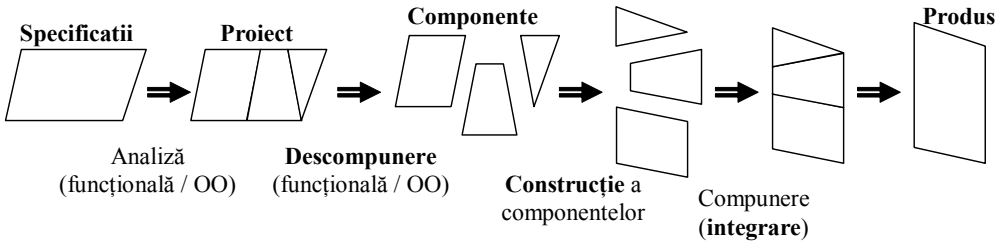
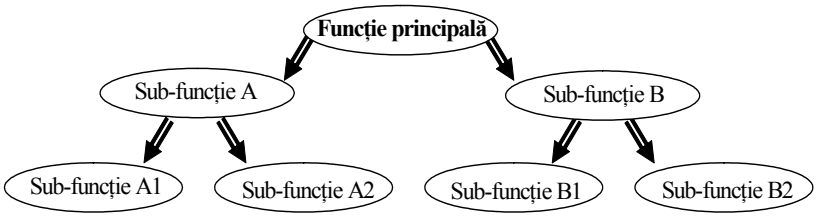
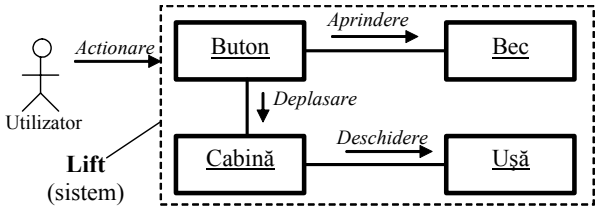
Proces de construcţie prin descompunere-reunire (*top-down*) generic



Descompunere funcţională ierarhică



Descompunere în "societati" de obiecte



1. Clasa = tipul (domeniul de definitie) pentru variabile numite **obiecte**

2. Obiectul = **reprezentare abstractă** a unor **entităţi** reale sau virtuale, caracterizată de:
- ***identitate***, prin care e deosebit de alte obiecte, implementata ca **variabila referinta la obiect**,
- ***comportament* (vizibil)**, accesibil altor obiecte, implementata ca **set de functii membru** = **operatii, metode**,
- ***stare internă* (ascunsă)**, proprie obiectului, implementata ca **set de variabile membru** = **atribute, campuri**.

3. Definitia clasei in Java:

```
1    class Nume {              // declaratie tip de date / structura de date
2       tip atribut;           // declaratie variabila membru, camp Java
3
4       tipReturnat operatie(tipParametru parametruFormal) {          // semnatura metoda Java
5           // corpul functiei membru (metodei) – returneaza valoare de tipul tipReturnat
6       }
7    }
```

4. Declararea variabilei referinta la un obiect Java si crearea dinamica a obiectului:

```
NumeClasa  numeObiect;                          // declararea variabilei referinta la obiect
numeObiect = new NumeClasa(tipParametru parametruActual);  // crearea dinamica a obiectului
```

- UML: **clasa**

Nume
atribut : tip
operatie(parametruFormal : tipParametru) : tipReturnat

obiectul: **numeObiect** : NumeClasa

```
1   public class Point {
2      // atribute (variabile membru)
3      private int x;
4      private int y;
5      // operatie care initializeaza atributele = constructor Java
6      public Point(int abscisa, int ordonata) {
7         x = abscisa;
8         y = ordonata;
9      }
10     // operatii care modifica atributele = metode (functii membru)
11     public void moveTo(int abscisaNoua, int ordonataNoua) {
12        x = abscisaNoua;
13        y = ordonataNoua;
14     }
15     public void moveWith(int deplasareAbsc, int deplasareOrd) {
16        x = x + deplasareAbsc;
17        y = y + deplasareOrd;
18     }
19     // operatii prin care se obtin valorile atributelor = metode Java
20     public int getX() {  return x;  }
21     public int getY() {  return y;  }
22  }
```

Declaratii (specificare) variabile

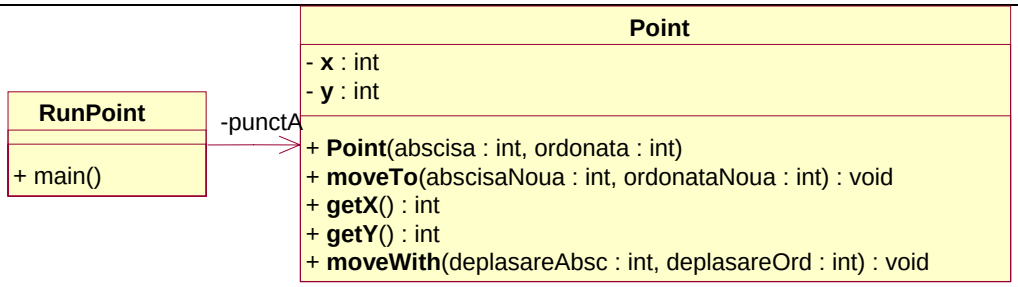
Semnaturi (declaratii, specificari) operatii

+

Implementari (corpuri) operatii

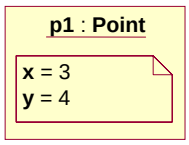
```
1   public class RunPoint {        // clasa de test pentru clasa Point
2      private static Point punctA;                    // atribut de tip Point
3
4      public static void main(String[] args) {        // declaratie metoda
5                                                      // corp metoda
6        punctA = new Point(3, 4);  // alocare si initializare atribut punctA
7        punctA.moveTo(3, 5);       // trimitere mesaj moveTo() catre punctA
8        punctA.moveWith(3, 5);     // trimitere mesaj moveWith() catre punctA
9     }
```

```
10    }
```



5. Crearea unui obiect `p1` de tip **`Point`** ale carui **atribute** au **valorile `x=3`** si respectiv **`y=4`** :

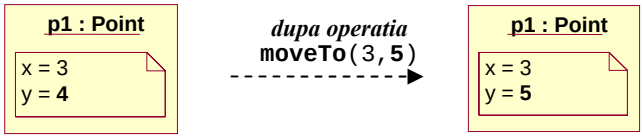
```
Point p1 = new Point(3, 4);
```



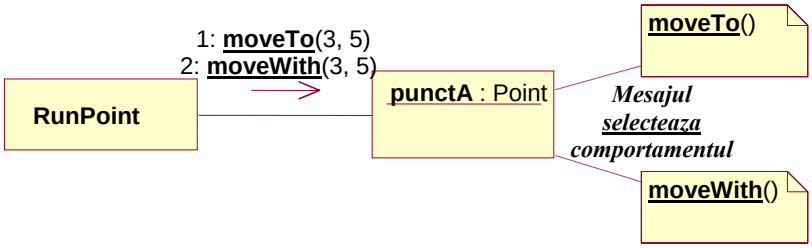
Obiectul `p1` de tip `Point` incapsuleaza informatiile unui punct in plan de coordonate {3, 4}.
Starea obiectului `p1` este perechea de coordonate **{3, 4}**.

6. Schimbarea starii obiectului `p1` in {3, 5}, prin deplasarea ordonatei (departarea cu 1 de abscisa).

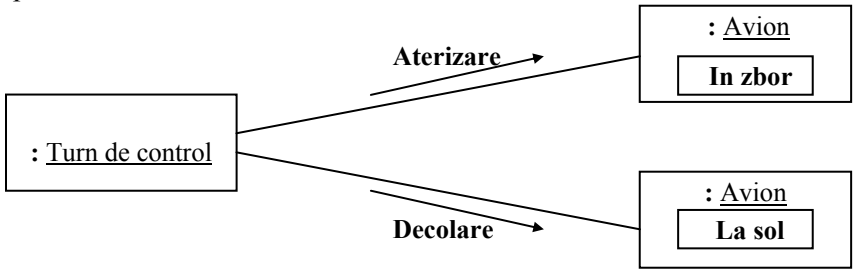
```
Point p1 = new Point(3, 4);
p1.moveTo(3, 5);
```



7. Mesajul selecteaza operatia si declanseaza comportamentul (activeaza operatia, prin invocarea / apelul metodei / functiei membru):

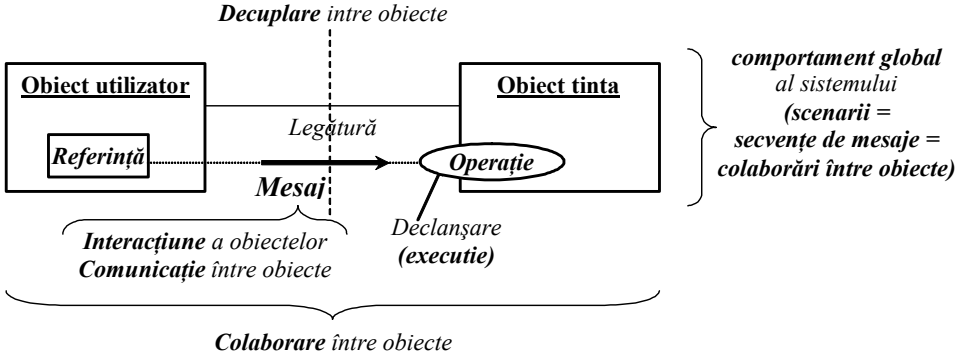


8. Starea si comportamentul sunt **dependente**. Comportamentul la un moment dat depinde de starea curenta. Starea poate fi modificata prin comportament.



In starea **"In zbor" doar comportamentul Aterizare e posibil**. El **duce la schimbarea starii** (in "La sol").
Dupa aterizare, starea fiind "La sol", operatia Aterizare nu mai are sens.

9. Sistemele software OO sunt **societăți de obiecte care colaborează** pentru a realiza funcțiile aplicației.



Elementele declaratiilor:

```
1   import java.util.Vector;                    // clase importate
2   import java.util.EmptyStackException;
3
4   public class Stack                  // declaratia clasei
5   {                                   // inceputul corpului clasei
6
7     private Vector elemente;                  // atribut (variabila membru)
8
9     public Stack() {                          // constructor
10      elemente = new Vector(10);              // (functie de initializare)
11    }
12
13    public Object push(Object element) {      // metoda
14      elemente.addElement(element);           // (functie membru)
15      return element;
16    }
17
18    public synchronized Object pop(){         // metoda
19      int lungime = elemente.size();          // (functie membru)
20      Object element = null;
21      if (lungime == 0)
22        throw new EmptyStackException();
23      element = elemente.elementAt(lungime - 1);
24      elemente.removeElementAt(lungime - 1);
25      return element;
26    }
27
28    public boolean isEmpty(){                 // metoda
29      if (elemente.size() == 0)               // (functie membru)
30        return true;
31      else
32        return false;
33    }
34  }                                   // sfarsitul corpului clasei
35
```

Declaratia de clasa

```
[public] [abstract] [final] class NumeClasa [extends NumeSuperclasa]
                                        [implements  NumeInterfata [, NumeInterfata]] {
  // Corp clasa
}
```

Declaratia de camp (atribut)

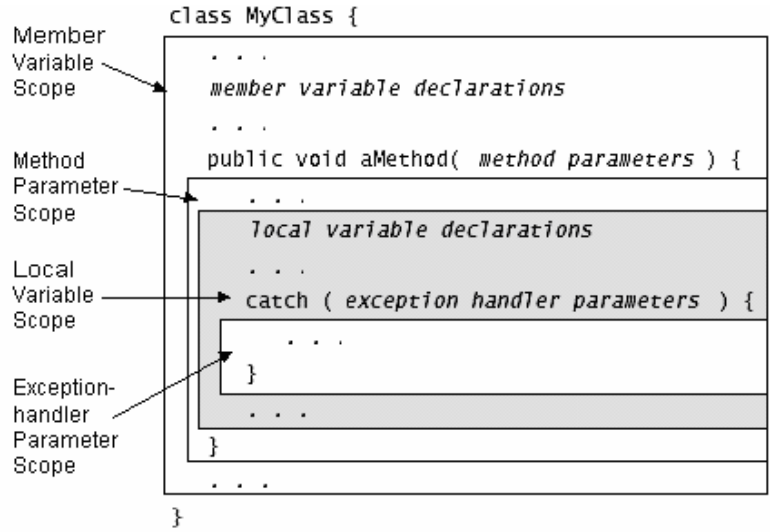
```
[nivelAcces] [static] [final] tipAtribut numeAtribut;
```

Declaratia de constructor

```
[nivelAcces] NumeClasa( listaParametri ) {
  // Corp constructor
}
```

Declaratia de metoda (operatie)

```
[nivelAcces] [static] [abstract] [final] [synchronized] tipReturnat numeMetoda (
                          [listaDeParametri] ) [throws NumeExceptie [,NumeExceptie] ] {
  // Corp metoda
}
```

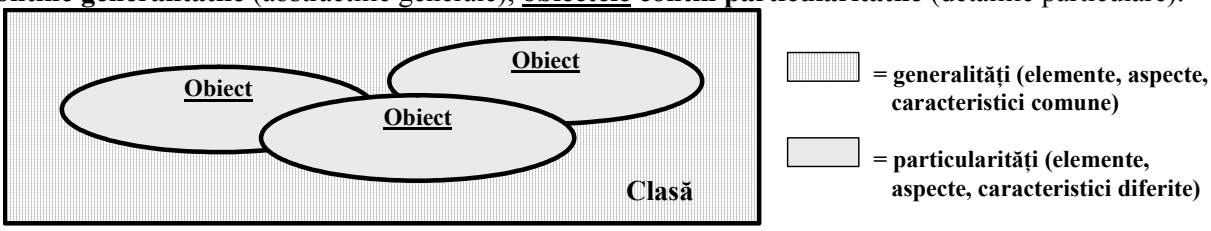


Scopul variabilelor

Exemplu

```
1   public class Complex        // declaratia clasei
2   {                           // inceputul corpului clasei
3     private double real;            // real = atribut (camp)
4     private double imag;            // imag = atribut (camp)
5
6     public void setReal(double real) {    // metoda, real = parametru
7       this.real = real;                   // real = atributul, real = parametrul
8     }
9
10    public void.setImag(double imag) {    // metoda, imag = parametru
11      this.imag = imag;                   // imag = atributul, imag = parametrul
12    }
13
14    public static void main(String[] args) {      // metoda, args = parametru
15      double real = Double.parseDouble(args[0]);  // real = variabila locala
16      double imag = Double.parseDouble(args[1]);  // imag = variabila locala
17
18      Complex c = new Complex();                  // c = variabila locala
19
20      c.setReal(real);                            // c, real = variabilele locale
21      c.setImag(imag);                            // c, imag = variabilele locale
22
23      System.out.println("{" + c.real +           // c.real = atributul lui c
24                         ", " + c.imag + "}");    // c.imag = atributul lui c
25    }
26  }                           // sfarsitul corpului clasei
```

Clasa contine generalitatile (abstractiile generale), **obiectele contin particularitatile** (detaliile particulare).



O parte din codul clasei `DatagramPacket` (continuturile anumitor metode au fost simplificate):

```
1   package java.net; // Importul pachetului de clase Java pentru comunicatii IP
2   public final class DatagramPacket {  // incapsuleaza pachete UDP (datagrame)
3     // Atribute, accesibile tuturor claselor din pachetul (directorul) java.net
4     byte[] buf;           // tabloul de octeti care contine datele pachetului
5     int offset;           // indexul de la care incep datele pachetului
6     int length;           // numarul de octeti de date din tabloul de octeti
7     int bufLength;        // lungimea tabloului de octeti
8     InetAddress address;  // adresa IP a masinii sursa/destinatie a pachetului
9     int port;             // portul UDP al masinii sursa/destinatie a pachetului
10
11    // Constructori - initializeaza obiectele de tip DatagramPacket
12    // Initializeaza un DatagramPacket pentru pachete de receptionat,
13    public DatagramPacket(byte buf[], int length) {
14       this.buf = buf;
15       this.length = length;
16       this.bufLength = length;
17       this.offset = 0;
18       this.address = null;
19       this.port = -1;
20    }
21    // Initializeaza un DatagramPacket pentru pachete de trimis
22    public DatagramPacket(byte buf[], int length, InetAddress address, int port) {
23       this.buf = buf;
24       this.length = length;
25       this.bufLength = length;
26       this.offset = 0;
27       this.address = address;
28       this.port = port;
29    }                          // Alti constructori, alte metode...
30    // Returneaza adresa IP a masinii sursa/destinatie a acestui pachet
31    public synchronized InetAddress getAddress() {  return this.address;  }
32
33    // Stabileste adresa IP a masinii sursa/destinatie a acestui pachet
34    public void setAddress(InetAddress iaddr) {  this.address = iaddr;   }
35  }
```

Cod care utilizeaza clasa `DatagramPacket`:

```
byte[] buffer = new byte[1024];                    // buffer date (de trimis/primit)
DatagramPacket packetDeTrimis = new DatagramPacket(buffer, buffer.length,
                          InetAddress.getByName("nume.elcom.pub.ro"), 2000);
DatagramPacket packetDeReceptionat = new DatagramPacket(buffer, buffer.length);
```

Specificatia generala a unei clase `Complex` (interfata publica) = **API-ul** (*Application Programming Interface*):

```
1   public class Complex {
2     // Atribute private (ascunse, inaccesibile din exteriorul clasei)
3     // Constructor – initializeaza obiectele de tip Complex
4     public Complex(double real, double imag) {  // Implementare  }
5     public double getReal() {  // Implementare  }    // Returneaza partea reala
6     public double getImag() {  // Implementare  }    // Returneaza partea imaginara
7     public double getModul() {  // Implementare  }    // Returneaza modulul
8     public double getFaza() {  // Implementare  }    // Returneaza faza
9   }
```

Urmatorul cod Java va conduce la acelasi rezultat, indiferent de implementarea clasei `Complex`:

```
Complex c1 = new Complex(2, -2);
System.out.println("Coordonatele carteziene: {" + c1.getReal() + ", " + c1.getImag() + "}");
System.out.println("Coordonatele polare: {" + c1.getModul() + ", " + c1.getFaza() + "}");
```

Implementarea carteziana a clasei `Complex` (atributele ascunse sunt coordonatele carteziene):

```
1   public class Complex {
2     private double real;          // partea reala (abscisa)
3     private double imag;          // partea imaginara (ordonata)
4     public Complex(double real, double imag) {
5        this.real = real;
6        this.imag = imag;
7     }
8     public double getReal() {       return this.real;    }
9     public double getImag() {       return this.imag;    }
10    public double getModul() {
11       return Math.sqrt(this.real*this.real + this.imag*this.imag);
12    }
13    public double getFaza() {
14       return Math.atan2(this.real, this.imag);
15    }
16  }
```

Implementarea polara a clasei `Complex` (atributele ascunse sunt coordonatele polare):

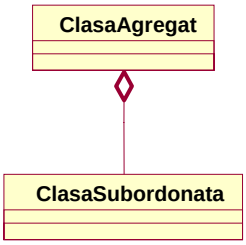
```
1   public class Complex {
2     private double modul;             // modulul (raza)
3     private double faza;              // faza (unghiul)
4     public Complex(double real, double imag) {
5        this.modul = Math.sqrt(real*real + imag*imag);
6        this.faza = Math.atan2(real, imag);
7     }
8     public double getReal() {    return this.modul*Math.cos(this.faza);    }
9     public double getImag() {    return this.modul*Math.sin(this.faza);    }
10    public double getModul() {       return this.modul;    }
11    public double getFaza() {        return this.faza;    }
12  }
```

In cazul de mai sus:

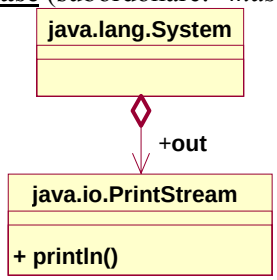
- **specificația nu este afectată de schimbarea reprezentării interne** (polară sau carteziană),
- **obiectele utilizator** al obiectelor instanțe ale clasei numerelor complexe **cunosc doar specificația și nu sunt nici ele afectate de schimbarea reprezentării interne**.

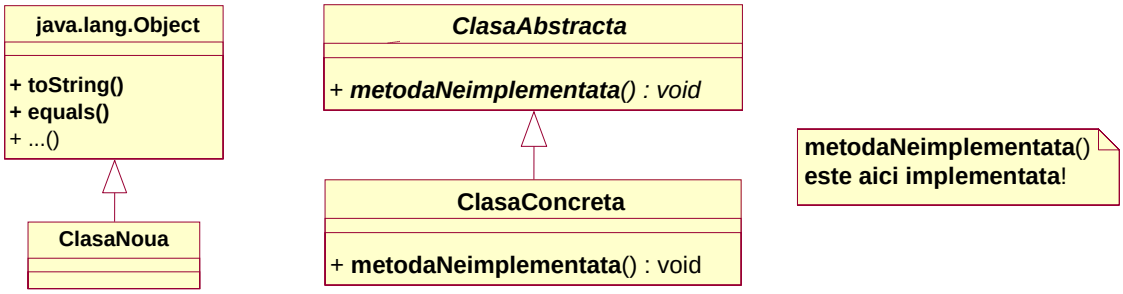
Agregarea exprimă cuplajul strâns între clase (subordonare: "*master-slave*", "întreg-părți", "ansamblu-componentă").

In UML:



Exemplu:





Mostenirea in UML:

Declaratia:

```
class NumeClasa { // urmeaza corpul clasei ...
```

este **echivalenta cu**:

```
class NumeClasa extends Object { // urmeaza corpul clasei ...
```

Metoda `toString()`

```
1   // Implementarea implicita a metodei toString(), mostenita de la clasa Object
2
3   public String toString() {
4      // (nu returneaza continutul ci numele clasei si codul obiectului!)
5      return getClass().getName() + "@" + Integer.toHexString(hashCode());
6   }
```

Metoda `equals()` implicita

```
1   // Implementarea implicita a metodei equals(), mostenita de la clasa Object
2
3   public boolean equals(Object obj) {
4     return (this == obj);     // (nu compara continutul ci referintele!!!)
5   }
```

Metoda `equals()` din clasa `String`

```
 1   // Implementarea explicita a metodei equals() in clasa String
 2
 3   public boolean equals(Object obj) {
 4     // se verifica existenta unui parametru (obiect) non-null
 5     // si faptul ca parametrul e obiect al clasei String
 6     if ((obj != null) && (obj instanceof String)) {
 7       String otherString = (String)obj;          // conversie de tip
 8       int n = this.count;
 9       if (n == otherString.count) {   // se compara numarul de caractere
10         char v1[] = this.value;
11         char v2[] = otherString.value;
12         int i = this.offset;
13         int j = otherString.offset;
14         while (n-- != 0)
15           if (v1[i++] != v2[j++]) return false; // se compara caracterele
16         return true;
17       }
18     }
19     return false;
20   }
```

Clasa abstracta, de baza

```
 1  public abstract class Multime {                  // clasa declarata abstract
 2
 3    protected Object[] elemente;                   // campuri declarate protected pentru a
 4    protected byte numarElem;                      // putea fi reutilizate in clasa derivata
 5
 6    public Multime(Object[] elemente) {            // parametru generic tip Object[]
 7        this.elemente = elemente;                  // acces la obiectul curent cu this
 8        numarElem = (byte) elemente.length;        // conversie de tip de la int la byte
 9    }
10    public abstract Multime intersectieCu(Multime m); // declarata abstract (NEIMPLEMENTATA)
11                                                   // valoare returnata generica tip Multime
12    public Object[] obtinereElemente() {           // valoare returnata generica tip Object[]
13        return elemente;                           // (MOSTENITA in clasa MultimeIntregi)
14    }
15    public byte numarElemente() {                  // implementare de baza
16        return numarElem;                          // (RESCRISA in clasa MultimeIntregi)
17    }
18  }
```

Clasa concreta, derivata

```
 1  public class MultimeIntregi extends Multime {
 2    public MultimeIntregi(Integer[] elemente) {     // parametru concret tip Integer[]
 3        super(elemente);                          // apelul constructorului clasei de baza Multime
 4    }
 5    public final Multime intersectieCu(Multime m) { // declarata abstract in clasa de baza
 6        Integer[] elementeIntersectie;              // aici devine concreta (IMPLEMENTATA)
 7        int nrElemente = 0;
 8        for (int i=0; i< elemente.length; i++)
 9           for (int j=0; j< m.elemente.length; j++)
10              if (elemente[i].equals(m.elemente[j])) nrElemente++;
11
12        int index = 0;
13        elementeIntersectie = new Integer[nrElemente];
14
15        for (int i=0; i< elemente.length; i++)
16          for (int j=0; j< m.elemente.length; j++)
17            if (elemente[i].equals(m.elemente[j]))
18              elementeIntersectie[index++] = new Integer(elemente[i].toString());
19
20        return new MultimeIntregi(elementeIntersectie);
21    }
22    public byte numarElemente() {                   // reimplementare cod (RESCRIERE)
23        return (byte) elemente.length;              // conversie de tip de la int la byte
24    }
25    public boolean contine(int intr) {              // metoda noua (EXTENSIE)
26        for (int i=0; i< elemente.length; i++) {
27          Integer inte = (Integer) elemente[i];
28          if (inte.intValue() == intr) {  return true;  }
29        }
30        return false;
31    }
32  }
```

Extindere prin mostenire, supraincarcare a metodelor, rescriere a metodelor, ascundere a atributelor:

```
 1  public class ClasaDeBaza {
 2    protected int atributMostenit;    // atribut partajat cu subclasa
 3    protected byte atributAscuns;     // atribut corespunzator clasei de baza
 4
 5    public void metodaMostenitaSupraincarcata() {
 6      // implementare corespunzatoare lipsei parametrilor
 7    }
 8    public void metodaMostenitaSupraincarcata(int argument) {
 9      // implementare corespunzatoare parametrului de tip int
10    }
11    public void metodaRescrisa() {
12      // implementare de baza
13    }
14  }
```

```
 1  public class ClasaCareExtinde extends ClasaDeBaza {
 2    protected byte atributAscuns;     // atribut corespunzator subclasei
 3    protected long atributNou;        // atribut nou, nepartajat cu clasa de baza
 4
 5    public void metodaRescrisa() {
 6      // reimplementare (rescriere a codului)
 7    }
 8    public void metodaNoua() {
 9      // metoda noua, nepartajata cu clasa de baza
10    }
11  }
```

```
 1  public class UtilizareClase {
 2    public static void main(String[] args) {
 3
 4      ClasaDeBaza obiectDeBaza = new ClasaDeBaza();     // clasa de baza (extinsa)
 5
 6      obiectDeBaza.atributMostenit    // utilizarea atributului partajat cu subclasa
 7      obiectDeBaza.atributAscuns      // utilizarea atributului din clasa de baza
 8
 9      // apelurile metodelor (din clasa de baza) care supraincarca acelasi nume
10      obiectDeBaza.metodaMostenitaSupraincarcata()
11      obiectDeBaza.metodaMostenitaSupraincarcata(1000)
12
13      // apelul metodei din clasa de baza (implementare de baza)
14      obiectDeBaza.metodaRescrisa()
```

```
15      ClasaCareExtinde obiectExtins = new ClasaCareExtinde();         // subclasa
16
17      obiectExtins.atributNou         // utilizarea atributului nou
18      obiectExtins.atributMostenit    // utilizarea atributului partajat
19      obiectExtins.atributAscuns      // utilizarea atributului din subclasa
20      super.atributAscuns             // utilizarea atributului din clasa de baza
21
22      // apelul metodei noi din subclasa
23      obiectExtins.metodaNoua()
24
25      // apelurile metodelor (din clasa de baza) care supraincarca acelasi nume
26      obiectExtins.metodaMostenitaSupraincarcata()
27      obiectExtins.metodaMostenitaSupraincarcata(1000)
28
29      // apelul metodei din subclasa (implementarea noua, rescrisa)
30      obiectExtins.metodaRescrisa()
31      // apelul metodei din clasa de baza (implementare de baza)
32      super.metodaRescrisa()
33    }
34  }
```

Implementarea unei interfete de tip stiva:

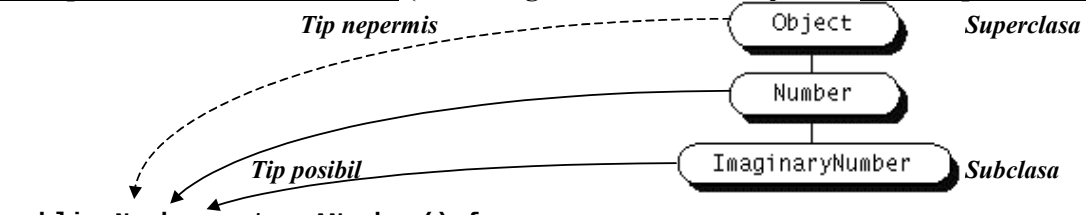
```
1  public interface StackInterface {
2    boolean empty();
3    void push( Object x);
4    Object pop() throws EmptyStackException;
5    Object peek() throws EmptyStackException;
6  }
```

```
1  public class Stack implements StackInterface {
2    private Vector v = new Vector(); // utilizeaza clasa java.util.Vector
3
4    public void push(Object item) { v.addElement(item); }
5    public Object pop() {
6      Object obj = peek();
7      v.removeElementAt(v.size() - 1);
8      return obj;
9    }
10   public Object peek() throws EmptyStackException {
11     if (v.size() == 0) throw new EmptyStackException();
12     return v.elementAt(v.size() - 1);
13   }
14   public boolean empty() { return v.size() == 0; }
15 }
```

```
1  public class Stack extends Vector implements StackInterface {
2    public Object push(Object item) { addElement(item); return item; }
3    public Object pop() {
4      Object obj;
5      int len = size();
6      obj = peek();
7      removeElementAt( len - 1);
8      return obj;
9    }
10   public Object peek() {
11     int len = size();
12     if (len == 0) throw new EmptyStackException();
13     return elementAt( len - 1);
14   }
15   public boolean empty() { return size() == 0;}
16 }
```

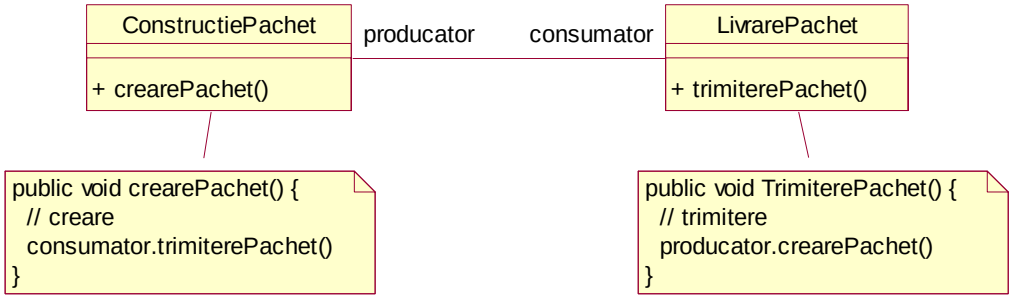
```
1  Vector v = new Stack();  // cod legal – referinta la clasa de baza
2                           // poate fi initializata cu obiect din subclasa
3  v.insertElementAt(x, 2); // cod legal – dar inserarea unui obiect in stiva
4                           // incalca principiul de functionare al stivei
```

Constrangeri ale tipurilor valorilor returnate (constrangeri similare exista pentru tipurile parametrilor)




```
public Number returnANumber() {
  //     . . .
}
```

Asocierile bidirectionale intre doua clase corespund situatiei in care **ambele clase au referinte una catre cealalta**. **Diagrama UML**:

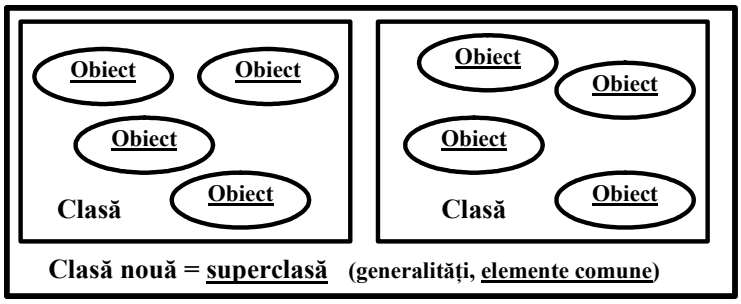


are drept corespondent **codul Java**:

```
public class ConstructiePachet {
  LivrarePachet consumator;

  public void crearePachet(  /* eventuali parametri */ ) {
    // creare
    consumator.trimiterePachet()
  }
}

public class LivrarePachet {
  ConstructiePachet producator;

  public void trimiterePachet( /* eventuali parametri */ ) {
    // trimitere
    producator.crearePachet()
  }
}
```

Generalizarea = **extragerea elementelor comune** (atribute, operaţii şi constrângeri) ale unui ansamblu de clase **într-o nouă clasă mai generală**, denumită **superclasă,**

- **superclasa** este **o abstracţie a subclaselor ei,**
- **arborii de clase** sunt **construiţi pornind de la frunze**
- **utilizată cand elementele modelului au fost identificate**, pentru a obţine o descriere detaşată a soluţiilor
- semnifică "**este un (fel de)**" sau "**este ca**", şi priveşte doar clasele, adică **nu este instanţiabilă**

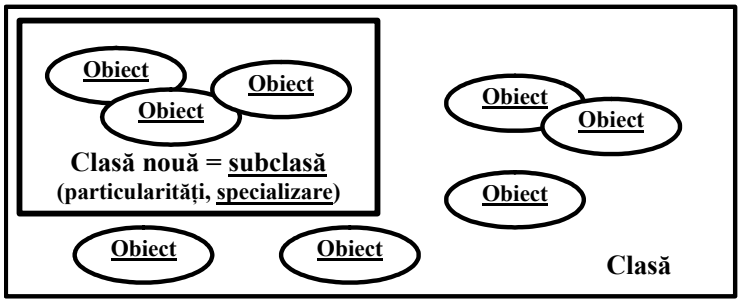


De fapt, generalizarea actioneaza in OO la doua niveluri:
- **clasele sunt generalizari ale ansamblurilor de obiecte** (un obiect este **de felul** specificat de o clasa),
- **superclasele sunt generalizari de clase** (obiectele **de felul** specificat in clasa sunt si **de felul** specificat in superclasa).

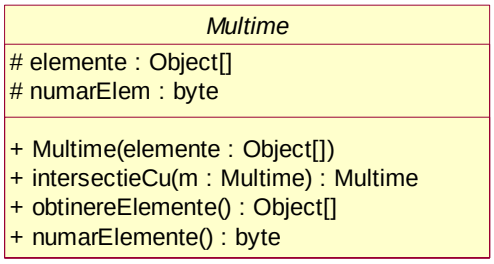
Limbajele OO ofera ambele mecanisme de generalizare. Limbajele care **ofera doar constructii numite obiecte** (si eventual clase) se pot numi **limbaje care lucreaza cu obiecte** (si eventual clase).

Specializarea = **capturarea particularităţilor** unui ansamblu de obiecte ale unei clase existente, noile caracteristici fiind reprezentate **într-o nouă clasă mai specializată**, denumită **subclasă,**

- utilă pentru **extinderea coerentă a unui ansamblu de clase**
- **bază a programării prin extindere şi a reutilizării**, noile cerinţe fiind încapsulate în subclase care extind în mod armonios (coerent) funcţiile existente

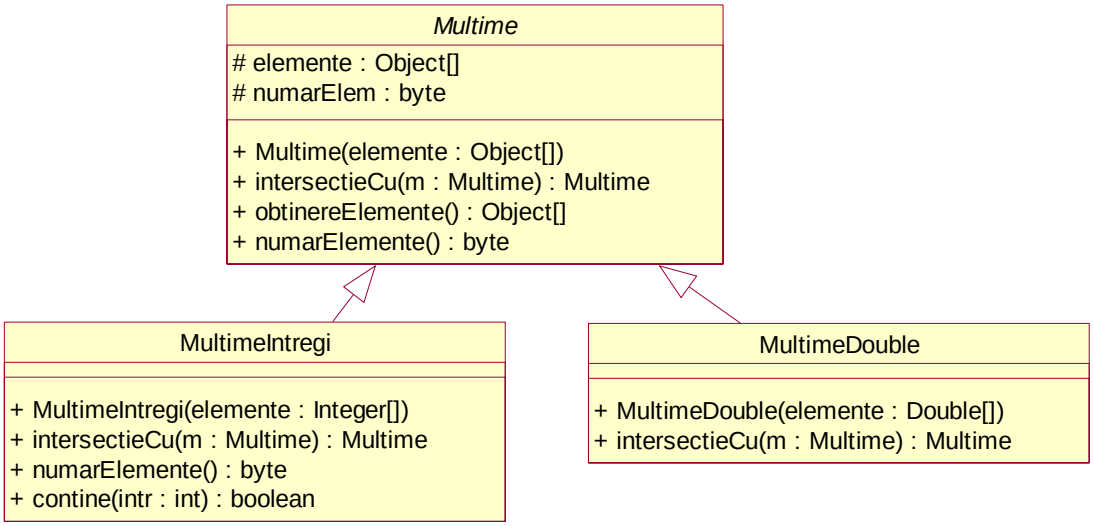


Moştenirea = **tehnică de generalizare** oferită de limbajele OO pentru a construi o clasă pornind de la una sau mai multe alte clase, **partajând atributele, operaţiile şi uneori constrângerile**, într-o ierarhie de clase.



Diagramei UML:
ii corespunde codul Java:

```
1  public abstract class Multime {                    // clasa declarata abstract
2    protected Object[] elemente;
3    protected byte numarElem;
4
5    public Multime(Object[] elemente) {       // parametru generic tip Object[]
6        this.elemente = elemente;             // acces la obiectul curent cu this
7        numarElem = (byte) elemente.length;   // conversie de tip de la int la byte
8    }
9
10   public abstract Multime intersectieCu(Multime m);  // metoda declarata abstract
11                                      // valoare returnata generica tip Multime
12
13   public Object[] obtinereElemente() { // valoare returnata generica tip Object[]
14       return elemente;
15   }
16
17   public byte numarElemente() {        // implementare de baza
18       return numarElem;
19   }
20 }
```

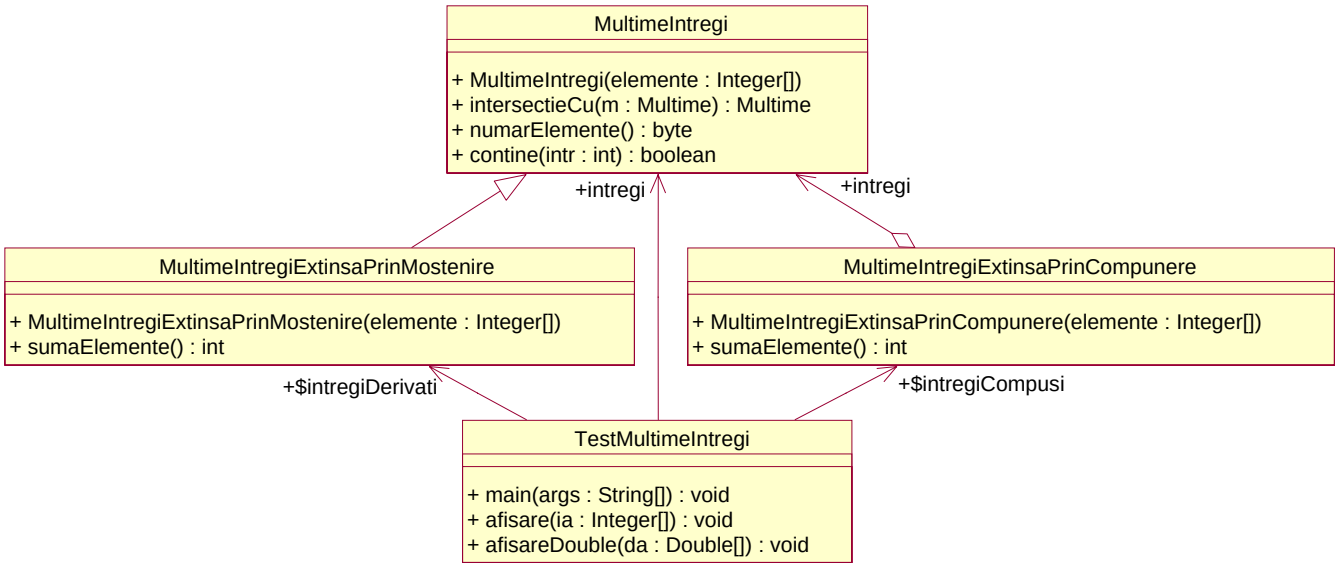


Diagramei UML:
ii corespunde codul Java:

```
1  public class MultimeIntregi extends Multime {
2
3    public MultimeIntregi(Integer[] elemente) { // parametru concret tip Integer[]
4        super(elemente);  // apelul constructorului clasei de baza Multime cu super
5    }
6
7    public final Multime intersectieCu(Multime m) { // implementarea metodei
8                                    // declarata abstract in clasa de baza
9        Multime mNoua;
10       Integer[] elementeIntersectie;
11       int nrElemente = 0;
12
13       for (int i=0; i< elemente.length; i++) {
14         for (int j=0; j< m.elemente.length; j++) {
15           if (elemente[i].equals(m.elemente[j])) {
16             nrElemente++;
17           }
18         }
```

```
19        }
20        int index = 0;
21        elementeIntersectie = new Integer[nrElemente];
22        for (int i=0; i< elemente.length; i++) {
23          for (int j=0; j< m.elemente.length; j++) {
24            if (elemente[i].equals(m.elemente[j])) {
25              elementeIntersectie[index++] = new Integer(elemente[i].toString());
26            }
27          }
28        }
29        mNoua = new MultimeIntregi(elementeIntersectie);
30
31        return mNoua;
32    }
33
34    public byte numarElemente() {          // reimplementare (rescriere cod)
35        return (byte) elemente.length;     // conversie de tip de la int la byte
36    }
37
38    public boolean contine(int intr) {     // metoda noua
39        for (int i=0; i< elemente.length; i++) {
40          Integer inte = (Integer) elemente[i];
41          if (inte.intValue() == intr) {
42            return true;
43          }
44        }
45        return false;
46    }
47  }
```

Diagramei UML:



ii corespund codurile Java:

- ale unei clase care extinde prin mostenire (extindere):

```
1   public class MultimeIntregiExtinsaPrinMostenire extends MultimeIntregi {
2
3     public MultimeIntregiExtinsaPrinMostenire(Integer[] elemente) {
4         super(elemente);
5     }
6
7     public int sumaElemente() {     // metoda noua
8         int suma = 0;
9         Integer[] ti = (Integer[]) elemente;   // utilizare atribut mostenit elemente
10        for (int i=0; i< ti.length; i++) {
11          suma = suma + ti[i].intValue();
12        }
13        return suma;
14    }
15  }
```

- ale unei clase care extinde prin delegare (compunere):

```
 1  public class MultimeIntregiExtinsaPrinCompunere {
 2    public MultimeIntregi intregi;                  // obiect componenta
 3
 4    public MultimeIntregiExtinsaPrinCompunere(Integer[] elemente) {
 5        intregi = new MultimeIntregi(elemente);
 6    }
 7
 8    public int sumaElemente() {     // metoda noua
 9        int suma = 0;
10        Integer[] ti = (Integer[]) intregi.obtinereElemente();
11        for (int i=0; i< ti.length; i++) {
12          suma = suma + ti[i].intValue();
13        }
14        return suma;
15    }
    }
```

- ale unei clase care permite testarea comportamentului (si compararea modului de utilizare) in cele doua cazuri:

```
 1  public class TestMultimeIntregi {
 2    public MultimeIntregi intregi;
 3    public static MultimeIntregiExtinsaPrinMostenire intregiDerivati;
 4    public static MultimeIntregiExtinsaPrinCompunere intregiCompusi;
 5
 6    public static void main(String[] args) {
 7      int i;
 8
 9      Integer[] tablouA = { new Integer(1), new Integer(3), new Integer(5) };
10      MultimeIntregi multimeA = new MultimeIntregi(tablouA);
11
12      intregiCompusi = new MultimeIntregiExtinsaPrinCompunere(tablouA);
13      int suma = intregiCompusi.sumaElemente();
14      System.out.println("Suma elementelor " + suma);
15
16      intregiDerivati = new MultimeIntregiExtinsaPrinMostenire(tablouA);
17      suma = intregiDerivati.sumaElemente();
18      System.out.println("Suma elementelor " + suma);
19    }
20  }
```

Subclasele (clasele care extind prin mostenire) pot sa:

- mareasca gradul de **detaliere al obiectelor**:
  - **adaugand noi atribute**, inexistente in clasa de baza (**detaliere mai mare a starilor** obiectelor in subclasa)
  - **adaugand noi metode**, inexistente in clasa de baza (**detaliere mai mare a comportamentului** in subclasa)
- mareasca gradul de **concretete a obiectelor**:
  - **implementand eventualele metode abstracte** din clasa de baza (**mai multa concretete a comportamentului**)
- introduca **diferentieri ale obiectelor**:
  - **redeclarand unele atribute existente** in clasa de baza (**ascunderea** – *hiding* - atributelor cu acelasi nume), adica introducerea unei **diferentieri a starii obiectelor din subclasa** fata de cele din clasa de baza,
  - **reimplementand unele metode existente** in clasa de baza (**rescrierea** – *overriding* - metodelor cu acelasi nume) adica introducerea unei **diferentieri a comportamentului obiectelor din subclasa** fata de clasa de baza

Subclasele mostenesc toate:

- **atributele din clasa de baza care nu sunt ascunse** prin redeclarare,
- **metodele din clasa de baza care nu sunt rescrise** prin reimplementare.

Obiectele din subclasa au toate atributele declarate in clasa de baza, si **pot utiliza toate metodele declarate in clasa de baza**. Aceasta **reutilizare a codului clasei de baza de catre subclase** este principalul beneficiu al mostenirii.

Subclasele nu mostenesc

- **constructorii** clasei de baza (**au constructori proprii**), dar **pot face apel la constructorii clasei de baza** (daca se intampla acest lucru, apelul la constructorul clasei de baza, realizat prin apelul **`super()`**, trebuie sa fie prima declaratie din corpul constructorului subclasei),
- **atributele cu caracter global** (static),
- **metodele cu caracter global** (static).

Membrii globali (statici) **ai clasei de baza nu pot fi mosteniti** de obiectele din subclasa (**tin strict de clasa in care au fost declarati**).

Descrierea unui caz de utilizare trebuie sa cuprinda urmatoarele elemente:

1. Numele cazului de utilizare (cat mai sugestiv, pentru a sintetiza cazul de utilizare).

Ex. ***Autentificarea utilizatorului***

2. Scurta descriere a cazului de utilizare.

Ex. ***Utilizatorul*** *ii* ***transmite sistemului*** *datele necesare autentificarii, si in caz de succes* ***capata drept de acces*** *la* ***sistem***.

3. Actori (entitati exterioare sistemului modelat, implicate in cazul de utilizare).

Ex. ***Utilizatorul***

4. Preconditii (conditiile necesare pentru declansarea cazului de utilizare).

Ex. ***Sistemul*** *e aflat in executie, cunoaste datele necesare autentificarii* ***Utilizatorului*** *si ii ofera* ***Utilizatorului*** *o modalitate de a-i transmite aceste date.*

5. Evenimentul care declanseaza cazul de utilizare (modul in care incepe cazul de utilizare)**.**

Ex. ***Utilizatorul cere*** *autentificarea de catre* ***sistem***.

6. Descriere a interactiunii dintre actori si fiecare caz de utilizare (scenariul principal)**.**

Ex. **I.** ***Utilizatorul transmite*** *datele necesare autentificarii de catre sistem.*

II. ***Sistemul verifica*** *autenticitatea datelor primite de la* ***Utilizator***.

III. ***Sistemul*** *prezinta un* ***mesaj de confirmare*** *catre* ***Utilizator***.

IV. ***Utilizatorul capata acces*** *la* ***sistem***.

7. Alternative la cazul de utilizare principal (scenariile alternative si situatiile exceptionale)**.**

Ex. **1.** *Daca* ***sistemul nu este lansat****, autentificarea* ***nu poate avea loc***.

2. *Daca* ***sistemul nu cunoaste*** *datele de autentificare, autentificarea* ***esueaza***.

3. *Daca* ***utilizatorul transmite in mod eronat*** *datele de autentificare, autentificarea* ***esueaza***.

8. Evenimentul care produce oprirea cazului de utilizare (modul in care se incheie cazul)**.**

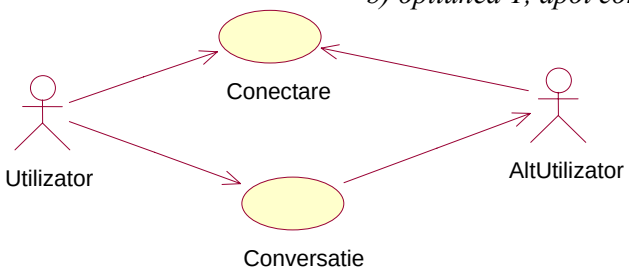
Ex. ***Utilizatorului*** *i se prezinta un* ***mesaj de confirmare***.

9. Postconditii (efectele incheierii cazului de utilizare).

Ex. ***Utilizatorul are acces*** *la* ***sistem***.

Este important sa fie clar prezentate:

- **schimburile de informatii** (parametrii interactiunilor) Ex. ***datele necesare autentificarii, mesaj de confirmare***
- **originea informatiilor** si **ordinea schimbarii** lor Ex. ***Utilizatorul*** *transmite datele necesare autentificarii*
- **repetitiile comportamentului**, prin **constructii pseudocod**
- **situatiile optionale** Ex. ***Utilizatorul*** *alege unul dintre* *a) optiunea X,*
  b) optiunea Y, apoi continua cu …



Caz de utilizare - Conectare

1. Nume (cat mai sugestiv, pentru a sintetiza cazul de utilizare).

Ex. ***Conectare***

2. Scurta descriere.

Ex. ***Clientul*** *se* ***conecteaza*** *la* ***server****, care* ***accepta*** *conexiunea si se pregateste pentru tratarea* ***clientului***.

3. Actori (entitati exterioare sistemului modelat, implicate in cazul de utilizare).

Ex. ***Utilizator***.

4. Preconditii (conditiile necesare pentru declansarea cazului de utilizare).

Ex. ***Serverul*** *e aflat in executie.* ***Clientul*** *cunoaste adresa si numarul de port pe care asculta* ***serverul***.

5. Evenimentul care declanseaza cazul de utilizare.

Ex. ***Utilizatorul lanseaza*** *componenta* ***client*** *a sistemului.*

6. Descriere a interactiunii dintre actori si fiecare caz de utilizare.

Ex. **I.** ***Utilizatorul lanseaza*** *componenta* ***client*** *a sistemului, care se* ***conecteaza*** *la* ***serverul*** *cunoscut.* ***Serverul accepta*** *conexiunea si* ***se pregateste*** *pentru tratarea clientului.*

II. ***Clientul ofera*** *o interfata grafica* ***utilizatorului***

7. Alternative la cazul de utilizare principal.

Ex. *Daca* ***serverul nu este lansat****, conectarea* ***esueaza****, iar* ***clientul anunta*** *acest lucru* ***utilizatorului***.

8. Evenimentul care produce oprirea cazului de utilizare.

Ex. ***Clientul*** *prezinta* ***utilizatorului*** *o interfata grafica.*

9. Postconditii (efectele incheierii cazului de utilizare)

Ex. *Sistemul este pregatit pentru ca* ***utilizatorul*** *sa poata conversa cu alti utilizatori prin interfeta grafica*

Caz de utilizare - Conversatie

1. Nume

Conversatie

2. Scurta descriere.

Clientii *preiau mesaje de la* ***utilizatori,*** *le* ***trimit*** *catre* ***server****, care le* ***difuzeaza*** *catre toti* ***clientii****, iar acestia le* ***prezinta*** *catre* ***utilizatori****.*

3. Actori

Utilizatori

4. Preconditii

Cazul de utilizare ***Conversatie*** *s-a incheiat cu succes.*

5. Evenimentul care declanseaza cazul de utilizare.

Un ***utilizator*** *scrie un mesaj in interfata grafica a* ***clientului*** *sau.*

6. Descriere a interactiunii dintre actori si cazul de utilizare (Transmisie si receptie).

I. ***Utilizatorul scrie*** *mesajul in interfata grafica a* ***clientului*** *sau,* ***clientul trimite*** *mesajul la* ***server****.*

II. ***Clientul*** *primeste inapoi mesajul* ***difuzat*** *de* ***server****, si* ***prezinta*** *mesajul in interfata grafica a* ***utilizatorului***

7. Alternativa la cazul de utilizare principal (Doar receptie).

Clientul *primeste mesajul trims de* ***alt utilizator****, mesaj* ***difuzat*** *de* ***server****, si* ***prezinta*** *mesajul in interfata grafica a* ***utilizatorului*** *sau.*

8. Evenimentul care produce oprirea cazului de utilizare.

Clientul *a prezentat* ***utilizatorului*** *mesajul primit.*

9. Postconditii

Sistemul este pregatit pentru ca ***utilizatorul*** *sa poata* ***continua conversatia*** *cu alti utilizatori prin intermediul interfetei grafice, sau pentru a* ***incheia conversatia*** *prin inchiderea interfetei grafice.*

Diagrama de secventa (MSC) si diagrama de comunicatie echivalenta – faza de analiza, nivel sistem

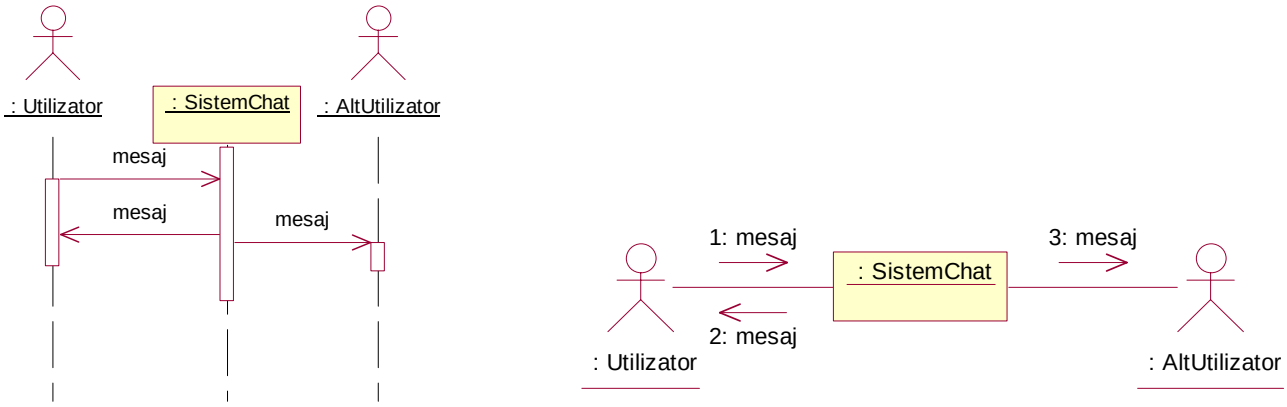


Diagrama de secventa (MSC) – faza de analiza, nivel subsisteme

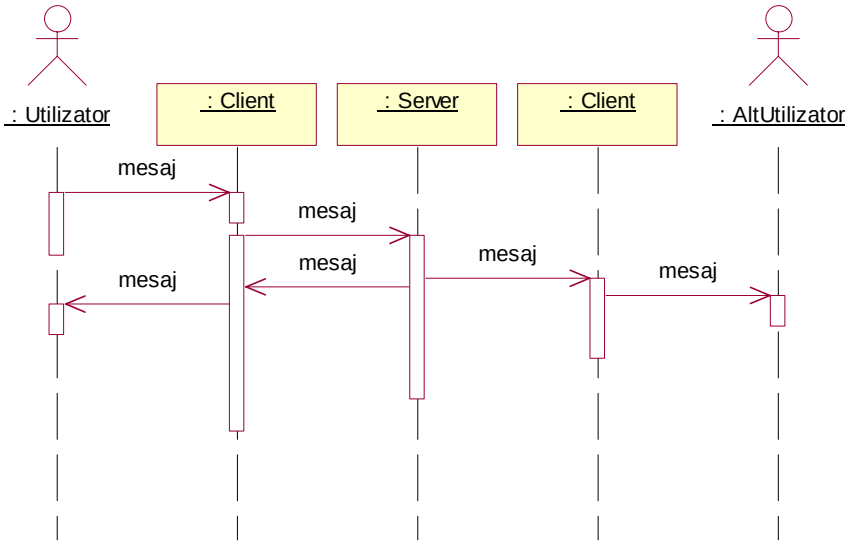


Diagrama de secventa (MSC) – <u>faza de analiza, nivel subsisteme, detaliere suplimentara</u>

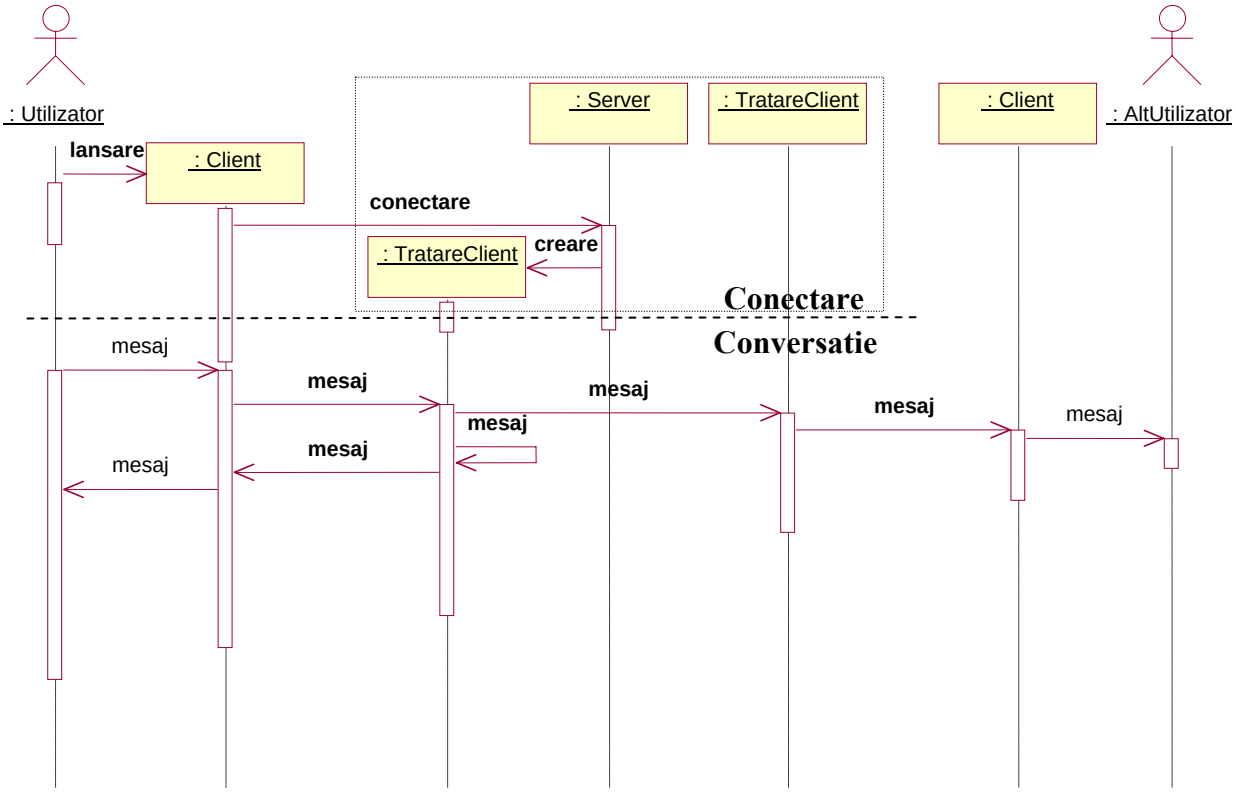


Diagrama de secventa (MSC) – <u>faza de proiectare, nivel subsisteme, detaliere suplimentara</u>

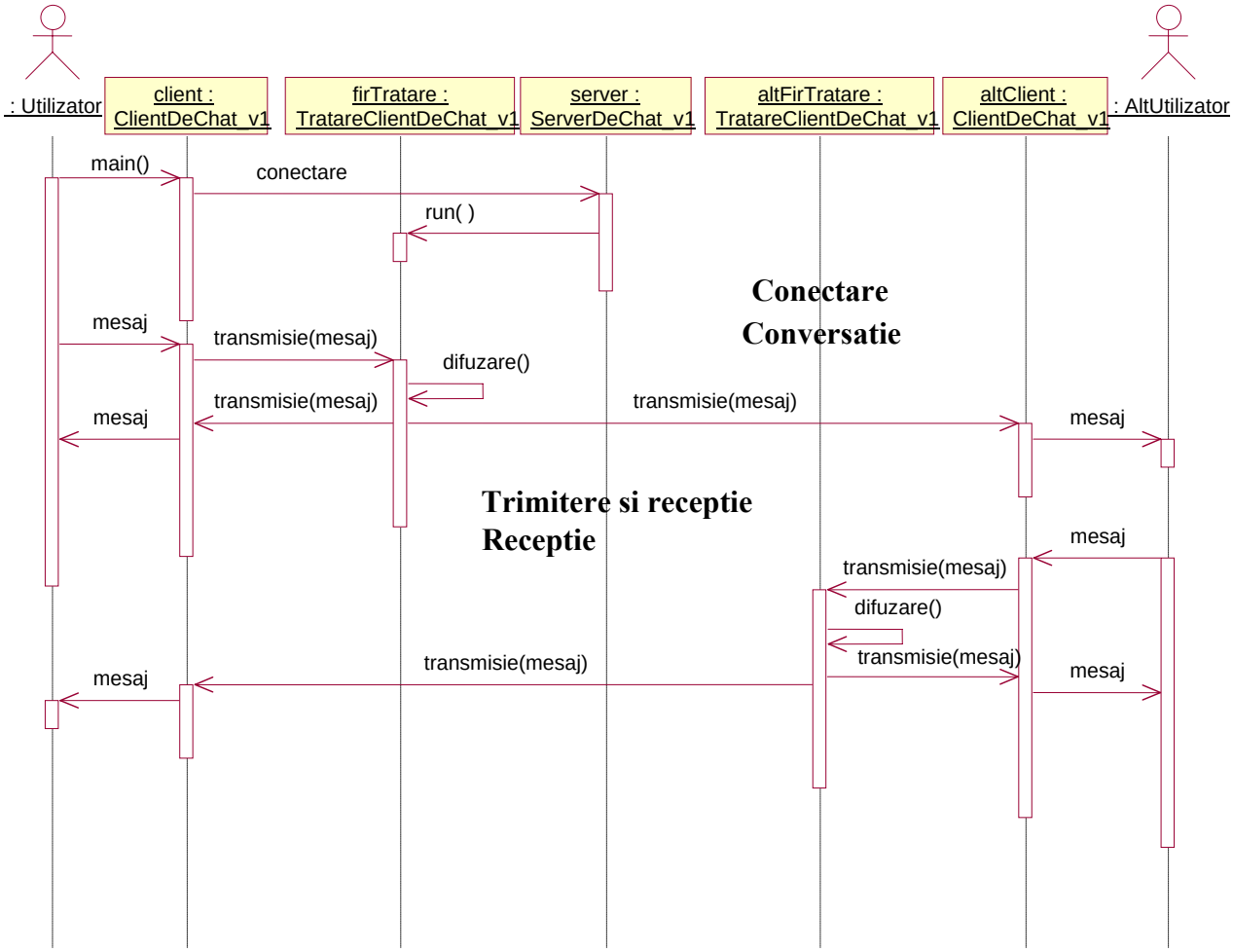


Diagrama de secventa (MSC) – faza de proiectare, subsistemul client

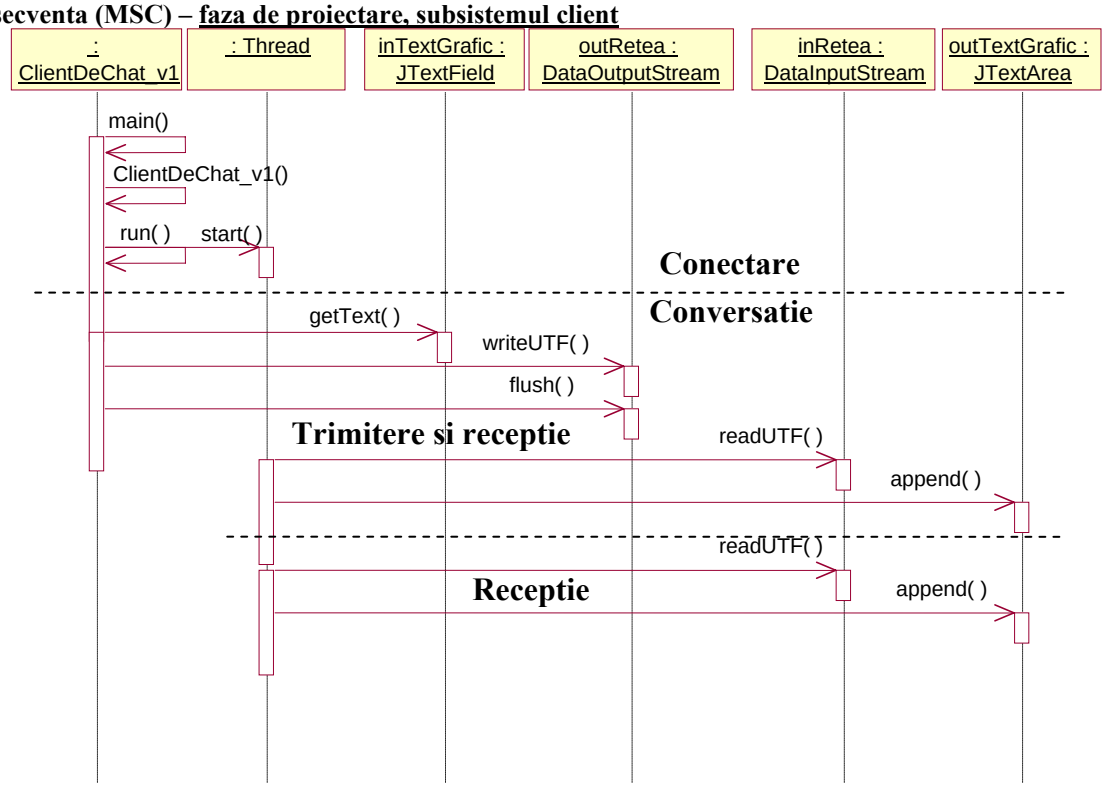


Diagrama de secventa (MSC) – faza de proiectare, subsistemul client

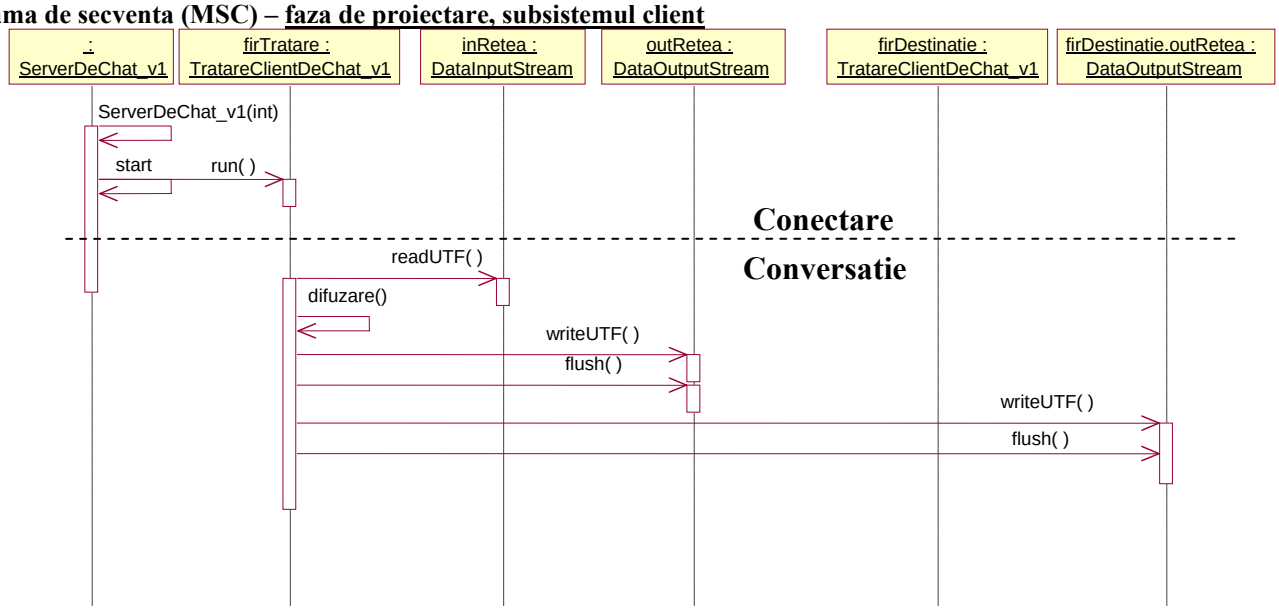


Diagrama de clase – subsistemul client

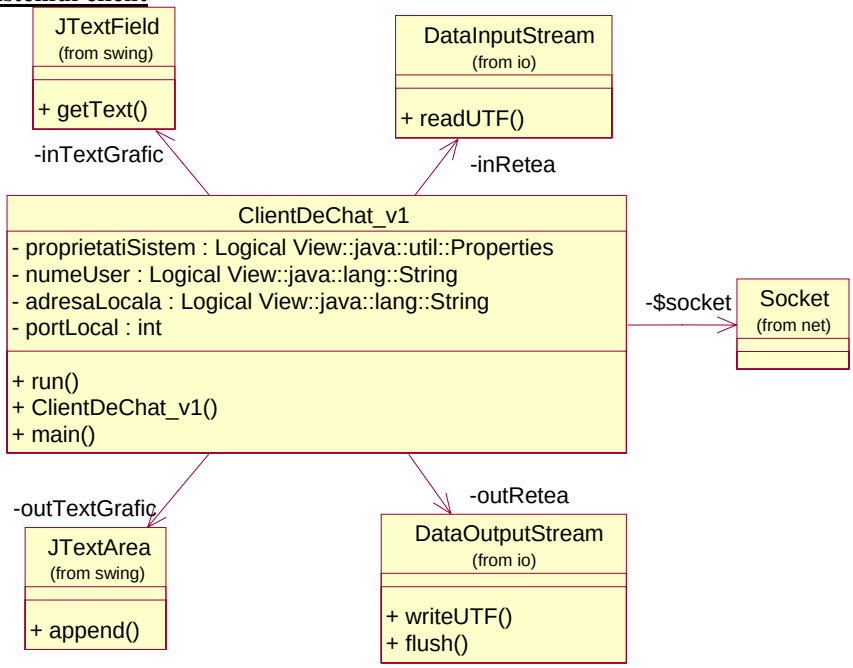


Diagrama de clase – subsistemul client

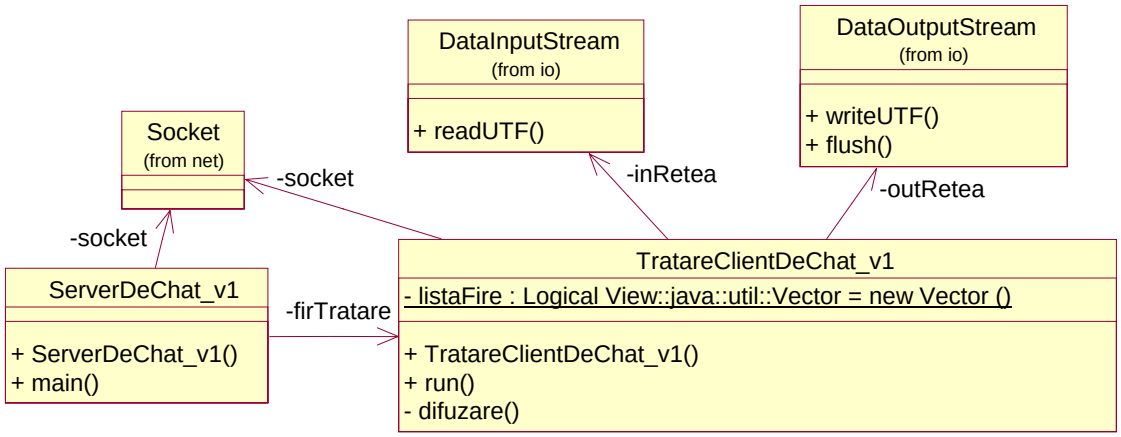
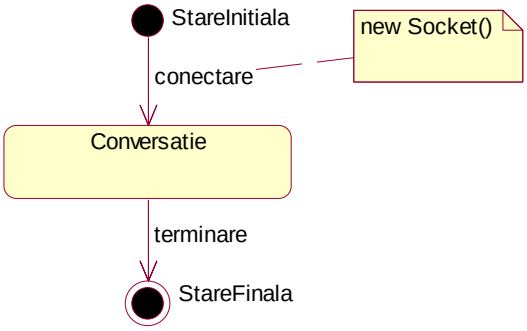
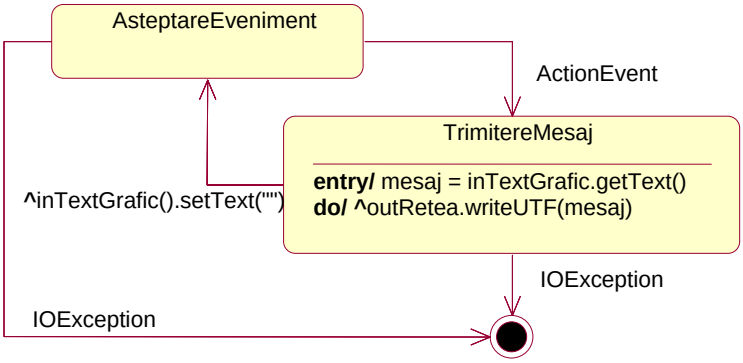


Diagrama masinii de stari a clientului:



1. Firul de executie care **trateaza evenimentele din interfata grafica**:



2. Firul de executie care **trateaza receptia mesajelor de la server**:

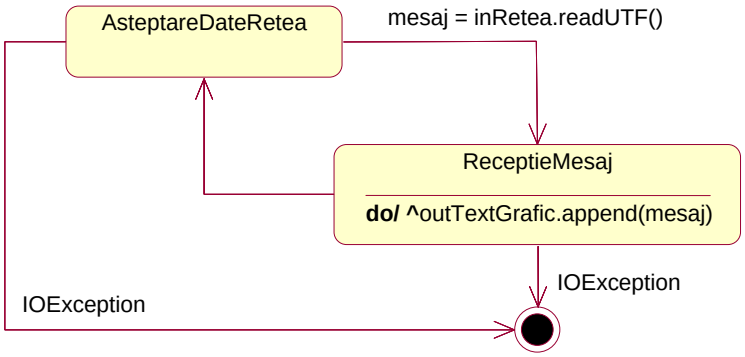


Diagrama masinii de stari (FSM) a serverului, **firul de executie principal (acceptor de conexiuni)**

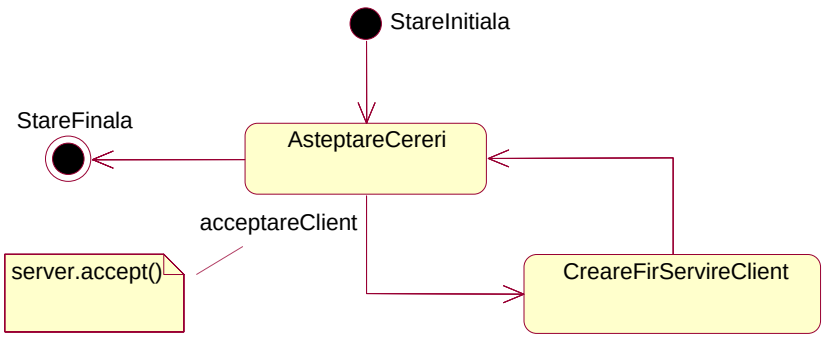


Diagrama masinii de stari (FSM) a serverului, **firul de tratare a clientilor**:

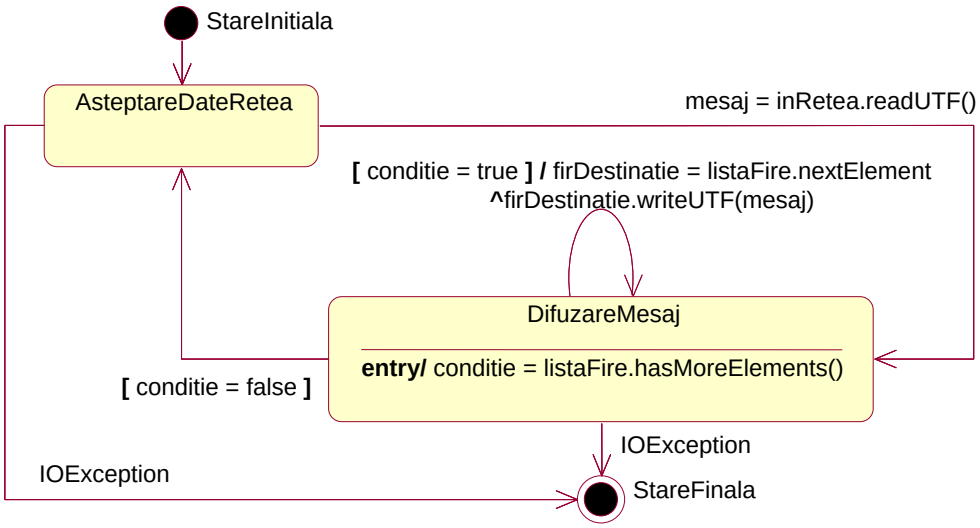
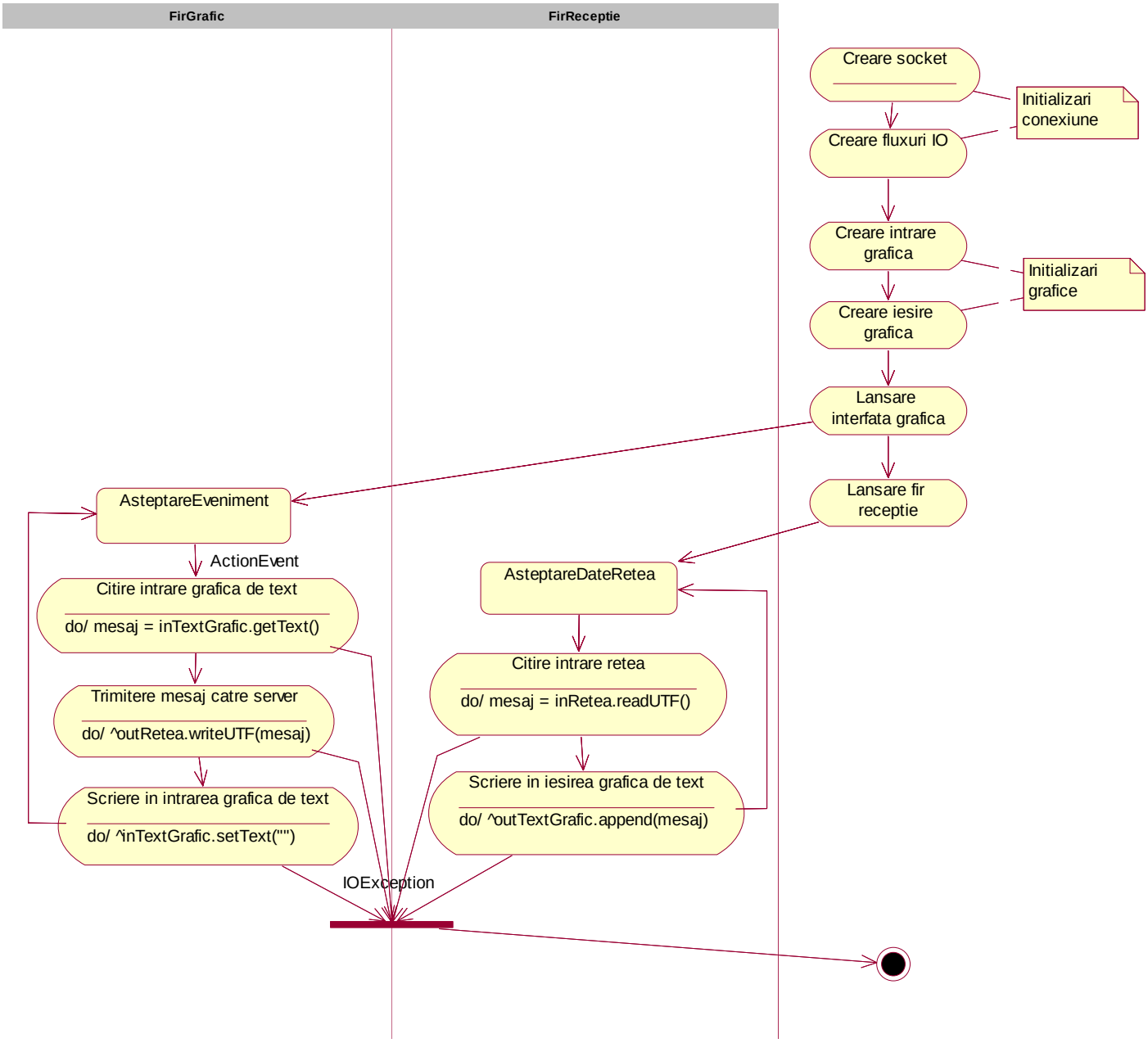


Diagrama de activitati (organigrama) a clientului, cu **firele de executie pe "coridoare de activitate" separate**:



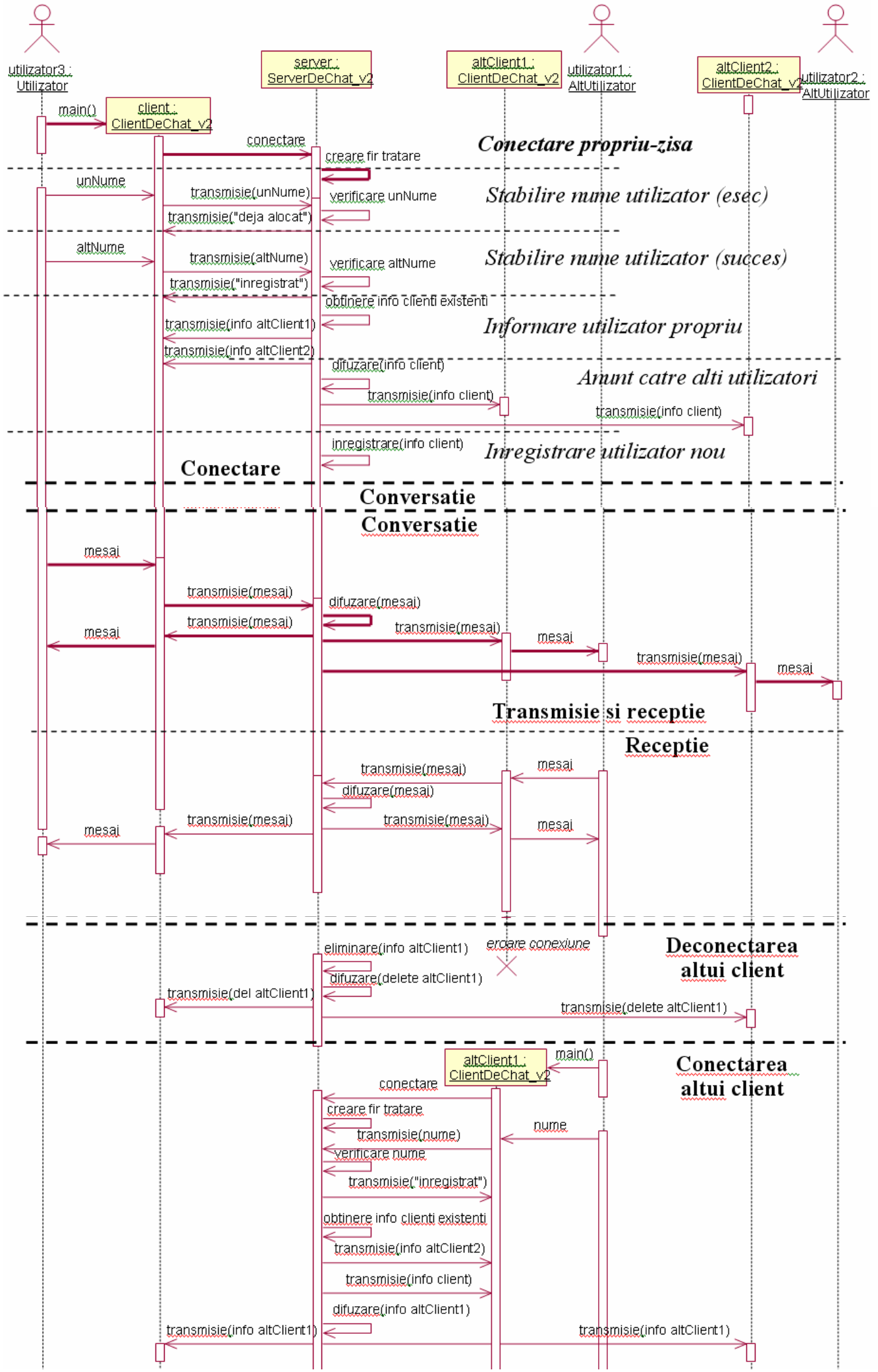

utilizator3 : Utilizator
server : ServerDeChat_v2
altClient1 : ClientDeChat_v2
utilizator1 : AltUtilizator
altClient2 : ClientDeChat_v2
utilizator2 : AltUtilizator
main()
client : ClientDeChat_v2
conectare
creare fir tratare
Conectare propriu-zisa
unNume
transmisie(unNume)
verificare unNume
Stabilire nume utilizator (esec)
transmisie("deja alocat")
altNume
transmisie(altNume)
verificare altNume
Stabilire nume utilizator (succes)
transmisie("inregistrat")
obtinere info clienti existenti
transmisie(info altClient1)
Informare utilizator propriu
transmisie(info altClient2)
difuzare(info client)
Anunt catre alti utilizatori
transmisie(info client)
transmisie(info client)
inregistrare(info client)
Inregistrare utilizator nou
Conectare
Conversatie
Conversatie
mesaj
transmisie(mesaj)
difuzare(mesaj)
transmisie(mesaj)
mesaj
transmisie(mesaj)
mesaj
transmisie(mesaj)
mesaj
Transmisie si receptie
Receptie
transmisie(mesaj)
mesaj
difuzare(mesaj)
mesaj
transmisie(mesaj)
transmisie(mesaj)
mesaj
eroare conexiune
eliminare(info altClient1)
Deconectarea altui client
difuzare(delete altClient1)
transmisie(del altClient1)
transmisie(delete altClient1)
altClient1 : ClientDeChat_v2
main()
Conectarea altui client
conectare
creare fir tratare
nume
transmisie(nume)
verificare nume
transmisie("inregistrat")
obtinere info clienti existenti
transmisie(info altClient2)
transmisie(info client)
difuzare(info altClient1)
transmisie(info altClient1)
transmisie(info altClient1)

Diagrama de clase actualizata a clientului:

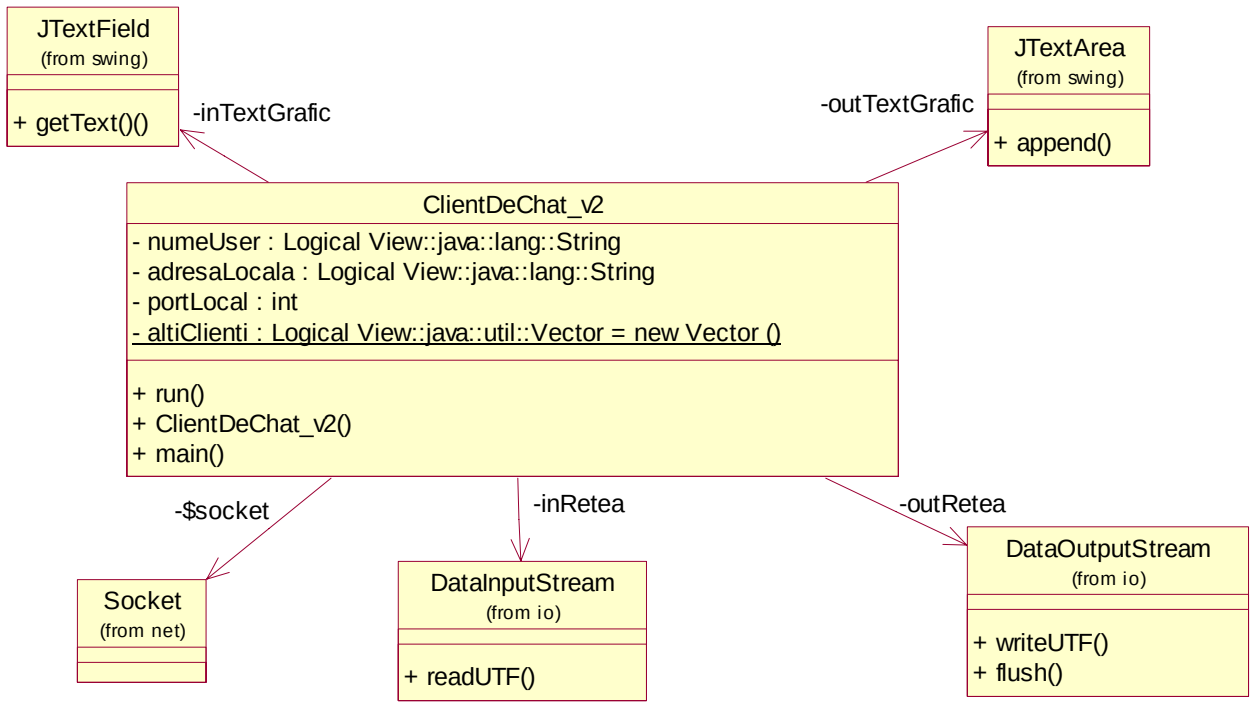
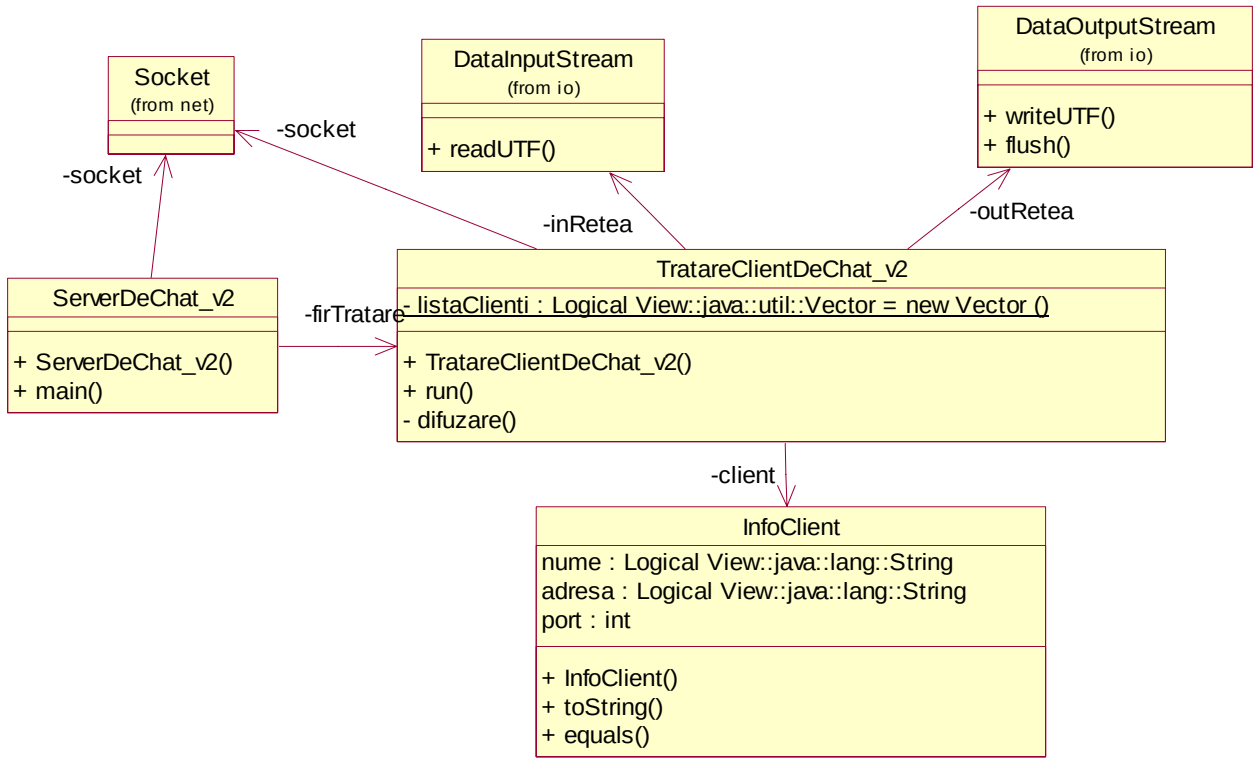
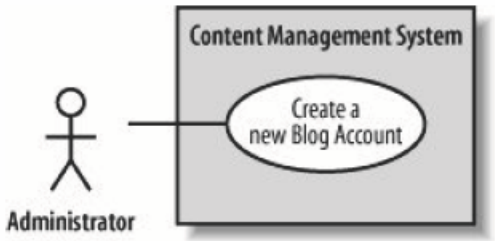
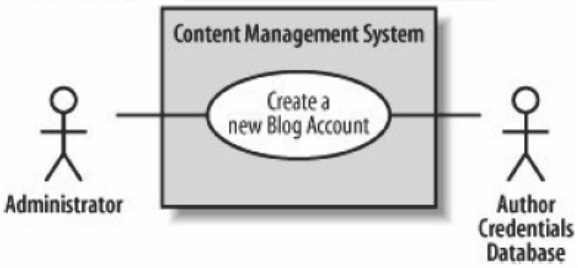
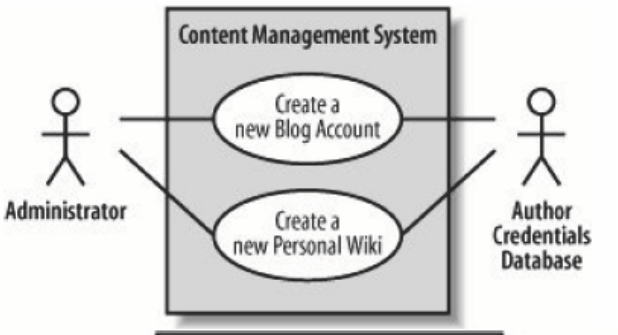


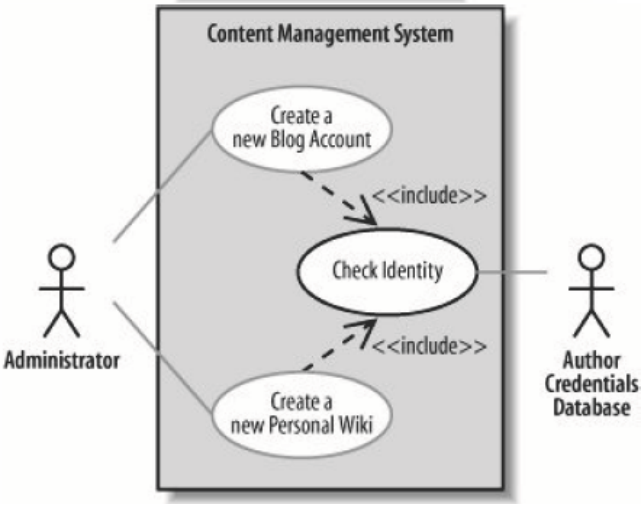
Diagrama de clase actualizata a serverului:

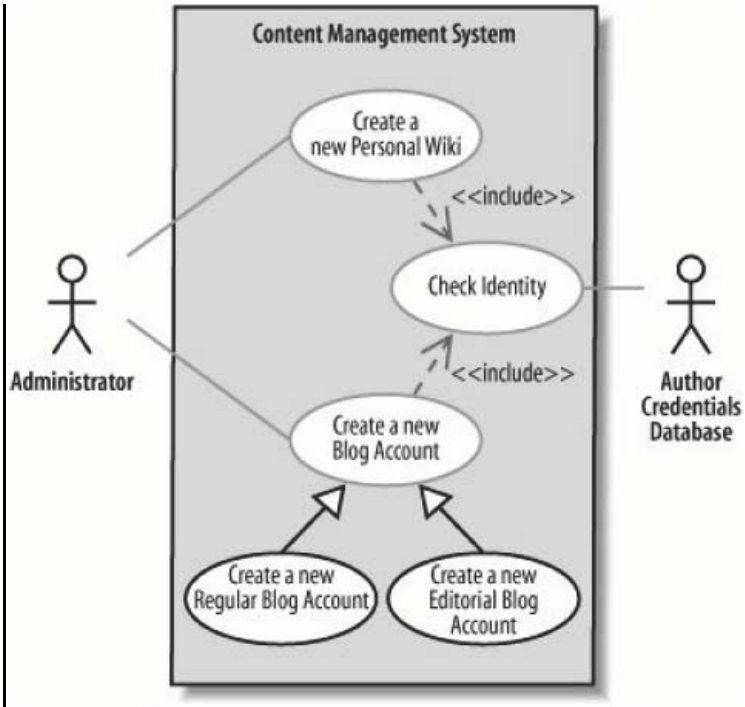


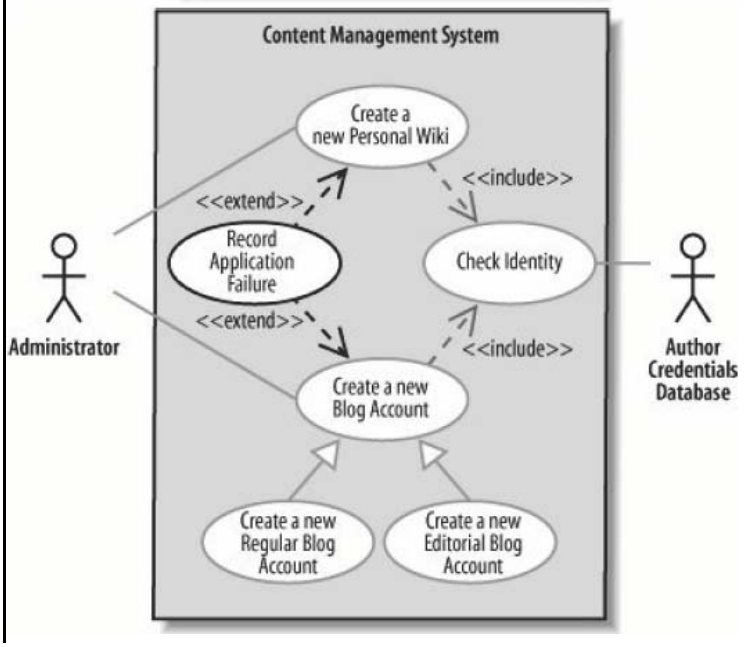

Content Management System
Create a new Blog Account
Administrator

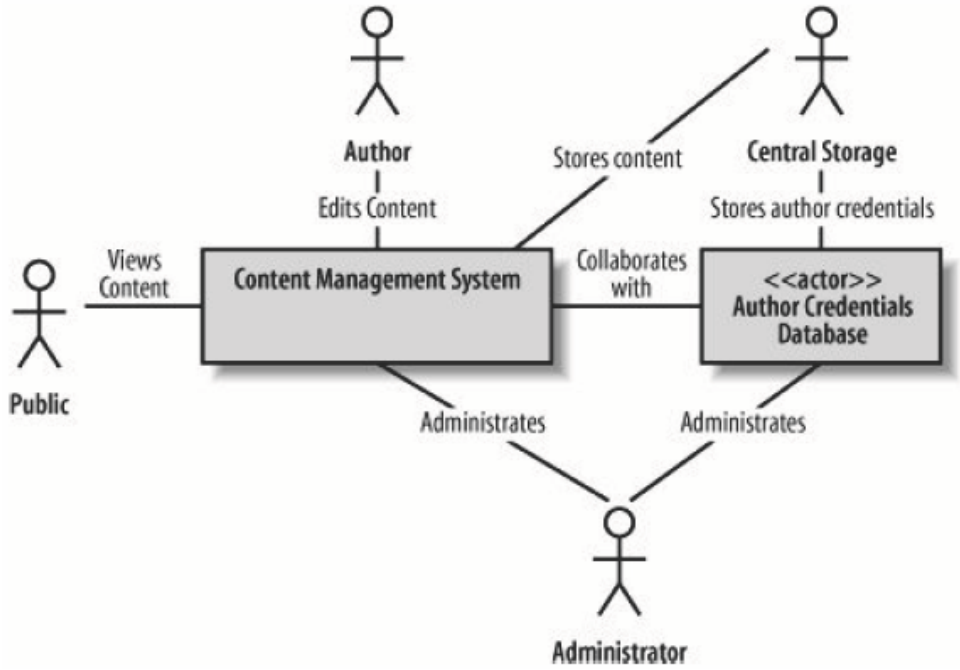

Content Management System
Create a new Blog Account
Administrator
Author Credentials Database

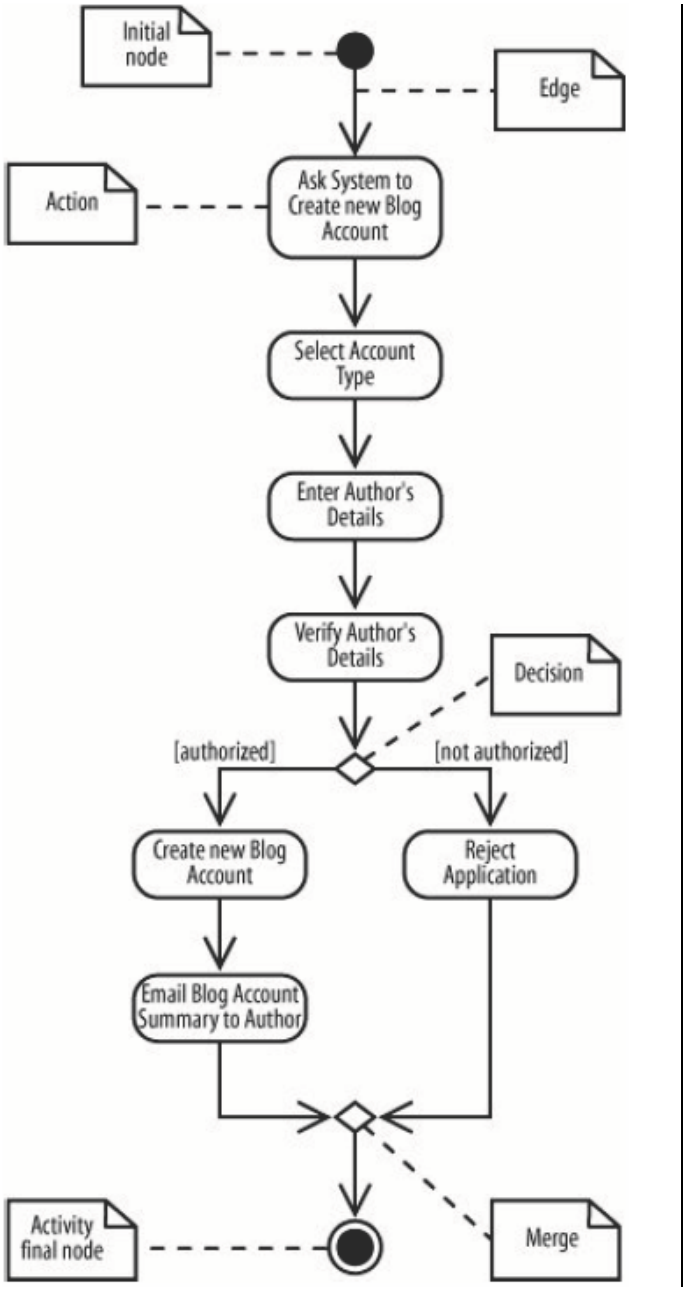

Content Management System
Create a new Blog Account
Create a new Personal Wiki
Administrator
Author Credentials Database

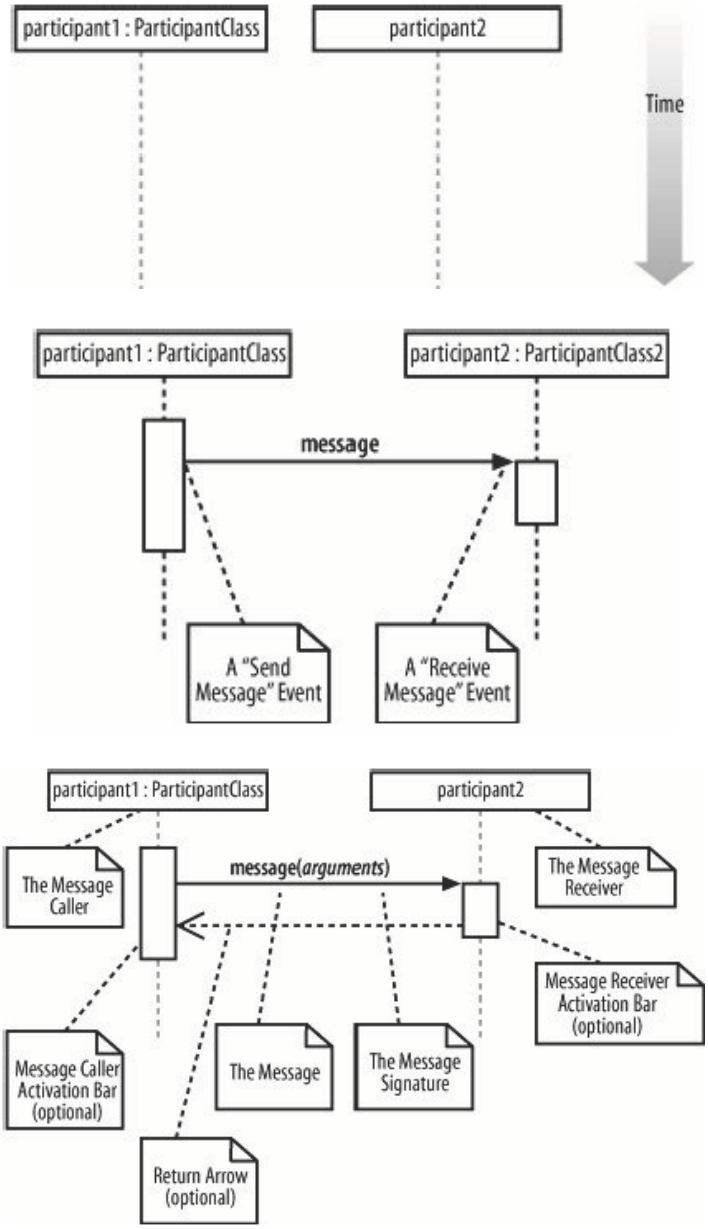

Content Management System
Create a new Blog Account
<<include>>
Check Identity
<<include>>
Create a new Personal Wiki
Administrator
Author Credentials Database

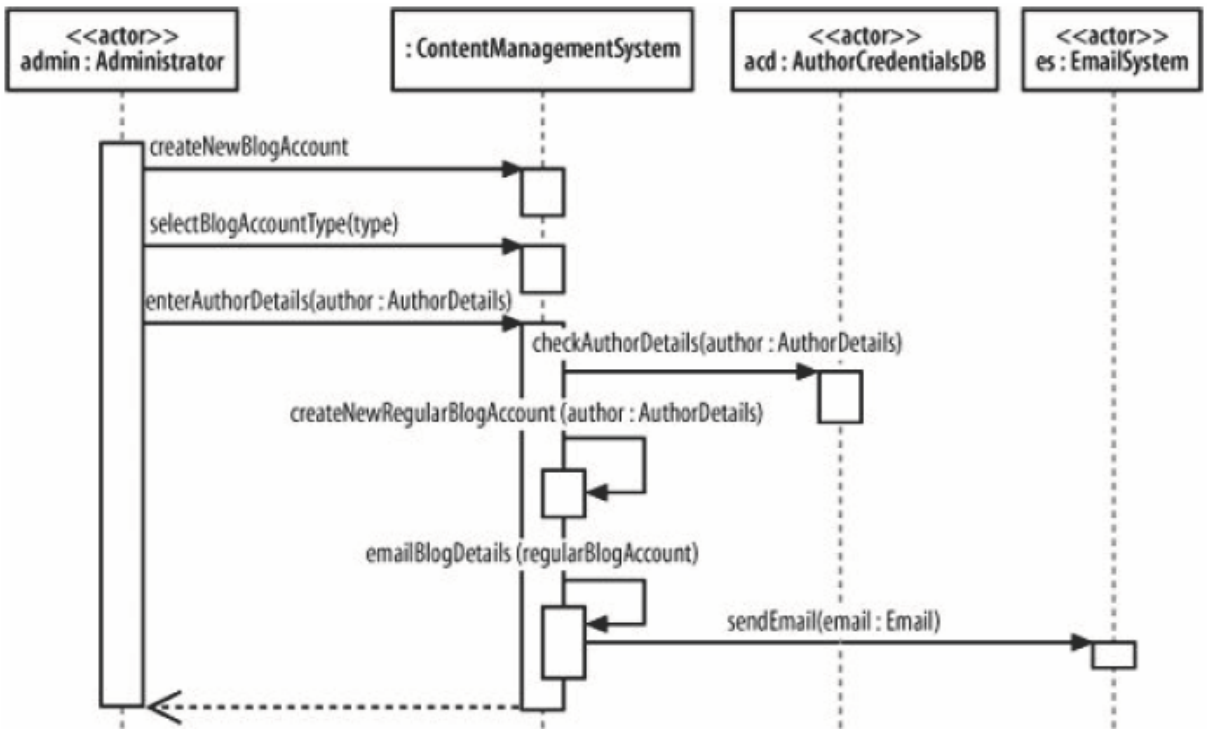

Content Management System
Create a new Personal Wiki
<<include>>
Check Identity
<<include>>
Create a new Blog Account
Create a new Regular Blog Account
Create a new Editorial Blog Account
Administrator
Author Credentials Database

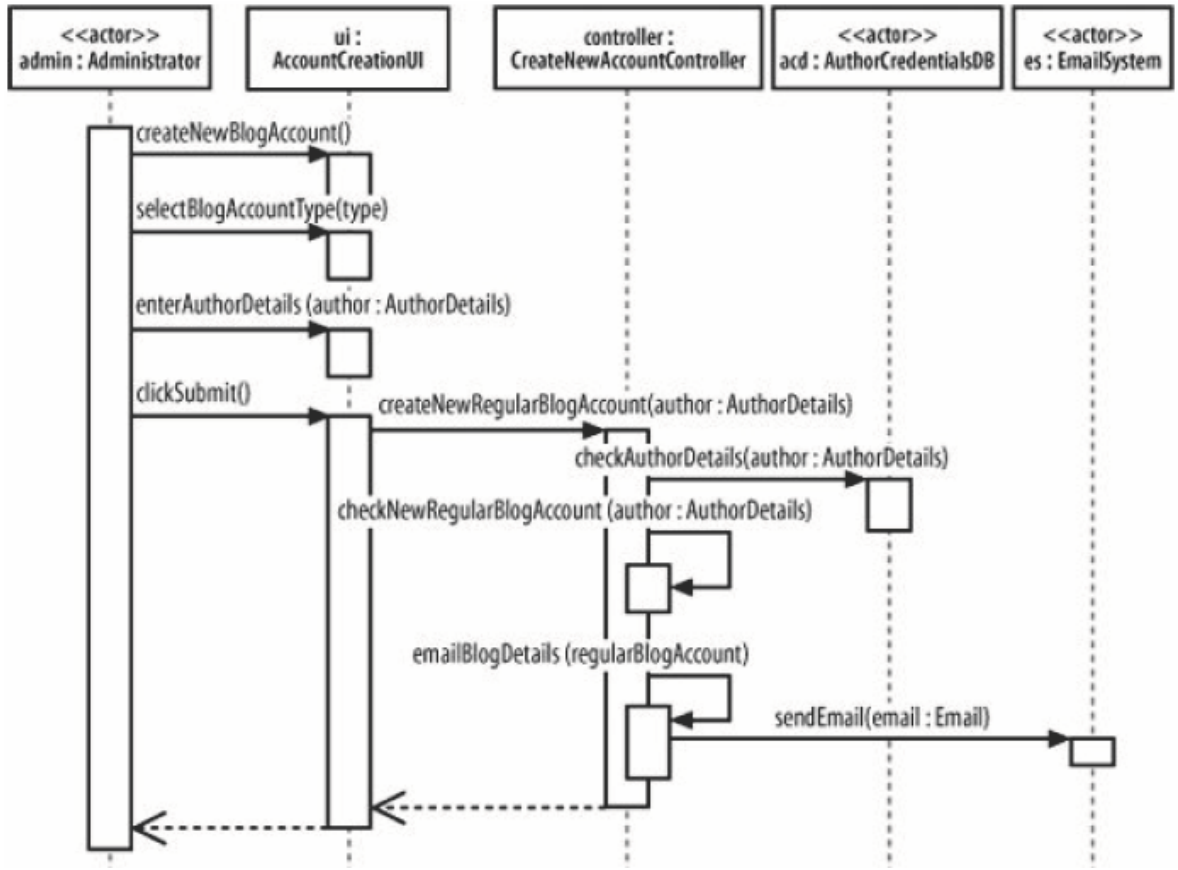

Content Management System
Create a new Personal Wiki
<<include>>
<<extend>>
Record Application Failure
Check Identity
<<extend>>
<<include>>
Create a new Blog Account
Create a new Regular Blog Account
Create a new Editorial Blog Account
Administrator
Author Credentials Database

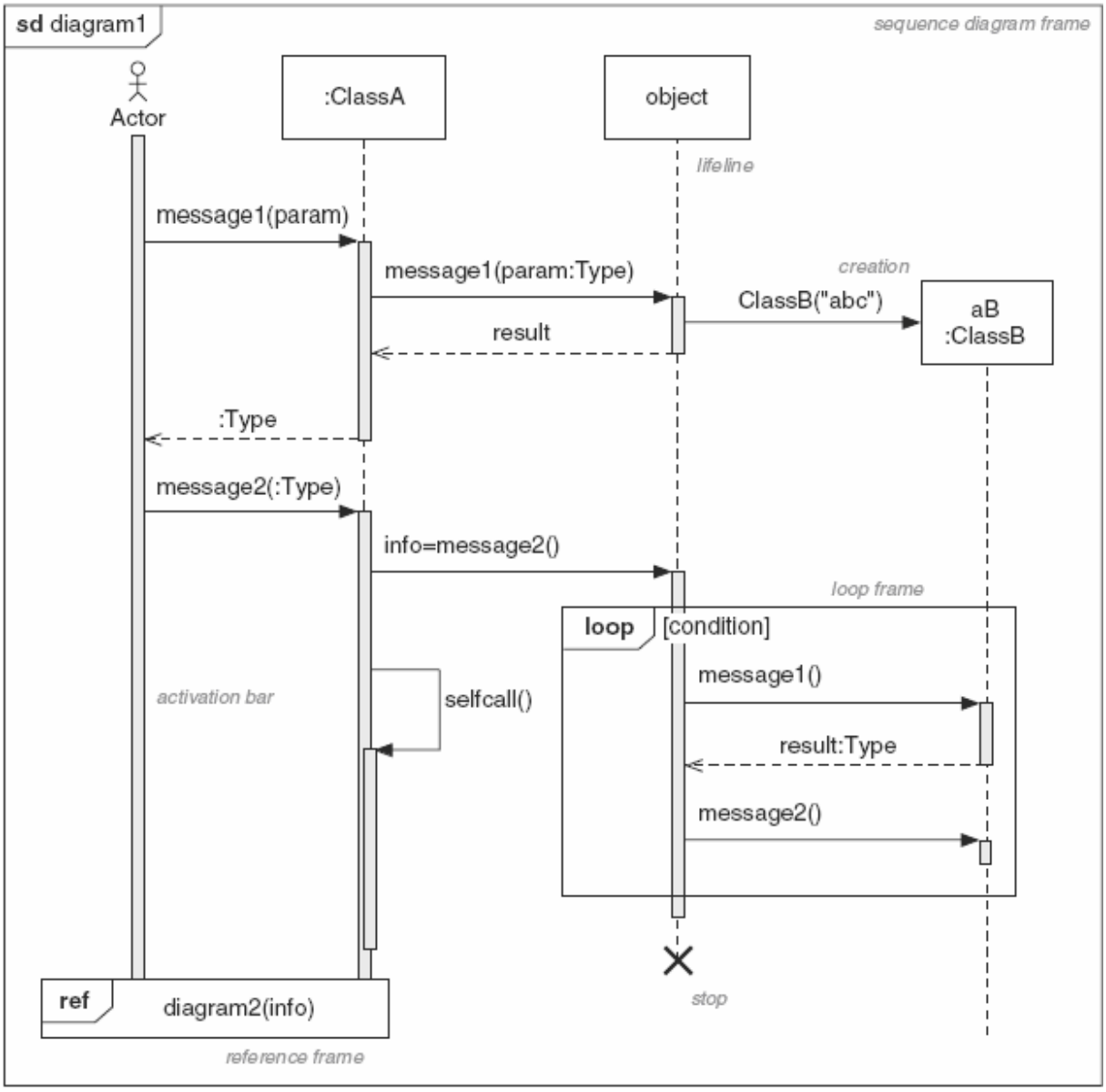

Author
Edits Content
Stores content
Central Storage
Stores author credentials
Views Content
Content Management System
Collaborates with
<<actor>> Author Credentials Database
Public
Administrates
Administrates
Administrator

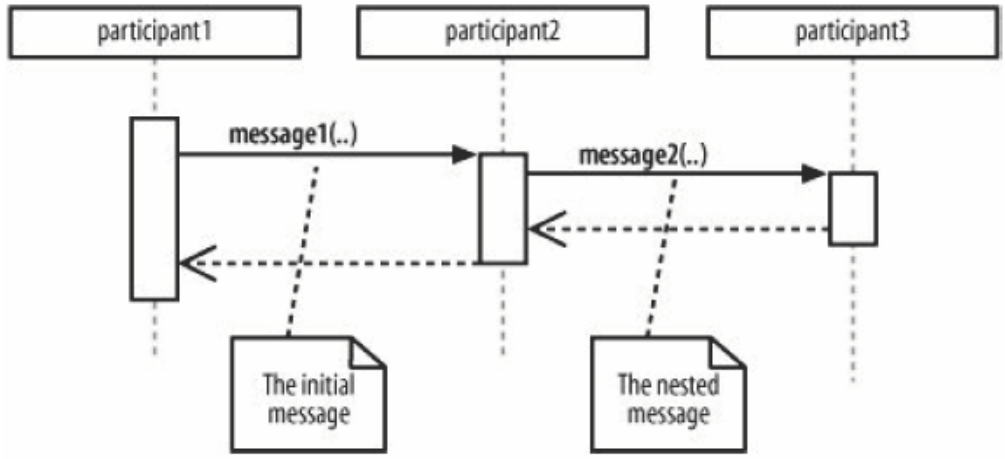

Initial node
Edge
Action
Ask System to Create new Blog Account
Select Account Type
Enter Author's Details
Verify Author's Details
Decision
[authorized]
[not authorized]
Create new Blog Account
Reject Application
Email Blog Account Summary to Author
Activity final node
Merge

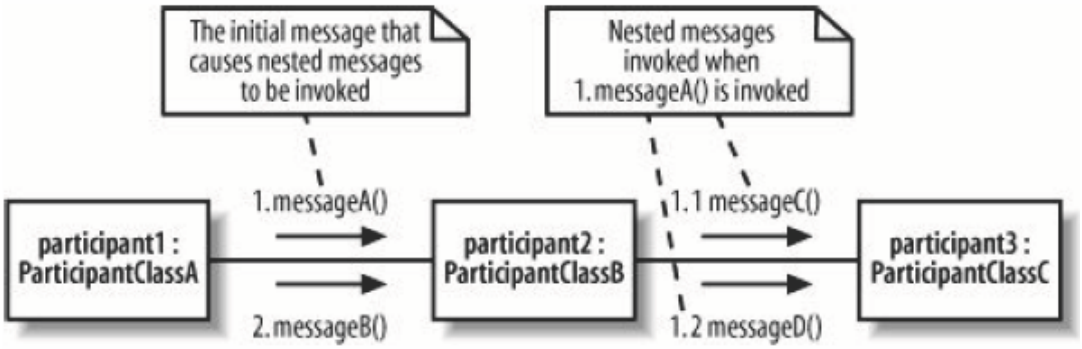

participant1 : ParticipantClass
participant2
Time
participant1 : ParticipantClass
participant2 : ParticipantClass2
message
A "Send Message" Event
A "Receive Message" Event
participant1 : ParticipantClass
participant2
The Message Caller
message(arguments)
The Message Receiver
Message Receiver Activation Bar (optional)
Message Caller Activation Bar (optional)
The Message
The Message Signature
Return Arrow (optional)

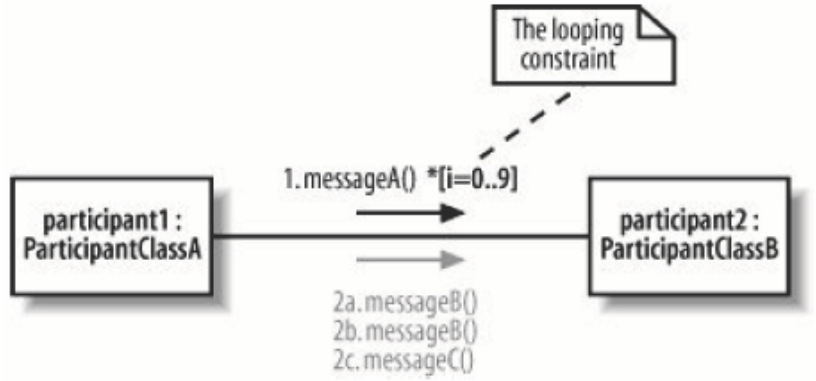

<<actor>>
admin : Administrator
: ContentManagementSystem
<<actor>>
acd : AuthorCredentialsDB
<<actor>>
es : EmailSystem
createNewBlogAccount
selectBlogAccountType(type)
enterAuthorDetails(author : AuthorDetails)
checkAuthorDetails(author : AuthorDetails)
createNewRegularBlogAccount (author : AuthorDetails)
emailBlogDetails (regularBlogAccount)
sendEmail(email : Email)

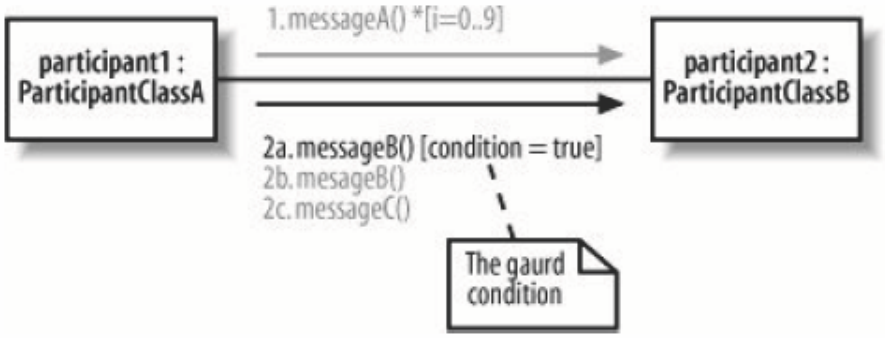

<<actor>>
admin : Administrator
ui :
AccountCreationUI
controller :
CreateNewAccountController
<<actor>>
acd : AuthorCredentialsDB
<<actor>>
es : EmailSystem
createNewBlogAccount()
selectBlogAccountType(type)
enterAuthorDetails (author : AuthorDetails)
clickSubmit()
createNewRegularBlogAccount(author : AuthorDetails)
checkAuthorDetails(author : AuthorDetails)
checkNewRegularBlogAccount (author : AuthorDetails)
emailBlogDetails (regularBlogAccount)
sendEmail(email : Email)

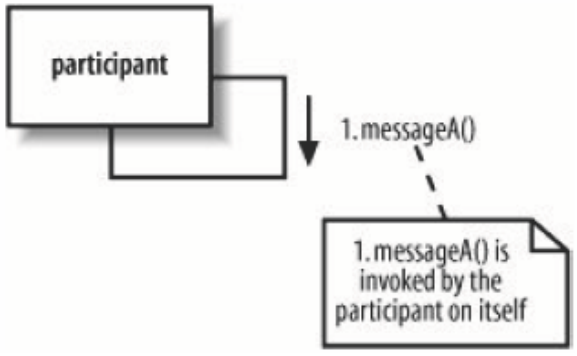

sd diagram1
sequence diagram frame
Actor
:ClassA
object
lifeline
message1(param)
message1(param:Type)
creation
ClassB("abc")
aB
:ClassB
result
:Type
message2(:Type)
info=message2()
loop frame
loop
[condition]
message1()
result:Type
message2()
activation bar
selfcall()
ref
diagram2(info)
stop
reference frame


participant1
participant2
participant3
message1(..)
message2(..)
The initial message
The nested message


The initial message that causes nested messages to be invoked
Nested messages invoked when 1.messageA() is invoked
1.messageA()
2.messageB()
1.1 messageC()
1.2 messageD()
participant1 : ParticipantClassA
participant2 : ParticipantClassB
participant3 : ParticipantClassC


The looping constraint
1.messageA() *[i=0..9]
2a.messageB()
2b.messageB()
2c.messageC()
participant1 : ParticipantClassA
participant2 : ParticipantClassB


1.messageA() *[i=0..9]
2a.messageB() [condition = true]
2b.mesageB()
2c.messageC()
The gaurd condition
participant1 : ParticipantClassA
participant2 : ParticipantClassB


participant
1.messageA()
1.messageA() is invoked by the participant on itself