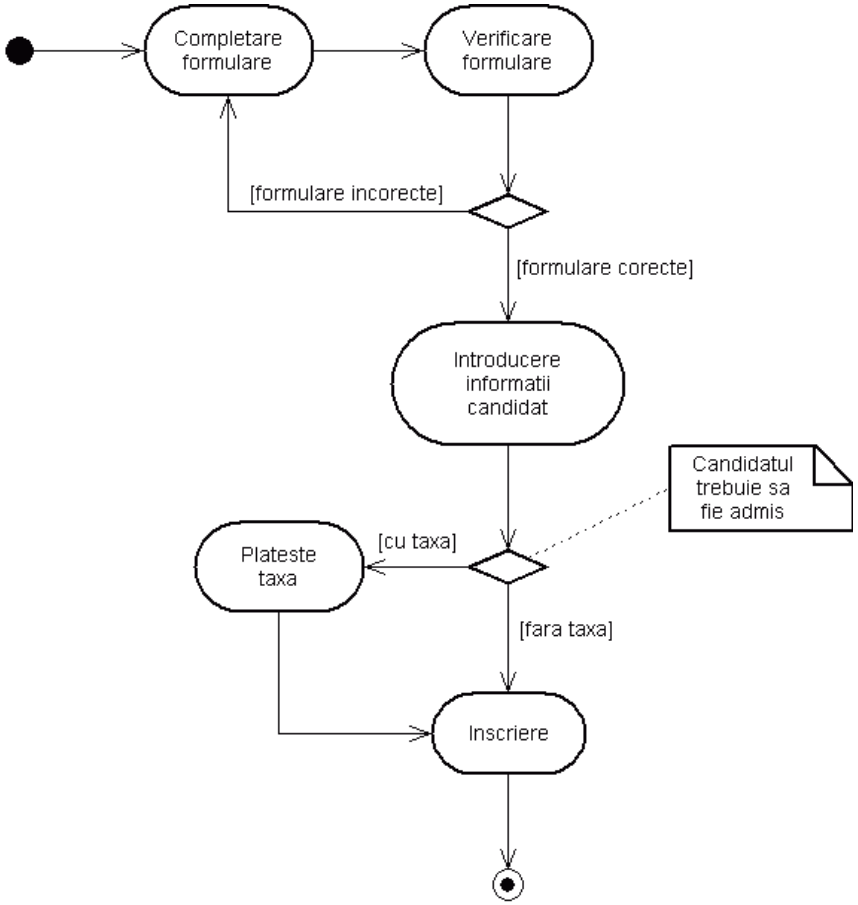

(1st Level Support)
Log Complaint
[known problem]
[else]
(Advanced Support)
Investigate Problem
[workaround exists]
[else]
(Product Engineering)
Investigate Severity
[urgent]
[else]
(Product Engineering)
Add to Change Request
Database for Next Release
(Product Engineering)
Add to Change Request
Database for Critical Patch
(1st Level Support)
Inform Customer of Fix
(Advanced Support)
Inform Customer
of Workaround
(Product Engineering)
Inform Customer of Status
(1st Level Support)
Close Call

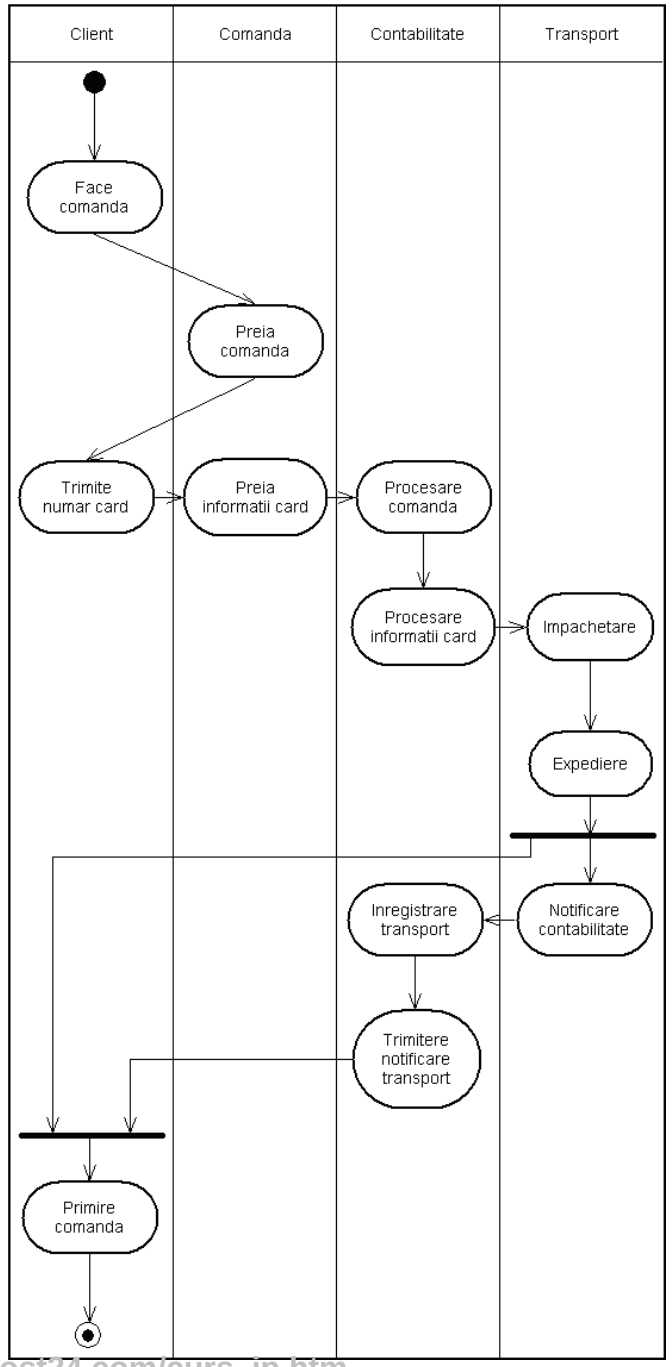
Conectori



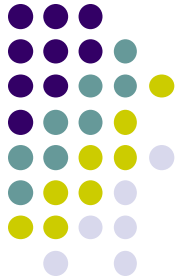
Un conector este reprezentat de obicei de o singură literă

Alte exemple



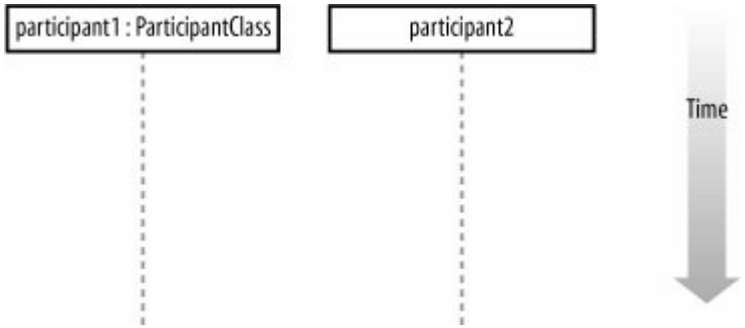


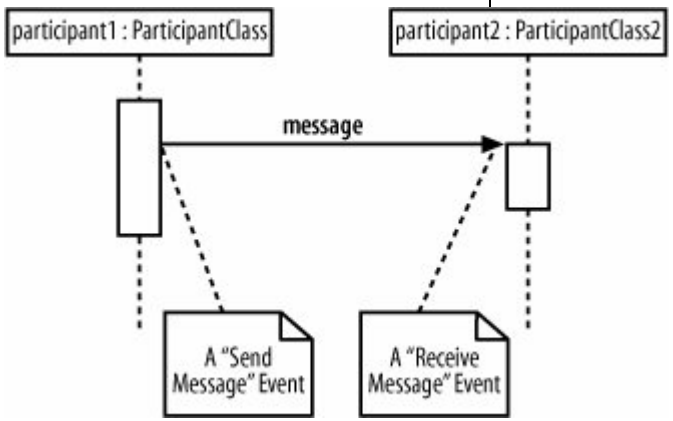
5. Diagrama de secvențe

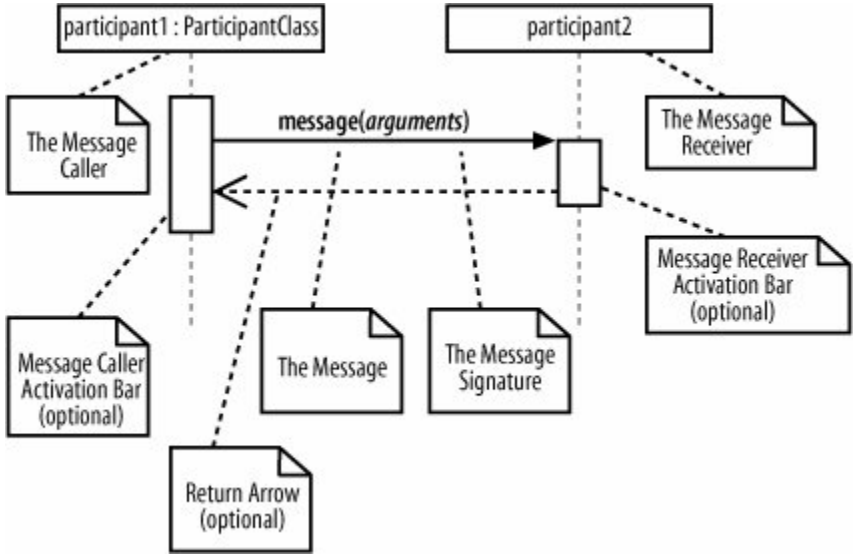


- Arată modul cum lucrează sistemul, ceea ce poate fi mai greu de înțeles doar din descrierea statică a structurii
- O diagramă corespunde unui singur scenariu
- Include în principal obiecte și mesaje
  - Un mesaj este în general un apel de metodă
- Indică ordinea evenimentelor
  - Timpul reprezintă aici ordinea, nu durata
  - Pentru durată, se folosește diagrama de cronometrare

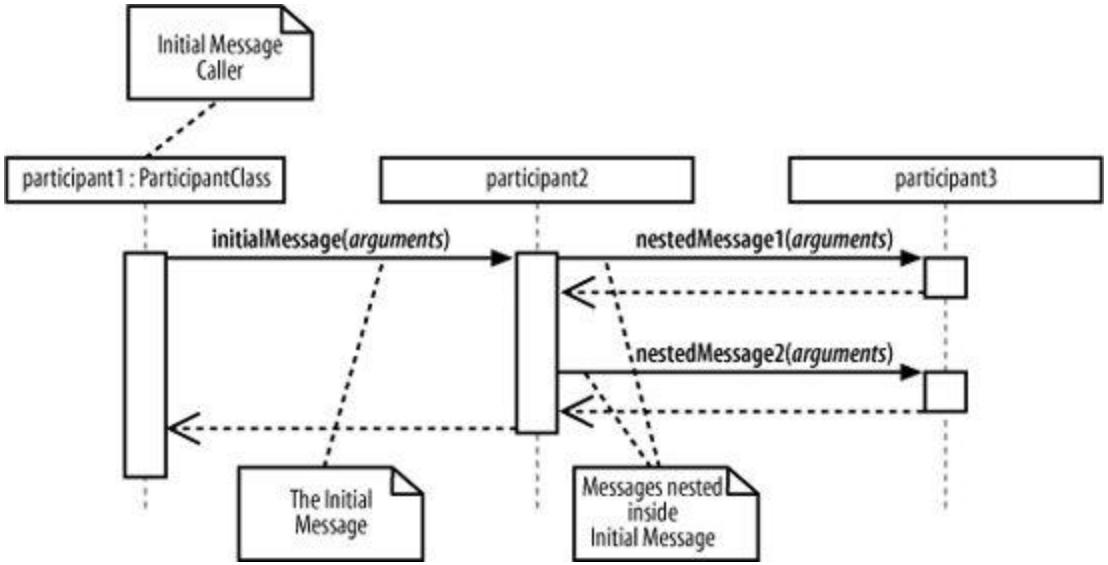
Participanți



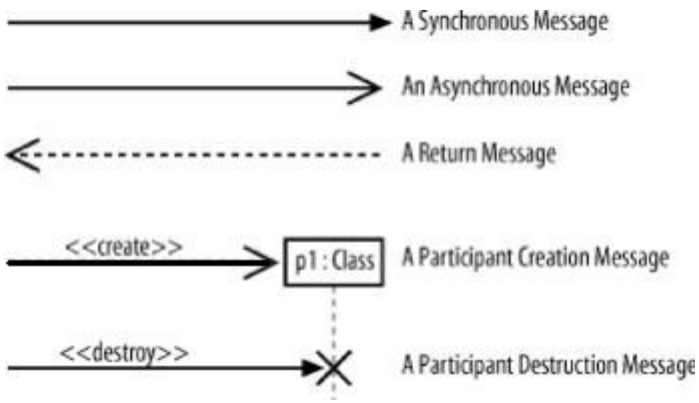


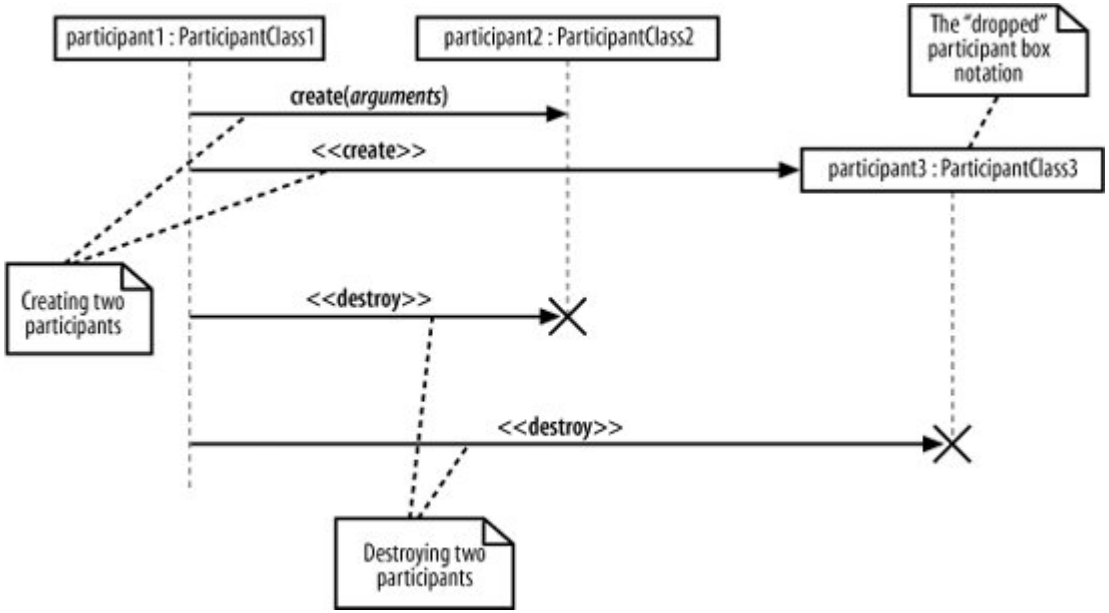


Mesaje imbricate

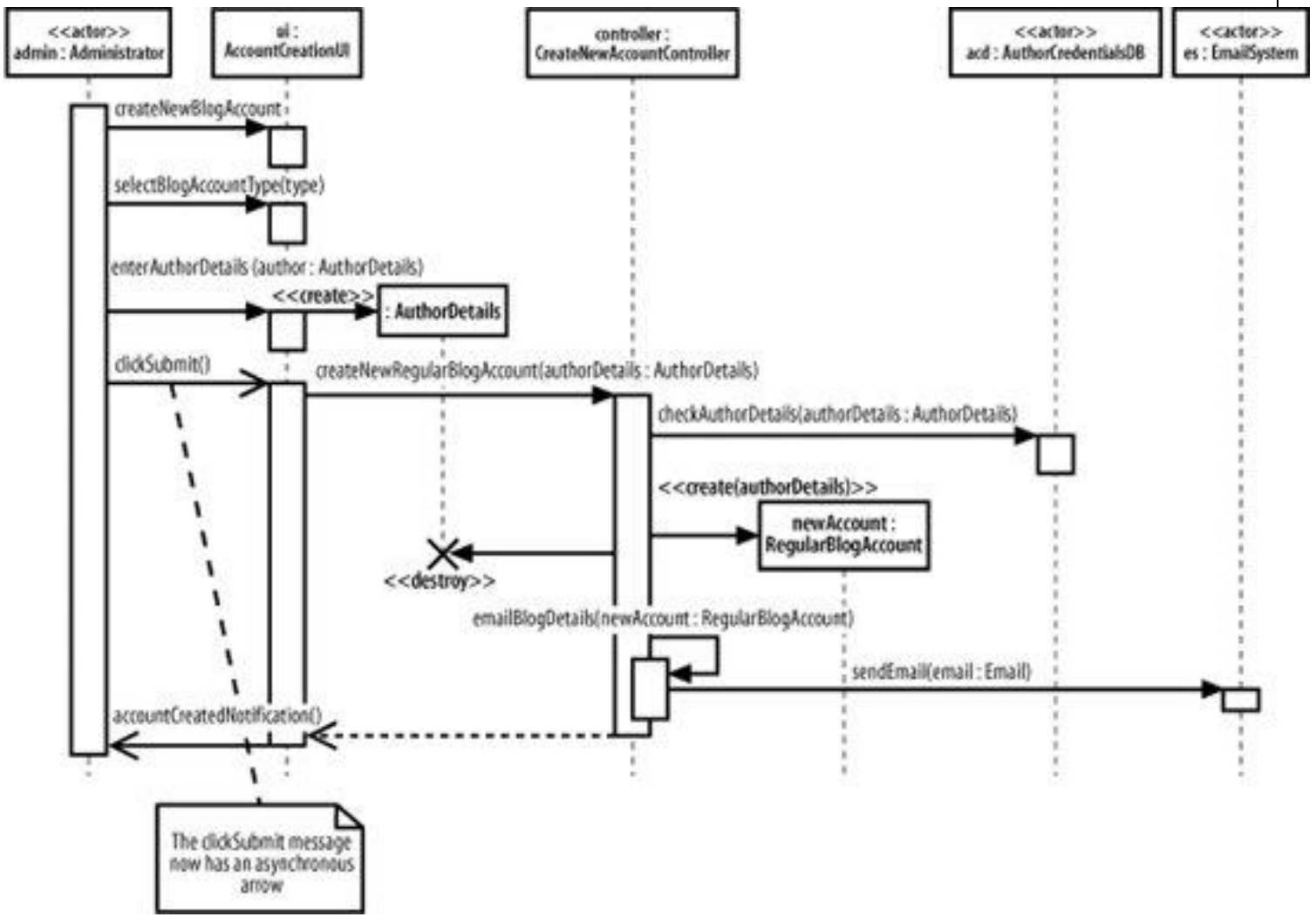


Tipuri de mesaje

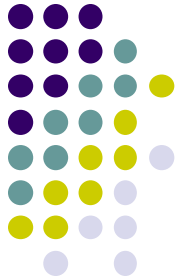




Diagramă complexă



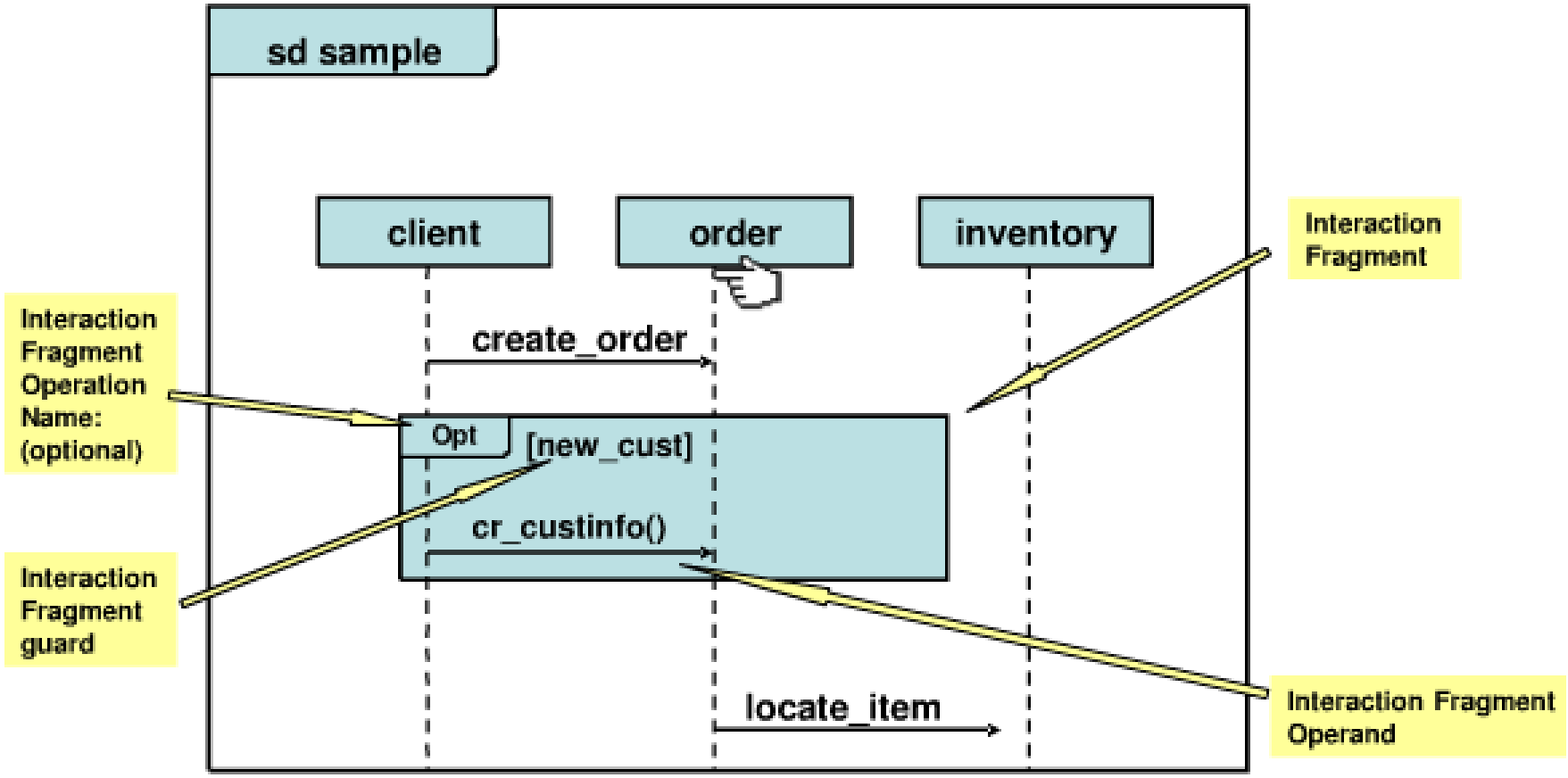
Fragmente



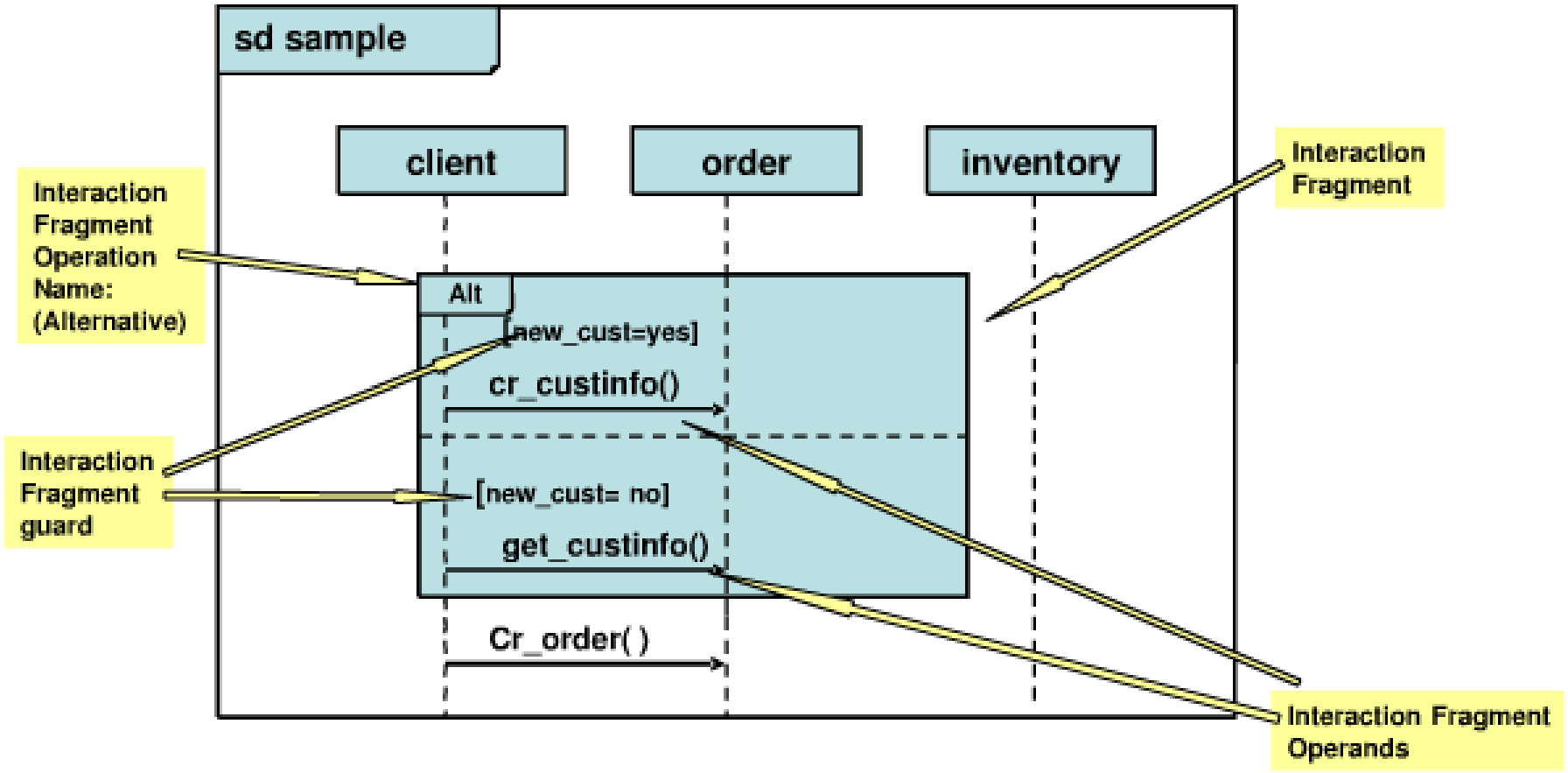
- Controlul secvențial natural poate fi extins cu ajutorul fragmentelor de interacțiune, introduse în UML 2.0
- Fragmentele sunt asemănătoare structurilor de control dintr-un limbaj de programare

Optional Fragment	**("if–then")**
Alternative fragment	**("if-then-else-if - - -" or "case")**
Break Fragment	**("break")**
Loop Fragment	**(iterations or "loop")**

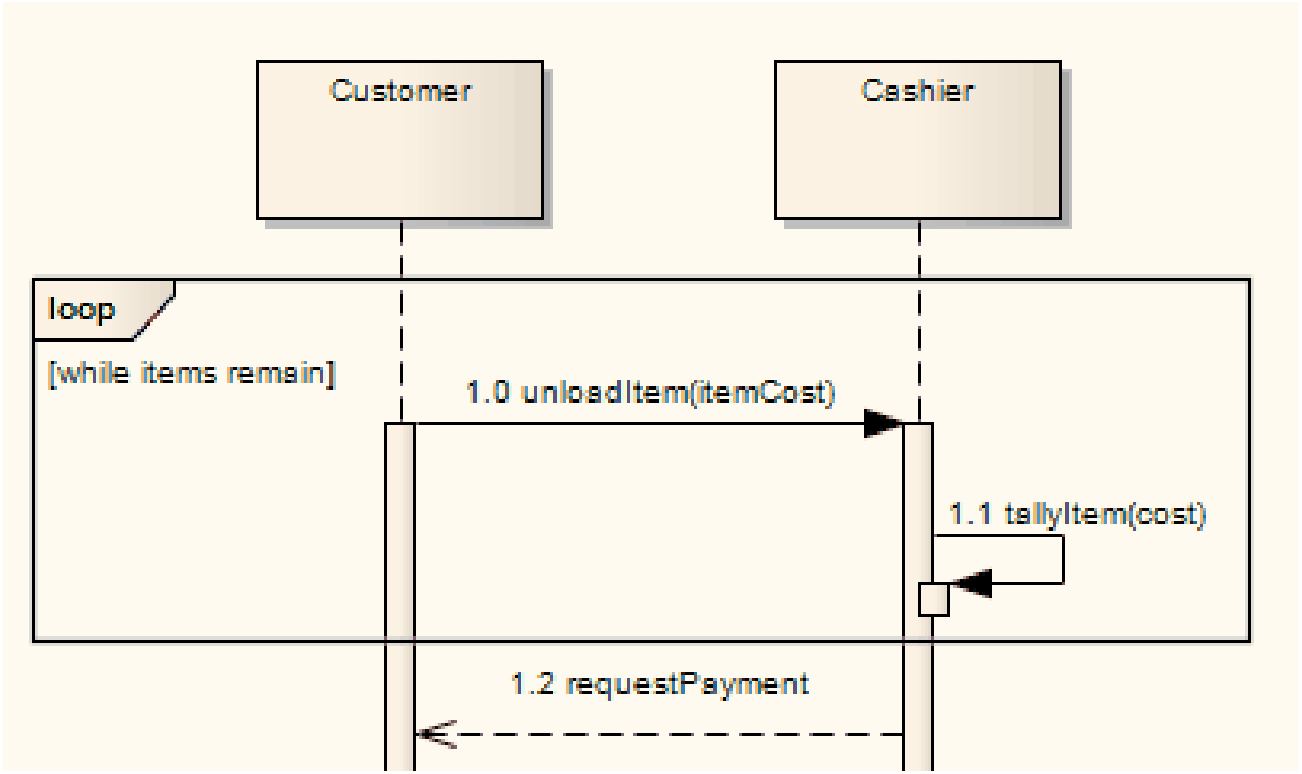
Fragmente opționale



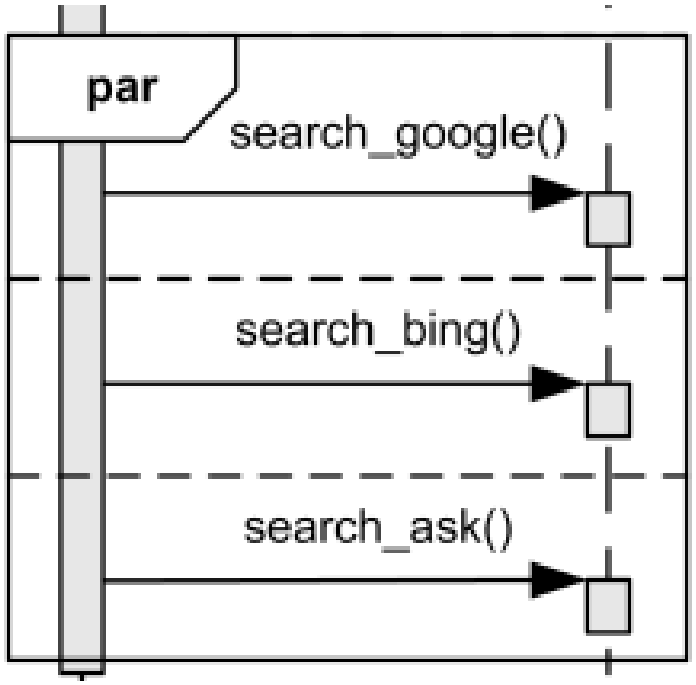
Fragmente alternative



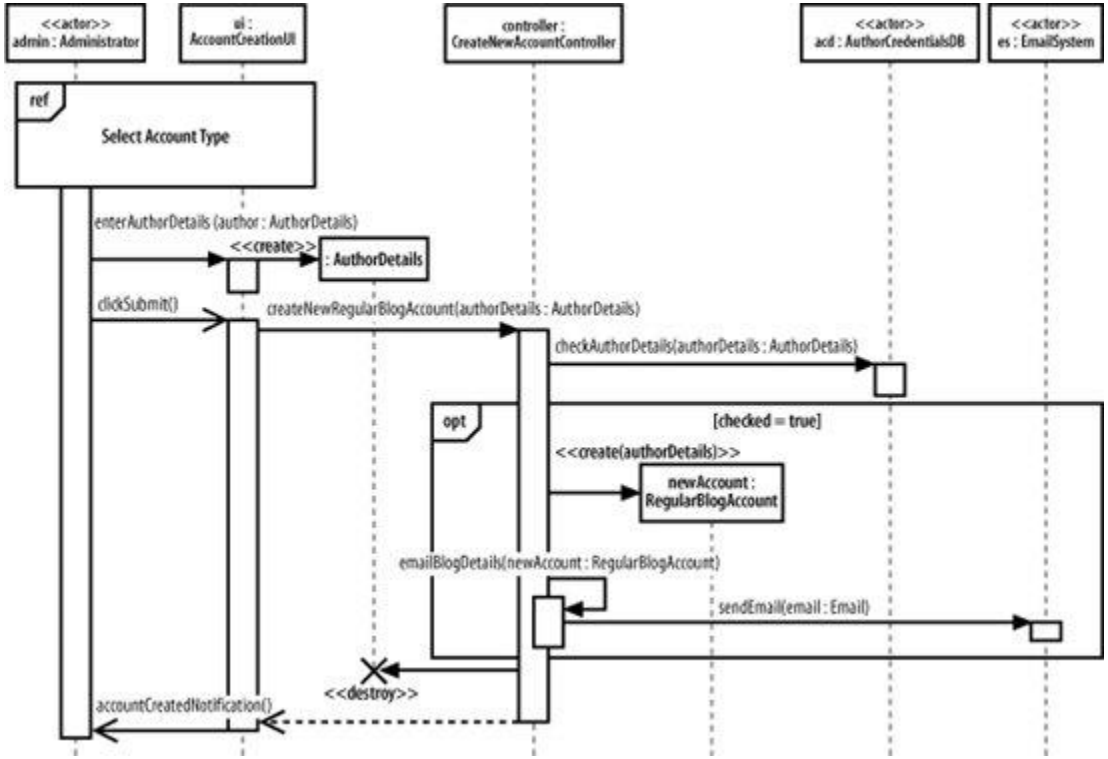
Bucle

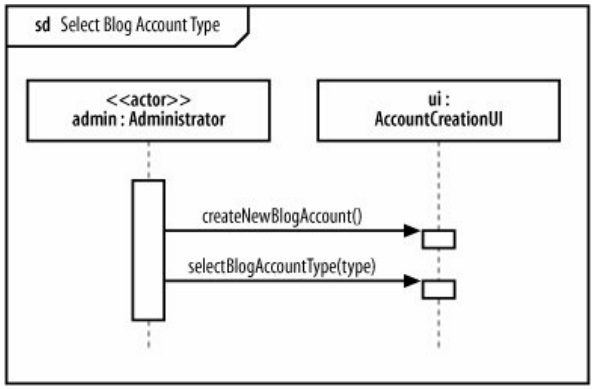

Customer
Cashier
loop
[while items remain]
1.0 unloadItem(itemCost)
1.1 tallyItem(cost)
1.2 requestPayment

Paralelism

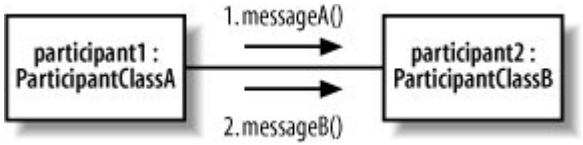


Referințe

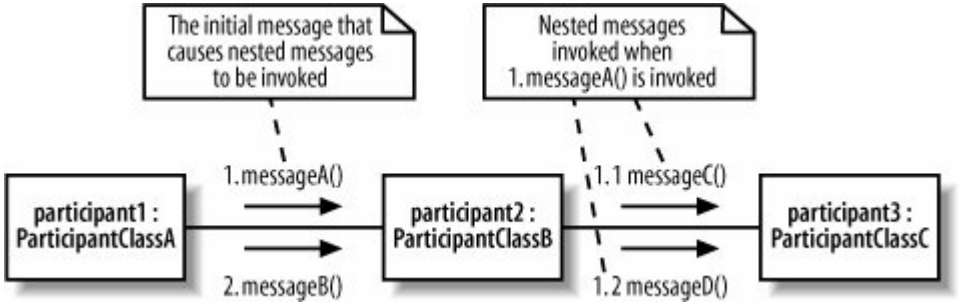

<<actor>>
admin : Administrator
ui :
AccountCreationUI
controller :
CreateNewAccountController
<<actor>>
acd : AuthorCredentialsDB
<<actor>>
es : EmailSystem
ref
Select Account Type
enterAuthorDetails (author : AuthorDetails)
<<create>>
: AuthorDetails
clickSubmit()
createNewRegularBlogAccount(authorDetails : AuthorDetails)
checkAuthorDetails(authorDetails : AuthorDetails)
opt
[checked = true]
<<create(authorDetails)>>
newAccount :
RegularBlogAccount
emailBlogDetails(newAccount : RegularBlogAccount)
sendEmail(email : Email)
<<destroy>>
accountCreatedNotification()


sd Select Blog Account Type
<<actor>>
admin : Administrator
ui :
AccountCreationUI
createNewBlogAccount()
selectBlogAccountType(type)

6. Diagrame de comunicare

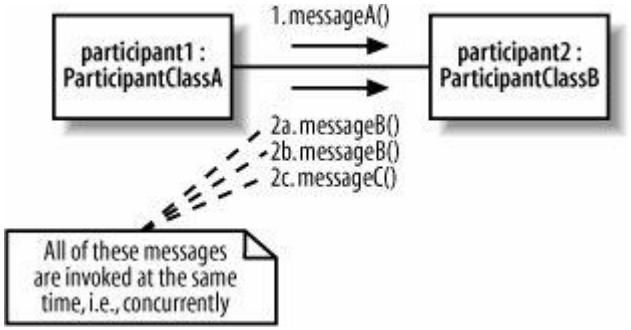


Similare diagramelor de secvențe, dar se concentrează pe legăturile dintre participanți



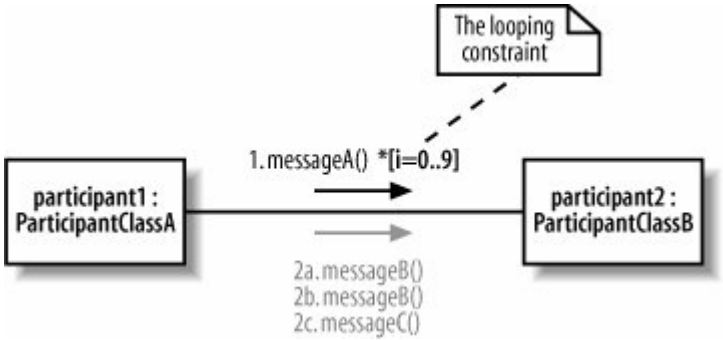
Diagramele de secvențe și cele de comunicare pot fi transformate automat unele în altele

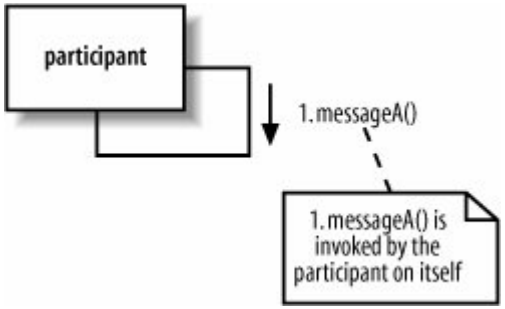
Mesaje imbricate (1 → 1.1, 1.2)

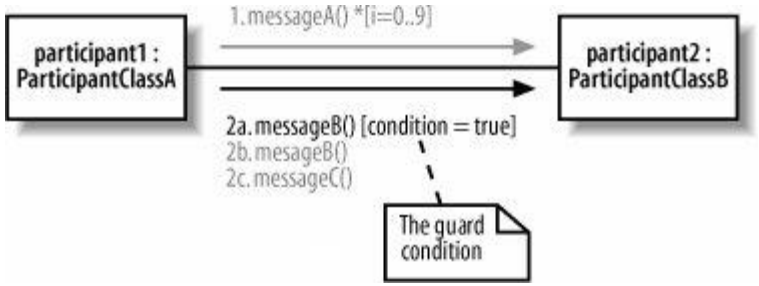


Mesaje concurente (2a, 2b, 2c)

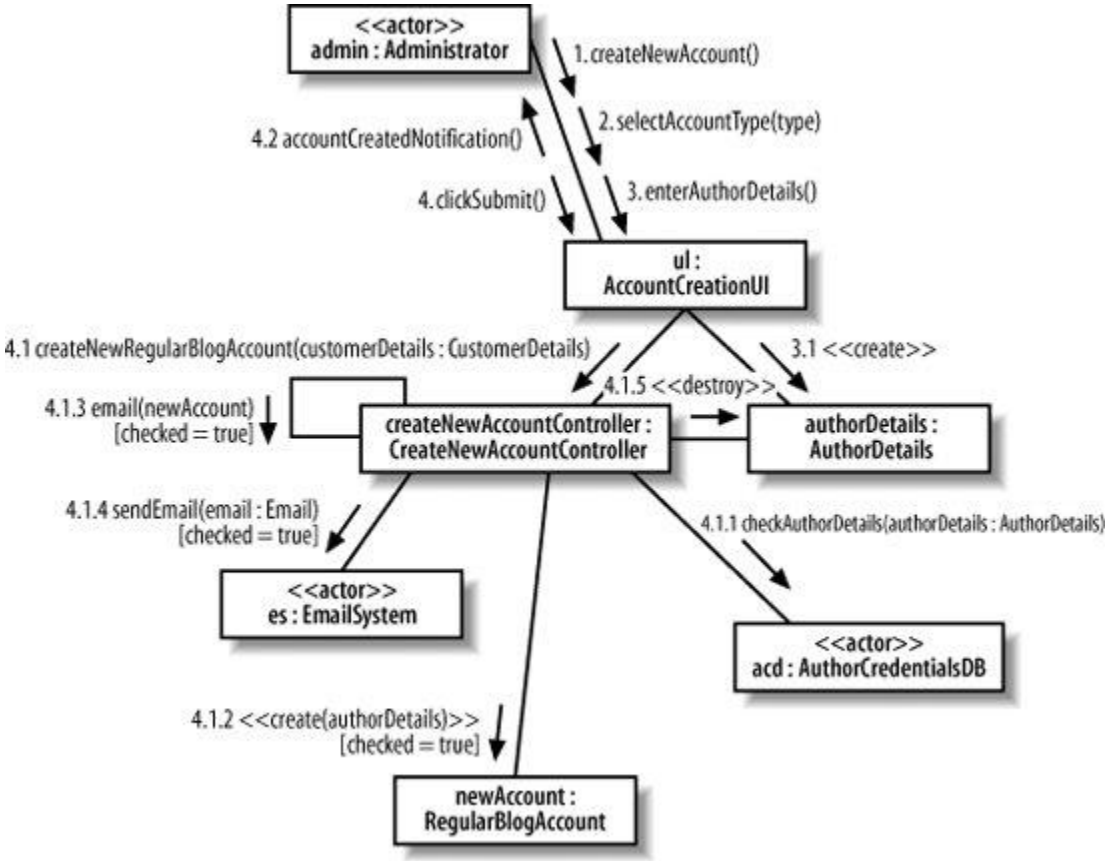
Alte tipuri de mesaje



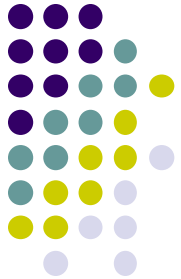




Diagramă complexă

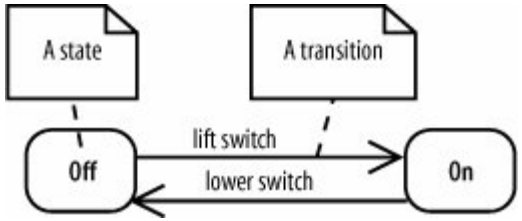


7. Diagrama mașinilor de stare

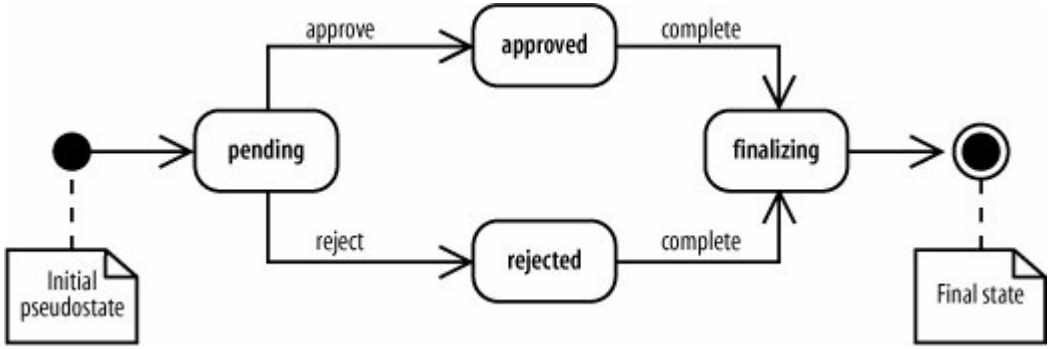


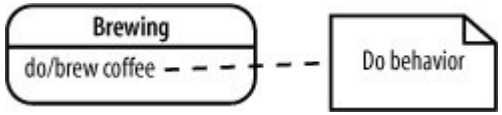
- Obiectele pot reacționa diferit în funcție de stare
- Diagramele mașinilor de stare se realizează pentru un singur obiect
  - Pot descrie comportamentul unui obiect de-a lungul mai multor cazuri de utilizare
  - Nu sunt potrivite pentru a descrie colaborarea mai multor obiecte
- Sunt folosite intens în anumite tipuri de aplicații software și hardware, precum:
  - Sisteme critice de timp real
  - Bancomate
  - Jocuri (de exemplu “first person shooter”)

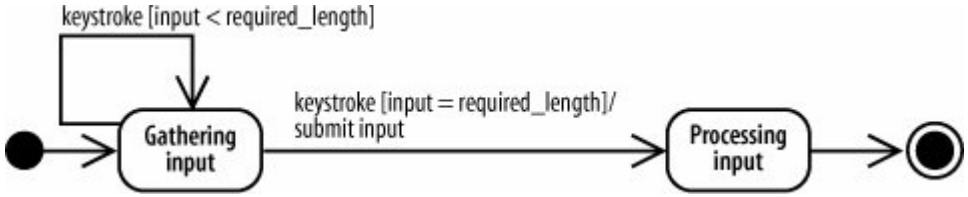
Stări, tranziții, declanșatoare, condiții



lift/lower switch = declanșatoare (triggers)

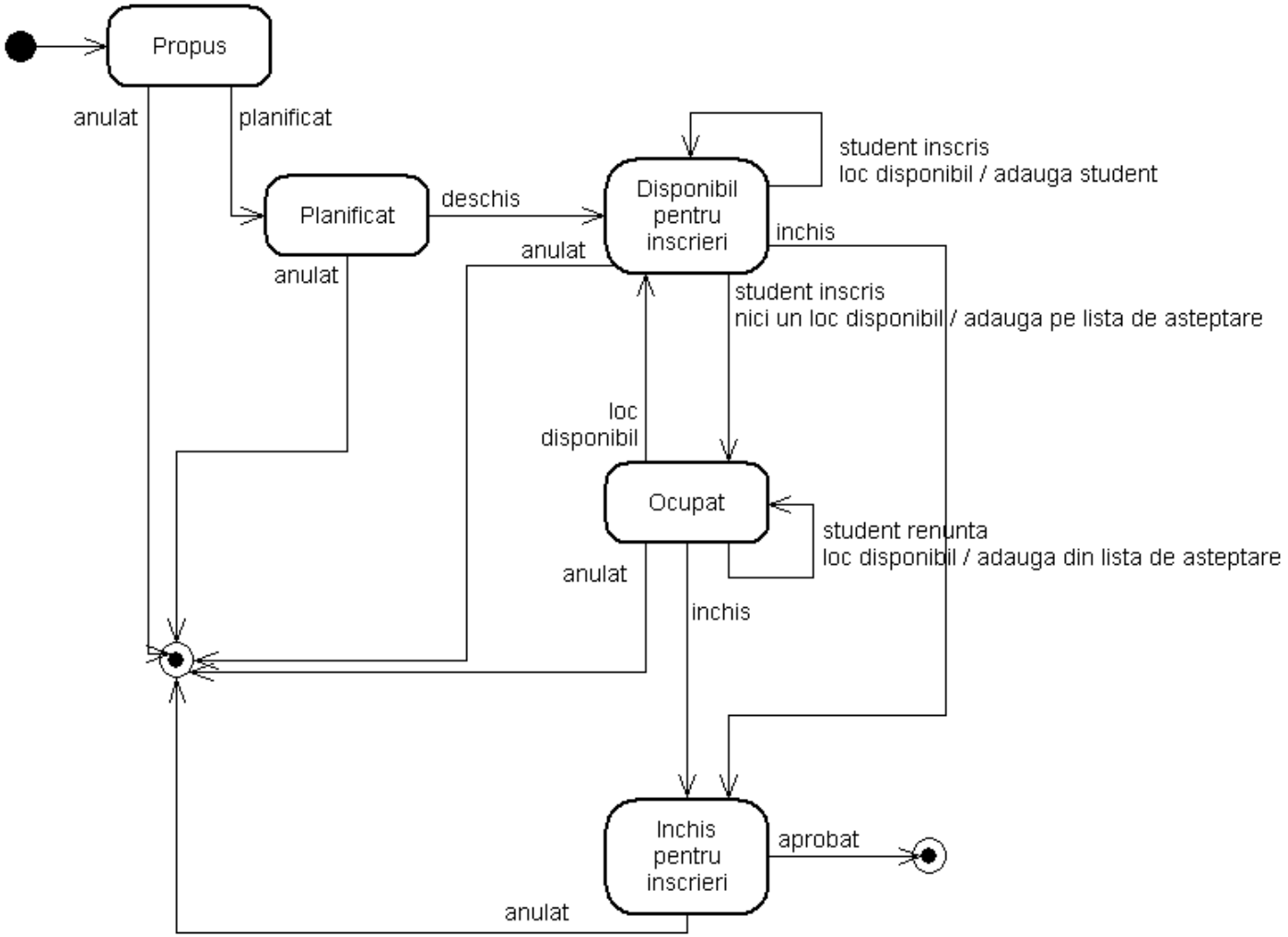




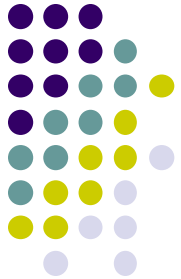


Stări active – ce face (de exemplu prepară cafeaua)
Stări pasive – cum este (de exemplu becul e stins sau aprins)

Diagramă complexă

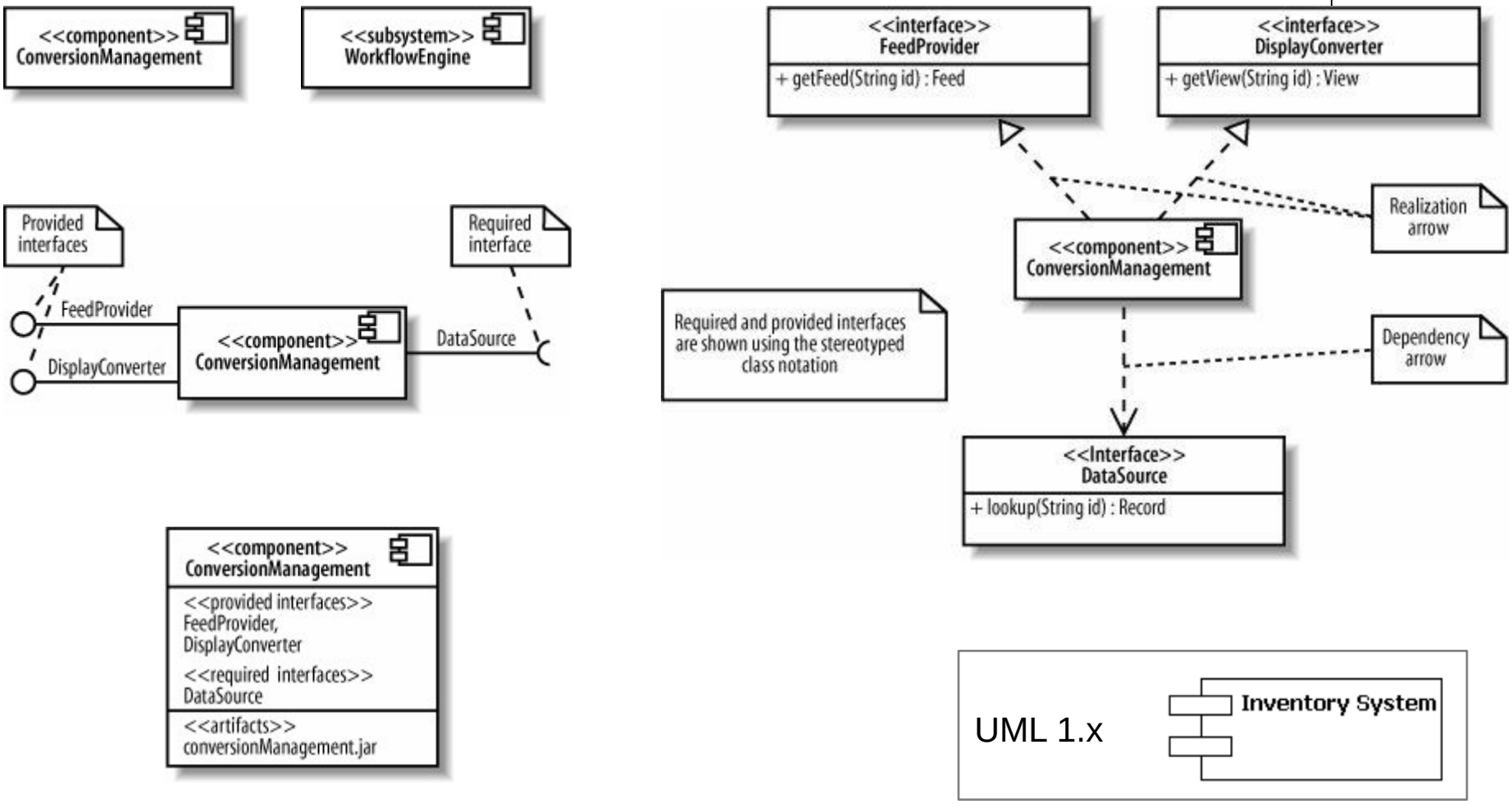


8. Diagrama de componente

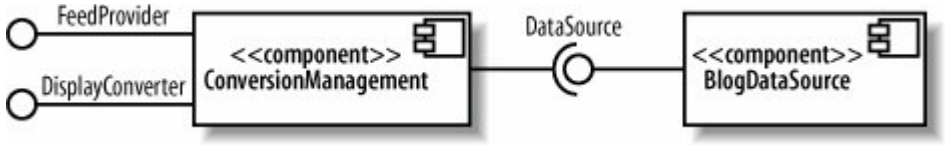


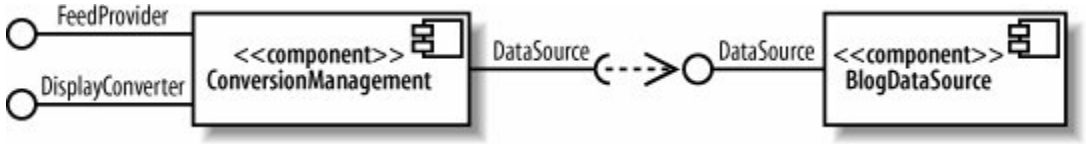
- Pentru proiectele mai complexe, este greu de trecut de la analiză direct la definirea claselor
- O componentă este o parte încapsulată, reutilizabilă și înlocuibilă a sistemului
- Componentele comunică prin interfețe, pentru a asigura cuplarea slabă
- Componentele pot fi compuse din câteva clase sau pot reprezenta subsisteme de mari dimensiuni

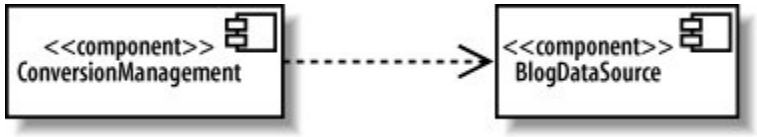
Notații pentru componente



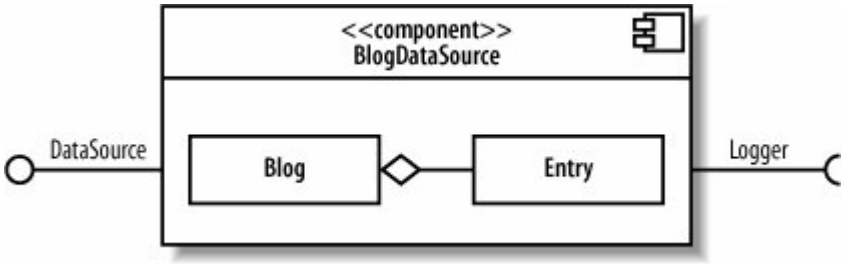
Notații pentru interfețe

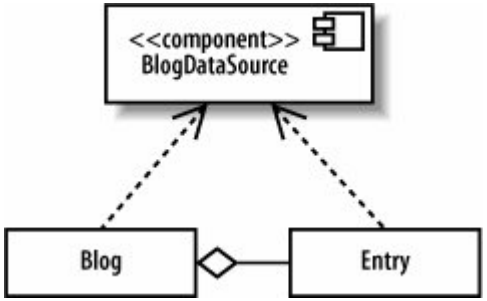


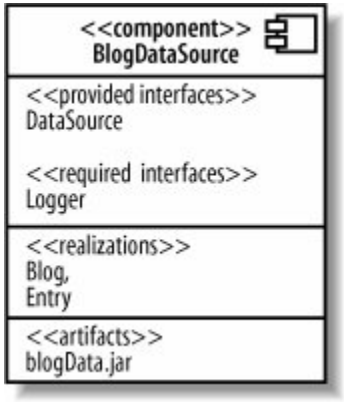




Clasele componentelor

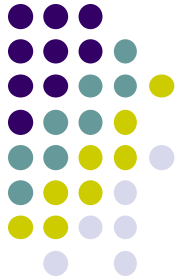






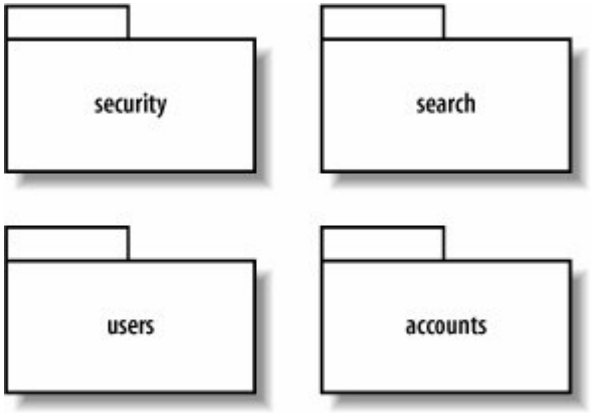
Notație compactă

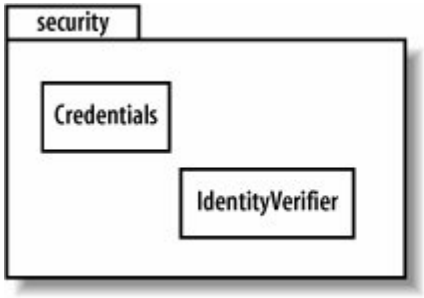
9. Diagrama de pachete

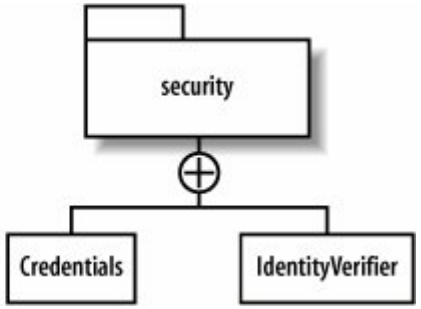


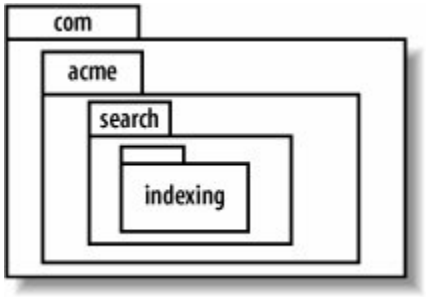
- Sistemele software pot avea sute de clase
- Pachetele modelează grupuri de clase
- Majoritatea limbajelor de programare importante au un corespondent al pachetelor
  - C#: *namespace*
  - Java: *package*
- Deseori, diagramele sunt folosite pentru a indica dependențele dintre pachete
- Pachetele pot cuprinde orice element UML, nu doar clase
  - De exemplu: cazuri de utilizare
- Componentele corespund nivelului conceptual (mai generale)
- Pachetele corespund nivelului logic, în corespondență cu faza de implementare

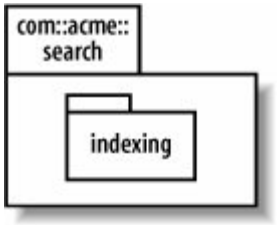
Pachete







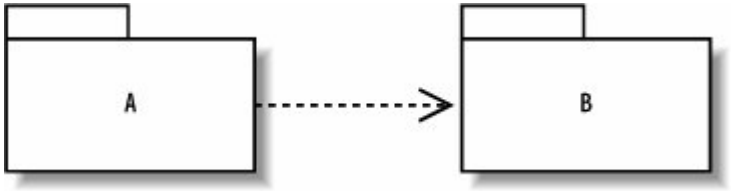


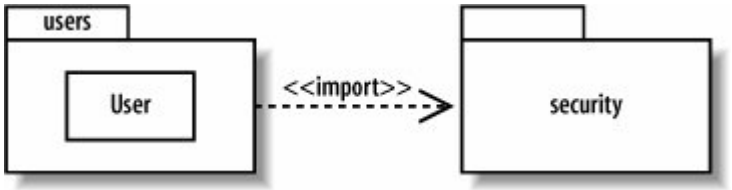


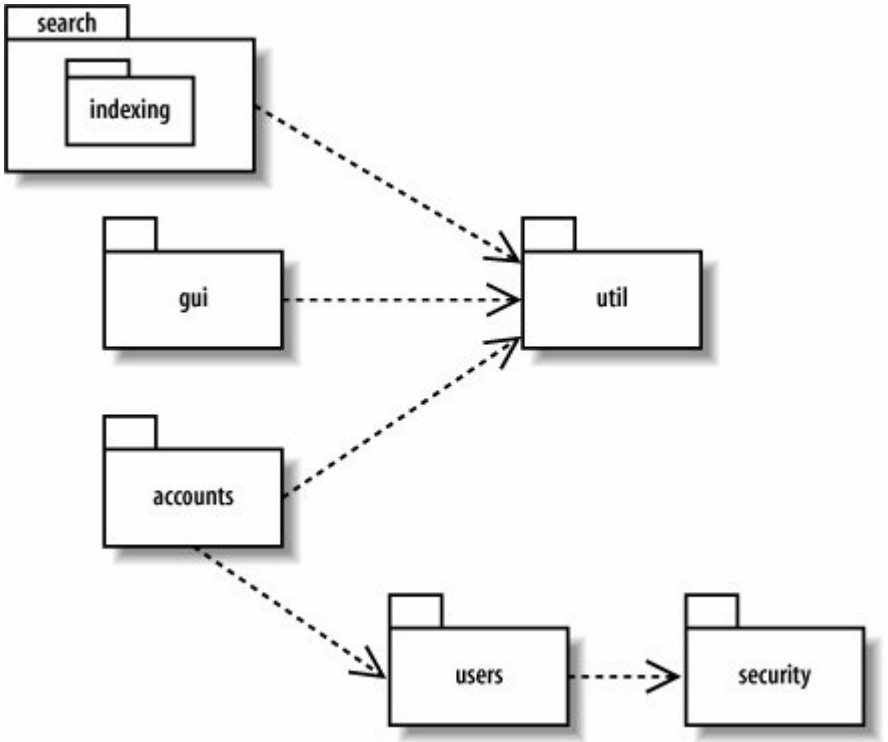
Organizarea logică

Clase în pachete

Dependențe

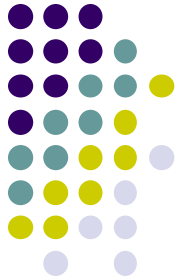






Java: *import*
C#: *using*

10. Diagrama de desfășurare



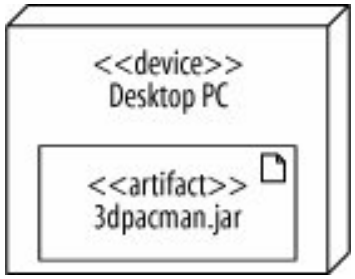
- Corespunde nivelului fizic: modelează elemente fizice ale sistemului
  - Fișiere executabile
  - Entități hardware
- Arată cum sunt atribuite entitățile software către cele hardware și cum comunică acestea
- Poate include hardware, firmware, sisteme de operare, medii de execuție, drivere etc.
  - Dar trebuie să cuprindă doar detaliile importante pentru audiență

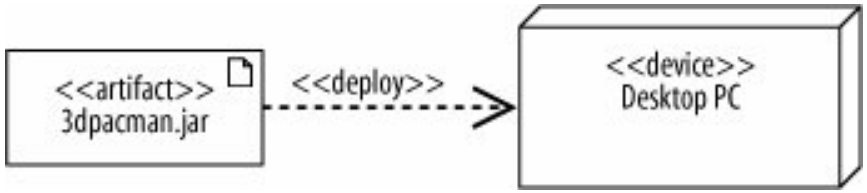
Dispozitive și artefacte

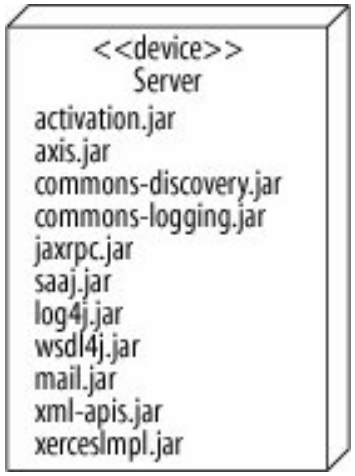
hardware

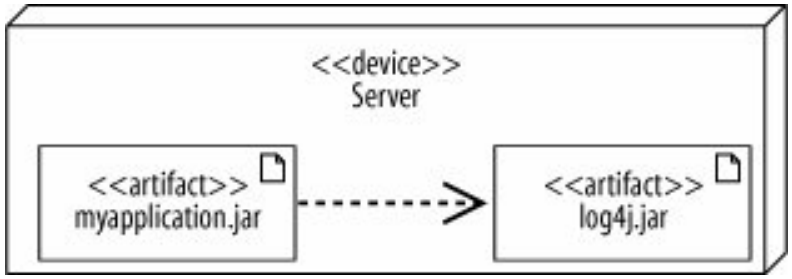
software

Dispozitive și artefacte



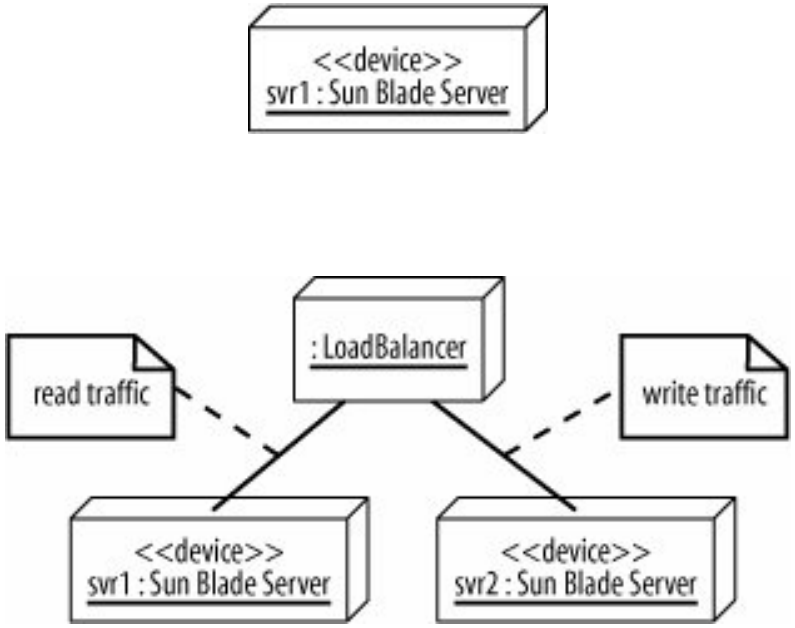




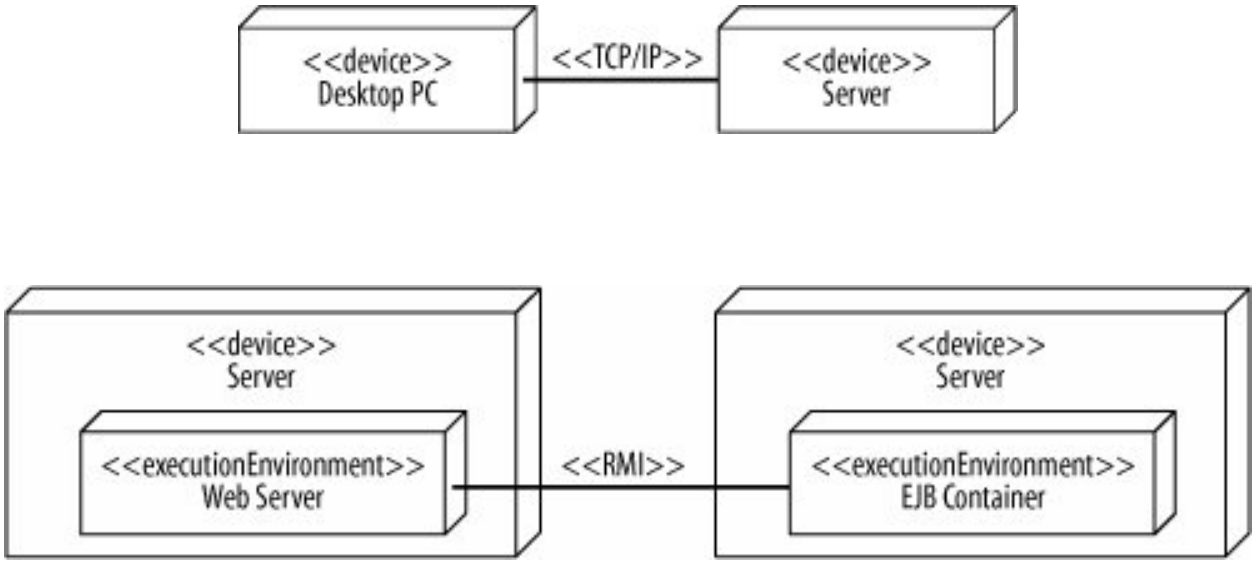


Organizarea fizică

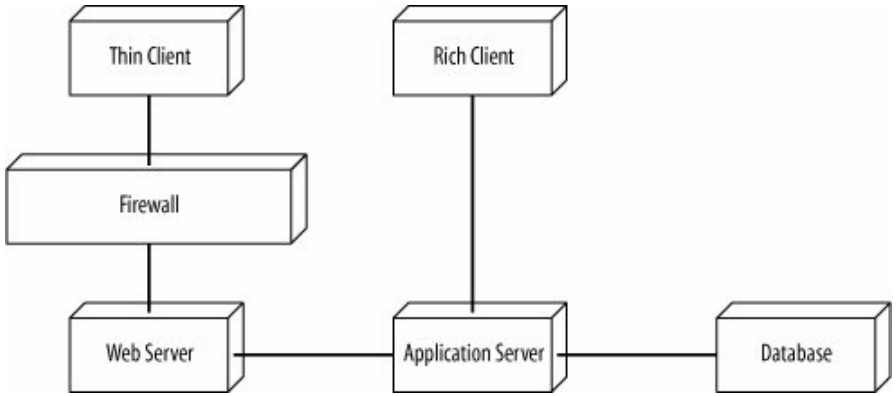
Instanțe de noduri

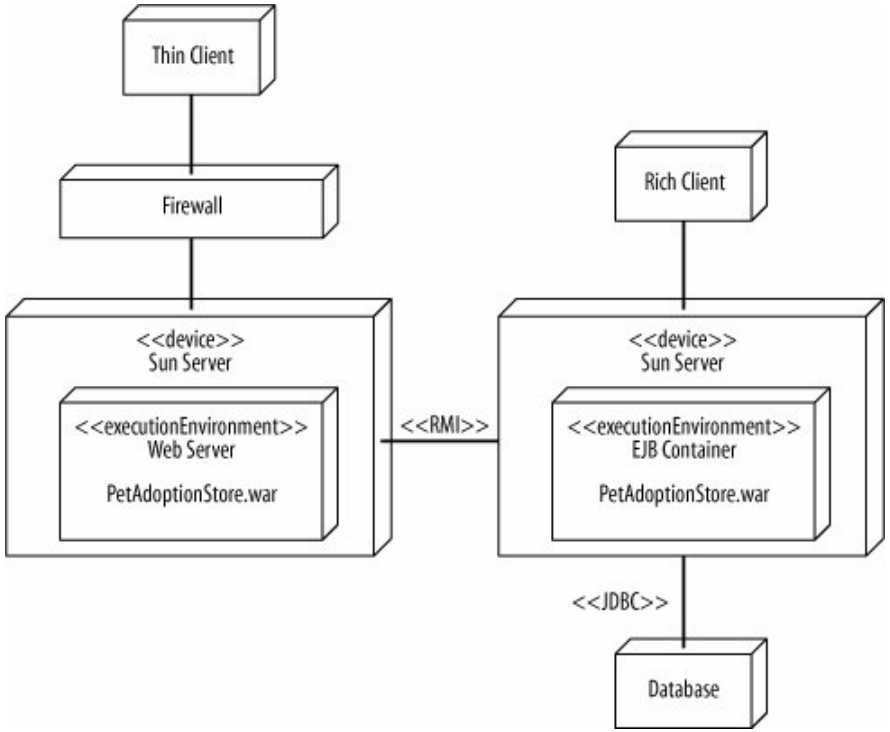


Comunicația între noduri

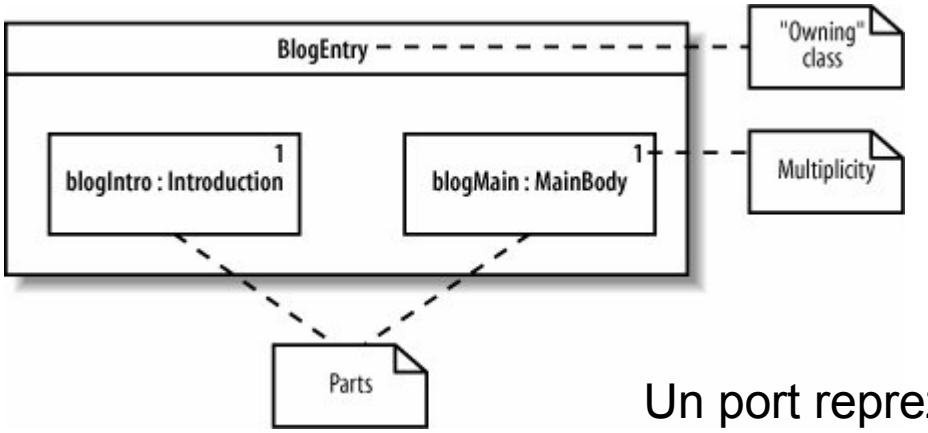


Exemplu


Thin Client
Rich Client
Firewall
Web Server
Application Server
Database

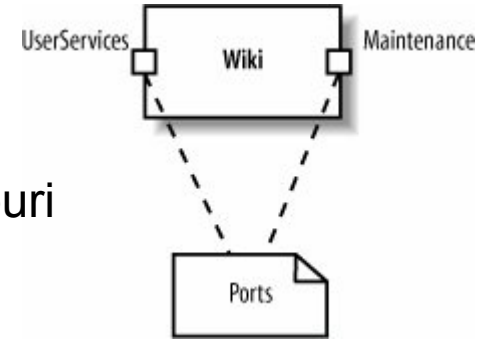

Thin Client
Firewall
Rich Client
<<device>>
Sun Server
<<executionEnvironment>>
Web Server
PetAdoptionStore.war
<<RMI>>
<<device>>
Sun Server
<<executionEnvironment>>
EJB Container
PetAdoptionStore.war
<<JDBC>>
Database

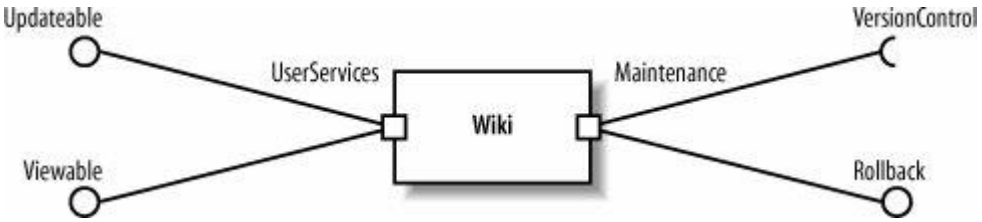
11. Diagrama structurilor compuse



Arată cum lucrează obiectele în interiorul unei clase sau cum îndeplinesc un scop

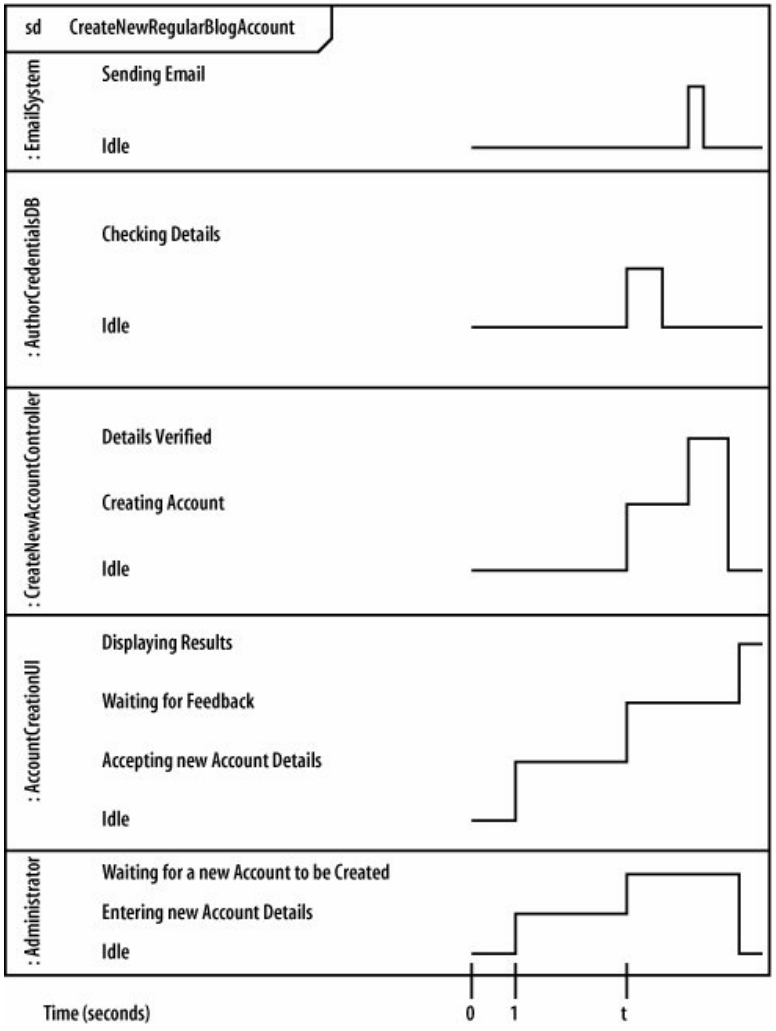
Un port reprezintă un mod distinct de utilizare a unei clase, de obicei de către tipuri diferite de clienți





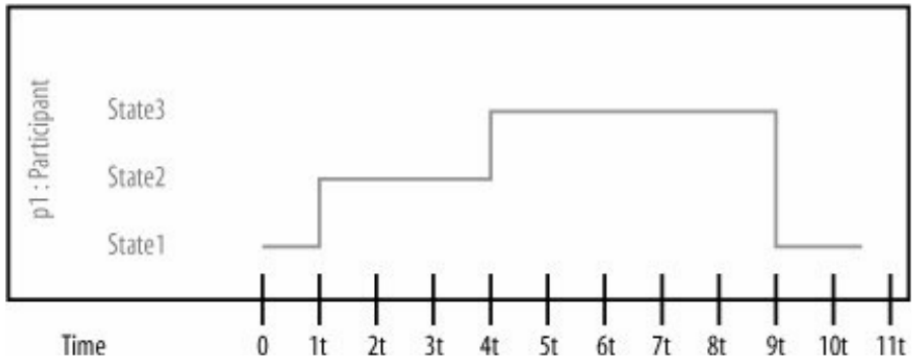
Porturile pot grupa interfețe înrudite

12. Diagrama de cronometrare

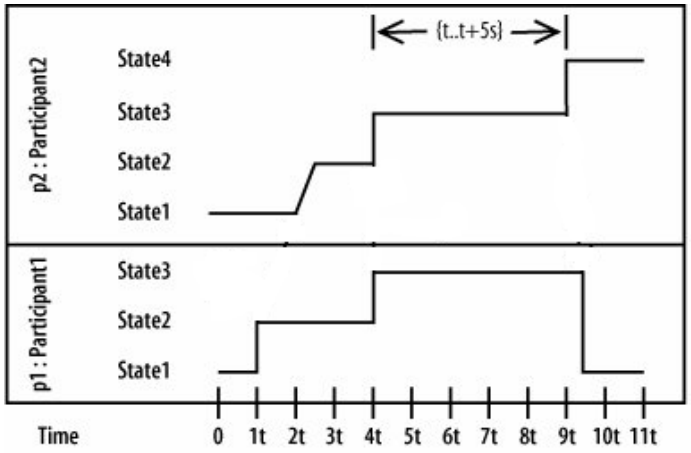


Folosită mai ales pentru sisteme de timp real sau încorporate

Arată stările unui obiect și momentele de timp când se află obiectul în aceste stări

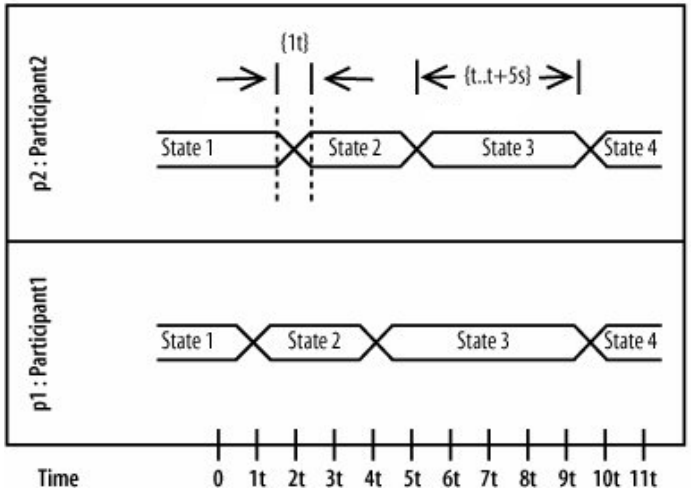


Notații alternative



The regular Timing Diagram notation : good when *fewer states* need to be shown.

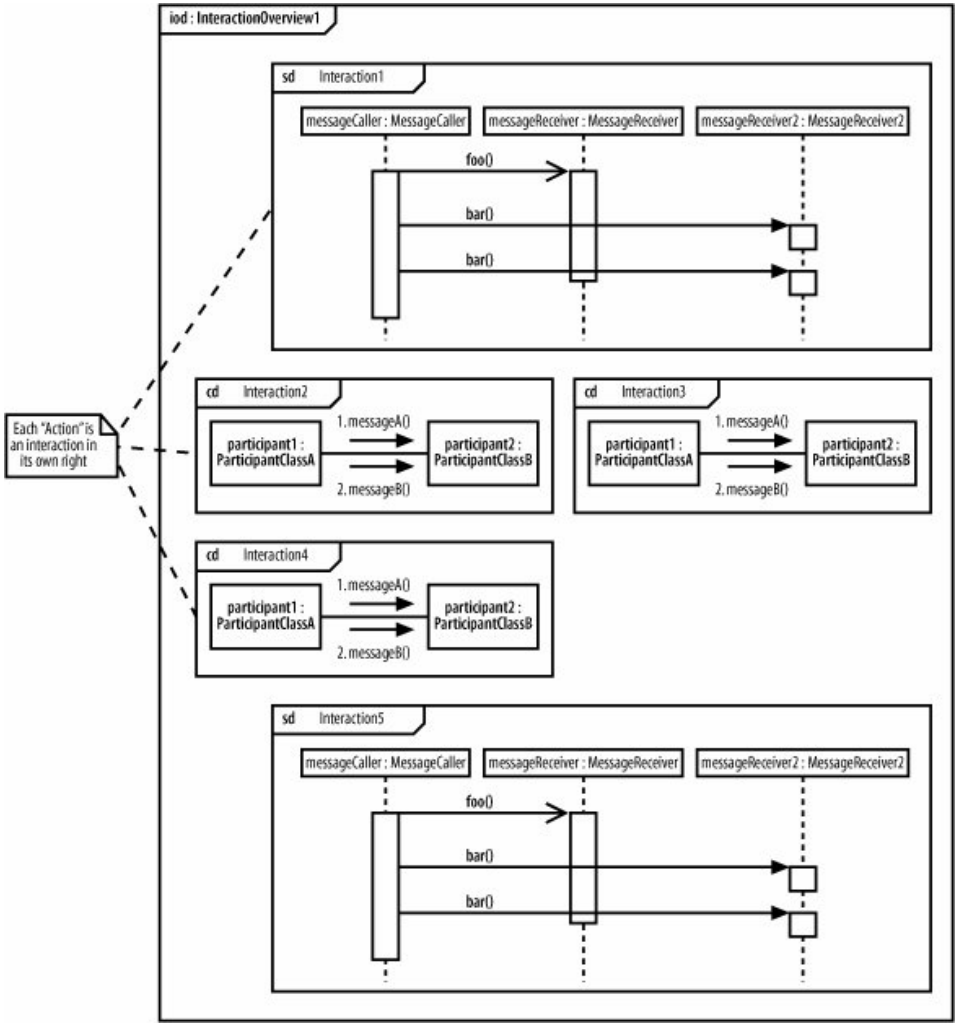
Recomandată pentru puține stări



The alternative Timing Diagram Notation : good when *many states* need to be shown.

Recomandată pentru multe stări

13. Diagrama interacțiunilor generale



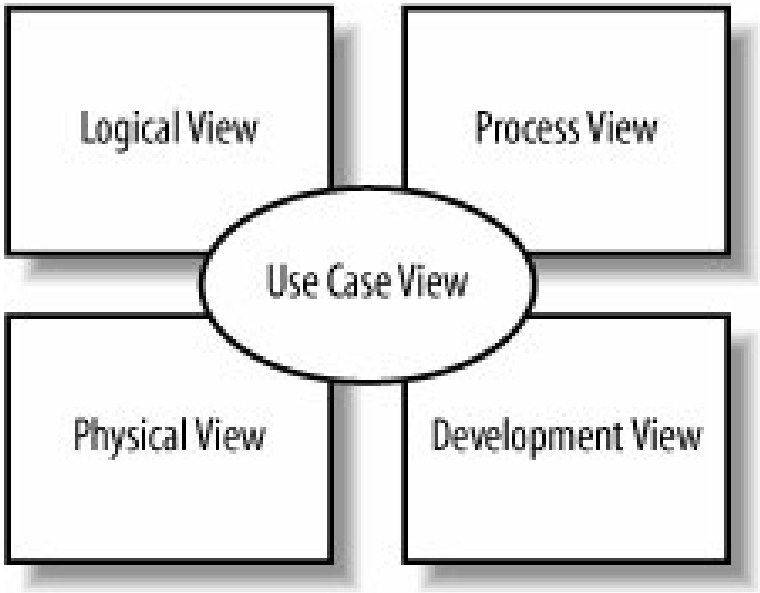
Arată cum interacționează mai multe entități pentru a realiza un caz de utilizare (sau un scop)

Poate fi văzută ca o diagramă de activități, dar fiecare acțiune este o interacțiune completă descrisă de o diagramă distinctă

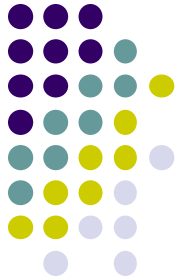
Diagramele UML și ciclul de dezvoltare software

- Perspectiva logică
  - Clase, mașini de stare, secvențe ș.a.
- Perspectiva de proces
  - Activități
- Perspectiva de dezvoltare
  - Pachete, componente
- Perspectiva fizică
  - Desfășurare

- Modelul vizualizării 4+1



Diagramele UML și ciclul de dezvoltare software



- Faza modelării cazurilor de utilizare
- Faza modelării domeniului
  - Diagrame statice de structură, diagrame de secvențe
- Faza modelării proiectării
  - Diagrame de clase, de mașini de stare, de activități
- Faza modelării implementării
  - Diagrame de componente, de desfășurare

Concluzii

- UML este un limbaj pentru specificarea, vizualizarea, construirea și documentarea elementelor sistemelor software
- Este un standard *de facto* pentru modelarea software
- UML 2.0 are 13 diagrame, clasificate în:
  - Diagrame de structură
  - Diagrame de comportament
  - Diagrame de interacțiune

Referințe

- Majoritatea diagramelor incluse în acest curs au fost preluate din următoarele surse:
  - Hamilton, K., Miles, R. (2006). *Learning UML 2.0*, O'Reilly
  - Ariadne Training. *UML Applied*, 2nd edition, http://ariadnetraining.com/software-courses/images/stories/YesNo/ariadne/file/UML_Applied_Second_Edition.pdf