

TEORIE IS

Diagrame de interactiune sau in diagrame de stare:

- Obiect care comunica prin mesaje cu alte obiecte
- **Actiune**
- Activitate
- O stare intr-o diagrama cu tranzitii declansate de trecerea unor intervale de timp
- **Tranzitie**
- Actor

Diagrama de comunicare , Clasele celor doua obiecte care interactioneaza pot fi conectate printr-o relatie de :

- Generalizare
- **Asociere**
- Mostenire
- Agregare
- Realizare
- **Dependenta**

PROCES UNIFICAT

- Consuma cele mai multe resurse :
 - Dezvoltare
 - Tranzitie
 - **Constructie**
 - Elaborare
 - Evolutie
 - Initiere
- >1 iteratie : **ITEC** (Initiere, Constructie, Tranzitie, Elaborare)

- Procesul unificat este orientat pe componente, utilizeaza limbajul UML pentru construirea modelelor orientate obiect, si are urmatoarele 3 caracteristici distinctive :
 - Ghidat pe cazuri de utilizare
 - Centrat pe arhitectura
 - Iterativ si incremental
- Arhitectura sistem se dezvolta in principal in : **Elaborare**

Caz de utilizare:

- sa descrie interacțiunea utilizatorului cu sistemul
- sa descrie pasii de calcul cei mai important pe care ii efectueaza sistemul
- sa descrie fluxul principal al unui calcul efectuat de sistem
- sa descrie designul interfetei utilizator din perspectiva unui actor particular
- sa descrie calculele pe care le efectueaza sistemul
- sa include numai actiunile prin care un actor interactioneaza cu sistemul

Activitati de baza in ingineria software (care se realizeaza impreuna cu unit testing)

- Specificare
- Proiectare sistem
- Analiza
- Proiectare de detaliu
- **Implementare**
- Intretinere
- Integrare

Specificare formală a unui sistem software este un model mathematic care:

- Face posibila verificarea formală
- In cazul unui sistem complex se dezvolta inaintea arhitecturii software
- Face posibila reducerea costurilor testarii software
- Reprezinta o implementare precisa a principalelor componente software
- Nu este potrivita pentru proiecte cu cerinte non-functionale stringente

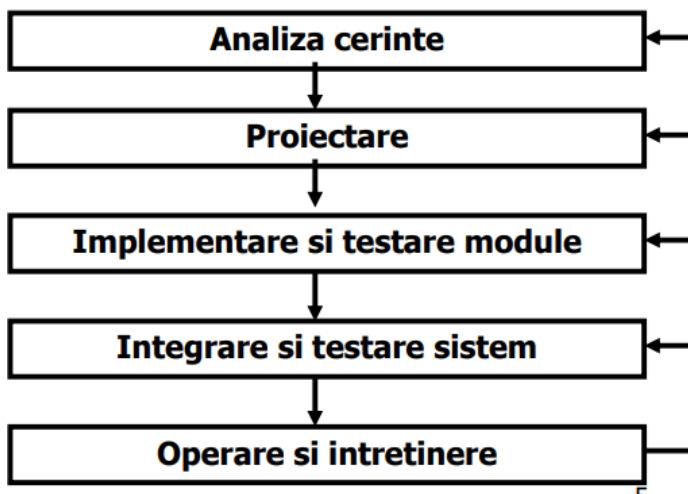
Markov Decision Process (MDPs)

- Un MDP poate modela atât comportamentul probabilist cât și nedeterminist, exemplu avem 2 stări neetichetate care sunt selectate într-o manieră nedeterministica. Fiecare acțiune în MDP are o probabilitate distribuită.

EXEMPLU

```
mdp
module M1
v1 : [0..1] init 0;
[] v1=0 & v2=0 -> 0.9:(v1'=0) + 0.1:(v1'=1);
[a] v1=0 & v2=1 -> 1:(v1'=1);
[b] v1=1 -> 1:true;
Endmodule
module M1
v1 : [0..1] init 0;
[] v1=0 & v2=0 -> 0.9:(v1'=0) + 0.1:(v1'=1);
[a] v1=0 & v2=1 -> 1:(v1'=1);
[b] v1=1 -> 1:true;
Endmodule
+diagrama
```

Cascada



Testarea incrementală

- Se proiectează module test driver pentru testare de integrare -> bottom UP
- Se proiectează module simulator stubs pentru testare de integrare -> top down

Cuplarea:

- Cuplarea de marca apare atunci cand una dintre clasele aplicatiei este utilizata ca tip pentru argumentul unei metode
- Cuplarea de marca se poate reduce prin :
 - Utilizarea unei interfete ca tip al argumentului.
 - Transmiterea de date simple
- Cuplarea externă apare atunci cand un modul are o dependență semantică față de :
 - sistemul de operare,
 - pachete software produse de o alta firma,
 - componente hardware
- Ordonati urmatoarea lista de tipuri de cuplare (modul sau subsistem) în ordinea descrescătoare a gradului de cuplare: prin apel rutina, prin date globale, utilizare tip, continut, control, import, marca, date(simple):
 - Continut
 - Prin date globale
 - Control
 - Marca
 - Date (simple)
 - Prin apel rutina
 - Utilizare tip
 - Import

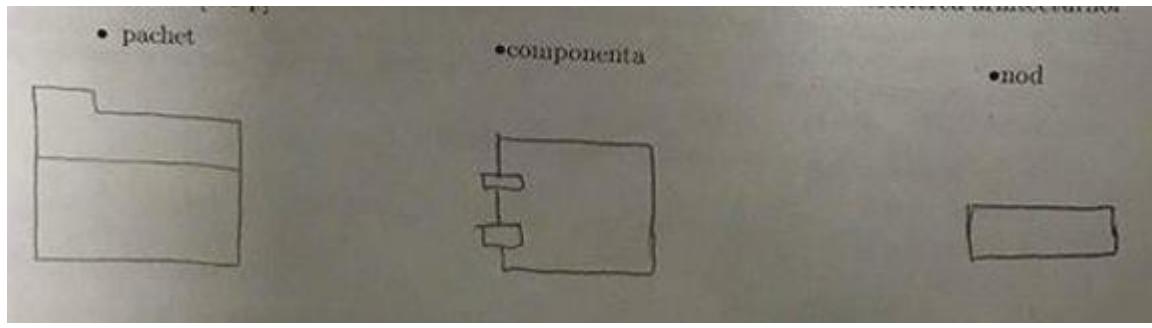
Facade:

- Sablonul de design **facade** poate reduce cuplarea externă prin simplificarea interfetei spre funcțiile externe

Coeziune:

- Coeziunea de comunicare este realizată când toate modulele(elementele de procesare) care **accesează sau manipulează anumite date** sunt păstrate împreună și orice altă funcționalitate este plasată în alta parte a sistemului.
- Ordonați urmatoarea lista de tipuri de coeziune (modul sau subsistem) în ordinea descrescătoare a gradului de coeziune:procedurală , de nivel, secentială, funcțională, temporală, comunicare.(fncspt)
 - **Coeziune funcțională**
 - **Coeziune de nivel**
 - **Coeziune de comunicare**
 - **Coeziune de secvențiere**
 - **Coeziune procedurală**
 - **Coeziune temporală**
- Efecte laterale pot să apară numai pentru module sau subsisteme care manifestă
 - **Coeziune de nivel**

SIMBOLURI UML:



CFG:

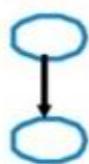
- Complexitatea ciclomatica atasata unui CFG reprezinta numarul maxim de **cai independente (path-uri)** care trebuie sa fie testate pentru a se acoperi toate muchiile posibile.
- $cc(\text{CFG}) = \text{nr_de_muchii} - \text{nr_de_vf} + 2 = \text{noBinaryDecision Predicate} + 1$
- un graf orientat cu un singur punct de intrare & un singur punct de iesire
- Toate muchiile CFG pentru aceasta metoda

In proiectarea prin contract , comportamentul unei metode este descris printr-un set de asertii care stabilesc :

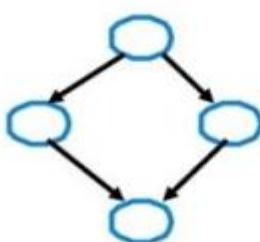
- **preconditiile** (solicitante de metoda inaintea executiei),
- **postconditiile** (pe care metoda le garanteaza la sfarsitul executiei)
- **invariantii** (pe care metoda se angajeaza sa nu ii modifice in timpul executiei).

Desenati graful de fluanta al controlului pentru constructiile de baza din programarea imperativa structurata:

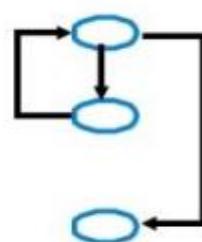
Sequence



If



While



Completați partea (partile lipsă în paragraful urmator):

- **Componenta** asigura o interfata unificata la un set de interfete dintr-un subsistem. **Facade** defineste o interfata simplificata (de nivel inalt) care face subsistemul mai usor de utilizat.
- **Design Pattern Observer** defineste o dependenta de tip one-to-many (unul sau mai multi) intre obiecte astfel incat atunci cand un obiect isi modifica starea toate obiectele care depind de obiectul respectiv sunt notificate si actualizate automat.
- Paradigma **O O** favorizeaza coeziunea de comunicare.
- **Proxy Pattern** furnizeaza un inlocuitor(sau un substitut) pentru un alt obiect pentru a controla accesul la obiectul respectiv.
- **Design Pattern Adapter** poate fi utilizat pentru a se converti interfata unei clase intr-o altainterfata ceruta de client, permitand astfel ca (unele) clase cu interfata incompatibila sa functioneze impreuna (power of polymorphism)
- **Abstraction Occurrence Pattern** (fara a duplica informatia comună)
- Un caz de utilizare este o descriere a unui set de **secvențe de acțiuni** pe care el efectueaza un sistem pentru a produce un rezultat observabil si semnificativ pentru un anumit **actor**.
- O diagrama de activitate este un tip de **Diagrama de tranzitie**
- Intr-o diagrama de activitate starile sunt stari de **activitate**.
- **Atunci** cand activitatea starii curente se termina

Dati un exemplu de diagrama UML de component care reprezinta o arhitectura pipe-and-filter (evidențiati componentele și dependentele între componente):

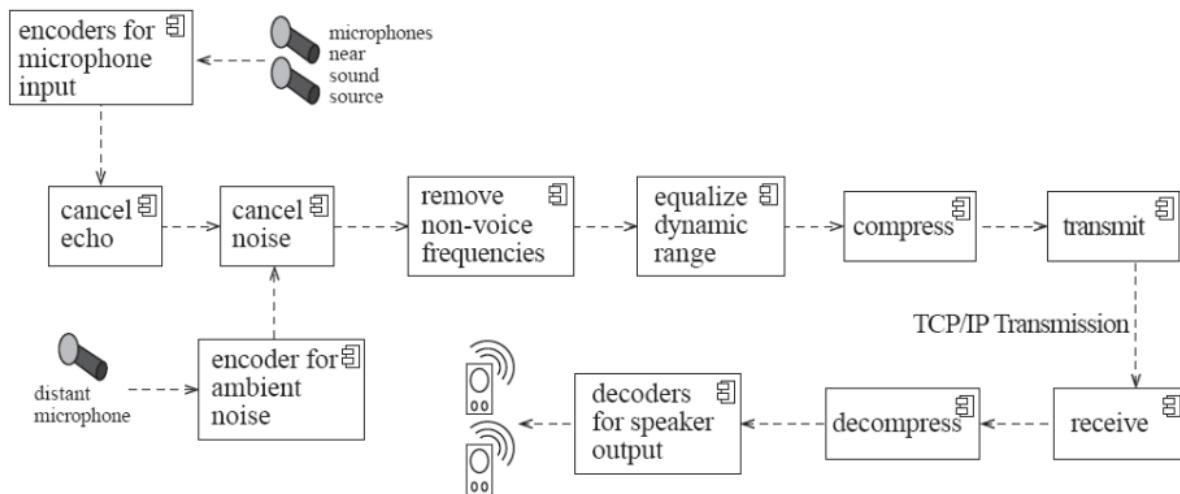
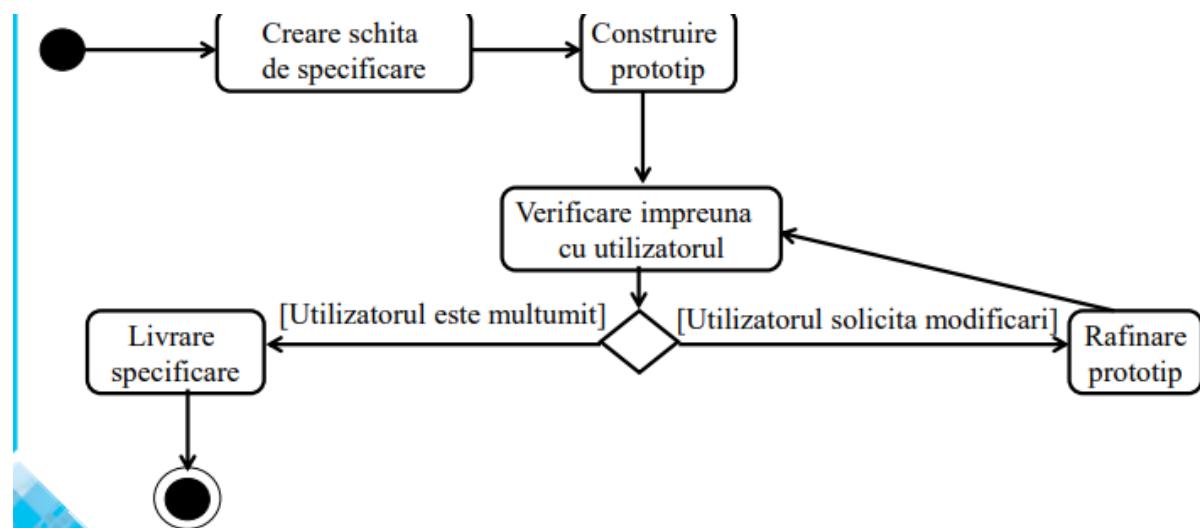
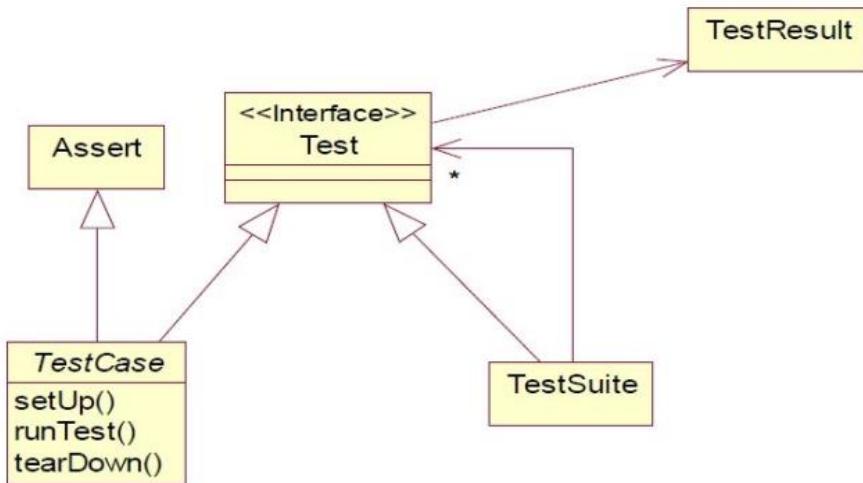


Diagrama de activitate assembly

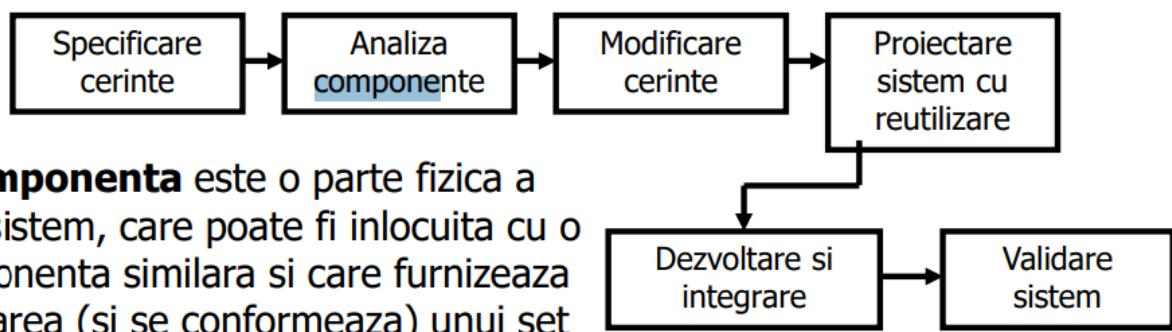


Desenati diagrama UML de clase care descrie designul pentru JUnit 3.x framework.

- Desenati diagrama UML de clase care descrie designul pentru JUnit 3.x framework.



Desenati diagrama de activitate pentru modelul de dezvoltare orientate pe componente.



Care dintre urmatoarele paradigmă de dezvoltare software include o fază de modificare a cerintelor?

- **Componente reutilizabile**
- **Inginerie concurrentă**

In timp ce toate formele de testare prezentate mai jos sprijina activitate de verificare software (are we building the product right?) , care dintre acestea sprijina in cea mai mare masura activitatea de validare software(are we building the right product?)

- **Testare de acceptanta**

Dificultatea in modelul cascada(waterfall) este :

- **Backtrack catre o stare anterioara**
- **Handle requirements modification**

Selectati practicile si instrumentele utilizate in Programarea extrema/Extreme pentru stabilirea cerintelor software:

- **Scenarii XP**
- **Ritm sustinut, fara excese**
- **Refactorizare**
- **Dezv ghidata de testare**

Se considera urmatoarea lista de categorii de cerinte. Selectati categoriile de cerinte non functionale:

- **Fiabilitatea**
- **Timpul de rasp al sis**
- **Mentanabilitatea**
- **Procesul sau metodologia de dezvoltare**

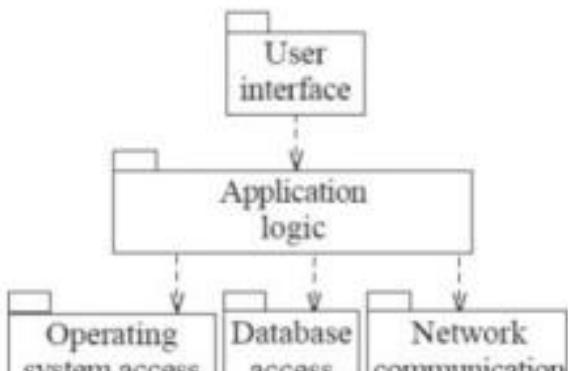
Completați parțial lipsa în paragraful următor:

- **In cazul unumitor clase, costul operatiilor de creare și initializare a instantelor este ridicat, deoarece baza de date **Proxy****

Completați campurile lipsă conform dicționariei tip/ instanță din UML

- Clasa/Obiect
- Caz de utilizare/ **scenariu**
- Asociere / **conexiune**

Dati un exemplu de diagrama de pachete UML care reprezinta o arhitectura multi-layer (evidențiați subsistemele și dependențele între acestea)



(a) Typical layers in an application program

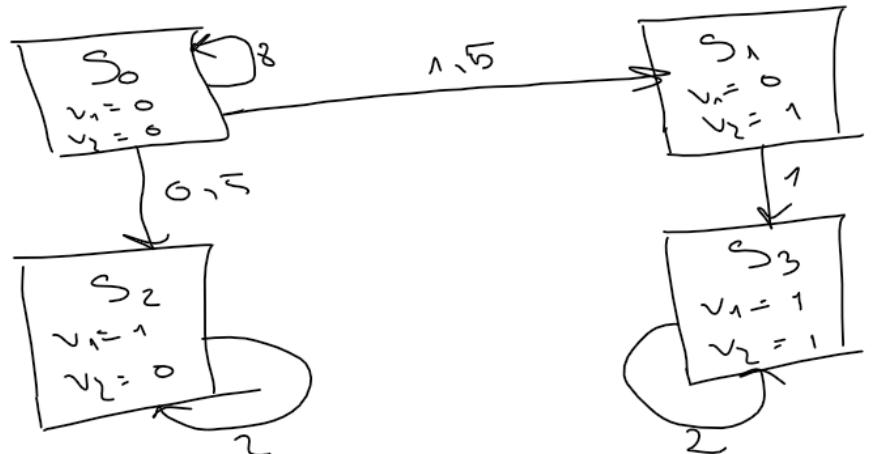
Prism

```

1. ctmc
2. module M1
3. v1 : [0..1] init 0;
4. [] v1=0 & v2=0 -> 4.5:(v1'=0) + 0.5:(v1'=1);
5. [a] v1=0 & v2=1 -> 1:(v1'=1);
6. [b] v1=1 -> 1:true;
7. endmodule
8. module M2
9. v2 : [0..1] init 0;
10. [] v1=0 & v2=0 -> 3.5:(v2'=0) + 1.5:(v2'=1);
11. [a] v1=0 & v2=1 -> 1:true;
12. [b] v1=1 -> 2:true;
13. endmodule
14. Module M
15. v1 : [0..1] init 0;
16. v2 : [0..1] init 0;
17. [] v1=0 & v2=0 -> 4.5:(v1'=0) + 0.5:(v1'=1); //c1
18. [] v1=0 & v2=0 -> 3.5:(v2'=0) + 1.5:(v2'=1); //c2
19. [a] v1=0&v2=1&v1=0&v2=1 -> 1:(v1'=1)&true;//c3
20. [b] v1=1&v1=1 -> 2:true&true;//c4
21. S={s0,s1,s2,s3} (00 01 10 11)
22. Sc1={s0} Sc2={s0} Sc3={s1} Sc4={s2,s3}
23. μc1,s0 = [s0->4.5,s1->0,s2->0.5,s3->0];
24. μc2,s0 = [s0->3.5,s1->1.5,s2->0,s3->0];
25. μc3,s1 = [s0->0,s1->0,s2->0,s3->1];
26. μc4,s3 = [s0->0,s1->0,s2->0,s3->2];
27. μc4,s2 = [s0->0,s1->0,s2->2,s3->0];
28.

```

	s0	s1	s2	s3
s0	8	1.5	0.5	0
s1	0	0	0	1
s2	0	0	2	0
s3	0	0	0	2



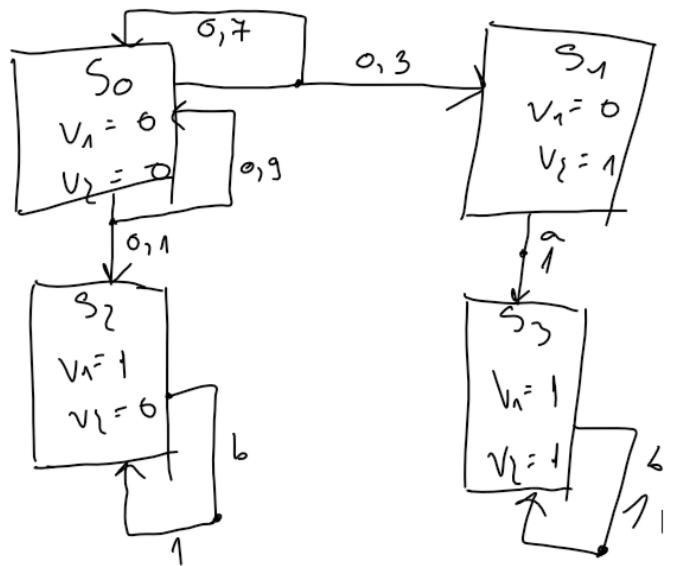
Sunday, January 22, 2023 12:16 PM

```

1. MDP
2. module M1
3. v1 : [0..1] init 0;
4. [] v1=0 & v2=0 -> 0.9:(v1'=0) + 0.1:(v1'=1);
5. [a] v1=0 & v2=1 -> 1:(v1'=1);
6. [b] v1=1 -> 1:true;
7. endmodule
8. module M2
9. v2 : [0..1] init 0;
10. [] v1=0 & v2=0 -> 0.7:(v2'=0) + 0.3:(v2'=1);
11. [a] v1=0 & v2=1 -> 1:true;
12. [b] v1=1 -> 1:true;
13. endmodule
14. Module M
15. v1 : [0..1] init 0;
16. v2 : [0..1] init 0;
17. [] v1=0 & v2=0 -> 0.9:(v1'=0) + 0.1:(v1'=1); //c1
18. [] v1=0 & v2=0 -> 0.7:(v2'=0) + 0.3:(v2'=1); //c2
19. [a] v1=0&v2=1&v1=0&v2=1 -> 1:(v1'=1)&true;//c3
20. [b] v1=1&v1=1 -> 2:true&true;//c4
21. S={s0,s1,s2,s3} (00 01 10 11)
22. Sc1={s0} Sc2={s0} Sc3={s1} Sc4={s2,s3}
23. μc1,s0 = [s0->0.9,s1->0,s2->0.1,s3->0];
24. μc2,s0 = [s0->0.7,s1->0.3,s2->0,s3->0];
25. μc3,s1 = [s0->0,s1->0,s2->1,s3->1];
26. μc4,s2 = [s0->0,s1->0,s2->0,s3->0];
27. μc4,s3 = [s0->0,s1->0,s2->0,s3->1];
28. Steps(s0)=[μc1s0,μc2s0]
  =[{s0->0.9,s1->0,s2->0.1,s3->0},s0->0.7,s1->0.3,s2->0,s3->0]
29. Steps(s1)=[μc3s1] = [{s0->0,s1->0,s2->0,s3->1}]
30. Steps(s2)=[μc4,s2] = [{s0->0,s1->0,s2->1,s3->0}]
31. Steps(s3)=[μc4,s3] = [{s0->0,s1->0,s2->0,s3->1}]

```

1. Steps'(s0) = [↑, μc1s0, μc2s0]
- 2.
3. =[↑, s0->0.9, s1->0, s2->0.1, s3->0], {↑, s0->0.7, s1->0.3, s2->0, s3->0}]
4. Steps'(s1) = [a, μc3s1] = [{a, s0->0, s1->0, s2->0, s3->1}]
5. Steps'(s2) = [b, μc4, s2] = [{b, s0->0, s1->0, s2->1, s3->0}]
6. Steps'(s3) = [b, μc4, s3] = [{b, s0->0, s1->0, s2->0, s3->1}]



(d) Se consideră următoarea specificare PRISM a unui sistem software modelat ca DTMC (Discrete-Time Markov Chain)

```

dtmc
module M1
    var1 : [0..3] init 0;
        []      var2<2          -> 0.25:(var1'>var2+1)+0.75:true;
    [interact1] var1=0 & var2<2 -> 0.55:(var1'>var2+1)+0.45:(var1'>var1+1);
    [interact2] var1>0         -> 0.43:(var1'=0)+0.57:(var1'>var1-1);
endmodule
module M2
    var2 : [0..3] init 0;
    [interact1] var2=0 & var1<2 -> 0.06:(var2'>var1+1)+0.95:(var2'>var2+1);
    [comm]   var1<2           -> 0.25:(var2'>var1+1) + 0.75:true;
    [interact2] var2>0         -> 0.33:(var2'=0)+0.33:(var2'>var2-1)+0.34:true;
endmodule

```

- Sa se proiecteze o structura de recompensa numita "str", care atribuie recompensa 1 oricarei stari in care var1=0 si tot recompensa 1 oricarei transiti care are eticheta de tranzitie "interact1" dintr-o stare in care variabila var1 are valoarea 0. [0.5p]
- Sa se elaboreze o proprietate care calculeaza valoarea medie (engl. expected value) a recompensei cumulata (in raport cu "str") pana cand sistemul ajunge eventual intr-o stare in care var1=2. [0.25p]

rewards "str"

$V_{\text{var1}} = 0.1$

$[C \text{ to d1}] V_{\text{var1}} = 0.1$

ind rewards;

$R\{\text{"str"}\} = ? [F \exists \text{var1} \leq 2]$

ctmc						
module M						
b_1 : bool init false;						
b_2 : bool init false;						
b_3 : bool init false;						
$\square b_2 = \text{false} \& b_3 = \text{false} \rightarrow 4: (b'_1 = \text{true}), // C_1$						
$\square b_2 = \text{false} \& b_3 = \text{true} \rightarrow 1: (b'_1 = \text{true}) + 3.25: (b'_1 = \text{false}), // C_2$						
$\square b_1 = \text{false} \& b_2 = \text{false} \& b_3 = \text{false} \rightarrow 0.25: (b'_1 = \text{false} \& b_2 = \text{false}) + 2: (b'_1 = \text{true} \& b_2 = \text{false}), // C_3$						
$\square b_1 = \text{true} \& b_2 = \text{false} \rightarrow 4: (b'_1 = \text{false}) + 5: (b'_1 = \text{true}), // C_4$						
$\square b = \text{true} \rightarrow 7: (b'_1 = \text{false}) \& (b'_2 = \text{false}) + 5: (b'_1 = \text{true}), // C_5$						
endmodule						
$S = \{ \lambda_0, \lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5, \lambda_6, \lambda_7 \}$	$S_0 = \{ 0, 0, 0 \}$	$S_1 = \{ 0, 1, 0 \}$	$S_2 = \{ 0, 1, 1 \}$	$S_3 = \{ 0, 1, 0 \}$	$S_4 = \{ 0, 0, 1 \}$	$S_5 = \{ 0, 0, 0 \}$
$\mu_{C_1, \lambda_0}(\lambda_1) = 4;$	$\mu_{C_2, \lambda_0}(\lambda_1) = 0.25$	$\mu_{C_3, \lambda_0}(\lambda_1) = 7$	$\mu_{C_4, \lambda_0}(\lambda_1) = 1$	$\mu_{C_5, \lambda_0}(\lambda_1) = 5$	$\mu_{C_1, \lambda_1}(\lambda_2) = 1$	$\mu_{C_2, \lambda_1}(\lambda_2) = 3.25$
$\mu_{C_1, \lambda_1}(\lambda_2) = 5;$	$\mu_{C_2, \lambda_1}(\lambda_2) = 2$	$\mu_{C_3, \lambda_1}(\lambda_2) = 5$	$\mu_{C_4, \lambda_1}(\lambda_2) = 5$	$\mu_{C_5, \lambda_1}(\lambda_2) = 5$	$\mu_{C_1, \lambda_2}(\lambda_3) = 1$	$\mu_{C_2, \lambda_2}(\lambda_3) = 4$
$\mu_{C_1, \lambda_2}(\lambda_3) = 1$	$\mu_{C_2, \lambda_2}(\lambda_3) = 4$	$\mu_{C_3, \lambda_2}(\lambda_3) = 7$	$\mu_{C_4, \lambda_2}(\lambda_3) = 7$	$\mu_{C_5, \lambda_2}(\lambda_3) = 7$	$\mu_{C_1, \lambda_3}(\lambda_4) = 1$	$\mu_{C_2, \lambda_3}(\lambda_4) = 5$
$\mu_{C_1, \lambda_3}(\lambda_4) = 1$	$\mu_{C_2, \lambda_3}(\lambda_4) = 5$	$\mu_{C_3, \lambda_3}(\lambda_4) = 1$	$\mu_{C_4, \lambda_3}(\lambda_4) = 1$	$\mu_{C_5, \lambda_3}(\lambda_4) = 1$	$\mu_{C_1, \lambda_4}(\lambda_5) = 1$	$\mu_{C_2, \lambda_4}(\lambda_5) = 3.25$
$\mu_{C_1, \lambda_4}(\lambda_5) = 1$	$\mu_{C_2, \lambda_4}(\lambda_5) = 3.25$	$\mu_{C_3, \lambda_4}(\lambda_5) = 1$	$\mu_{C_4, \lambda_4}(\lambda_5) = 1$	$\mu_{C_5, \lambda_4}(\lambda_5) = 1$	$\mu_{C_1, \lambda_5}(\lambda_6) = 1$	$\mu_{C_2, \lambda_5}(\lambda_6) = 5$
$\mu_{C_1, \lambda_5}(\lambda_6) = 1$	$\mu_{C_2, \lambda_5}(\lambda_6) = 5$	$\mu_{C_3, \lambda_5}(\lambda_6) = 1$	$\mu_{C_4, \lambda_5}(\lambda_6) = 1$	$\mu_{C_5, \lambda_5}(\lambda_6) = 1$	$\mu_{C_1, \lambda_6}(\lambda_7) = 1$	$\mu_{C_2, \lambda_6}(\lambda_7) = 5$
$\mu_{C_1, \lambda_6}(\lambda_7) = 1$	$\mu_{C_2, \lambda_6}(\lambda_7) = 5$	$\mu_{C_3, \lambda_6}(\lambda_7) = 1$	$\mu_{C_4, \lambda_6}(\lambda_7) = 1$	$\mu_{C_5, \lambda_6}(\lambda_7) = 1$	$\mu_{C_1, \lambda_7}(\lambda_0) = 1$	$\mu_{C_2, \lambda_7}(\lambda_0) = 5$
$\mu_{C_1, \lambda_7}(\lambda_0) = 1$	$\mu_{C_2, \lambda_7}(\lambda_0) = 5$	$\mu_{C_3, \lambda_7}(\lambda_0) = 1$	$\mu_{C_4, \lambda_7}(\lambda_0) = 1$	$\mu_{C_5, \lambda_7}(\lambda_0) = 1$		

Haskell to C

2. Consider the following Haskell declarative specification: [4.5p]

```
data F = Fnil | Fa Int F | Fs Int [F] | Fb F Int F
ff :: F -> [Int] -> [Int]
ff Fnil ys = ys
ff (Fa n f) ys = n:ff f ys
ff (Fs n fs) ys = n:ffs fs ys
ff (Fb Fnil n r) ys = n:ff r ys
ff (Fb (Fa nf f) n r) ys = nf:ff f (n: ff r ys)
ff (Fb (Fs ln lfs) n r) ys = ln:ffs lfs (n:ff r ys)
ff (Fb (Fb ll ln lr) n r) ys = ff (Fb ll ln (Fb lr n r)) ys

ffs :: [F] -> [Int] -> [Int]
ffs [] ys = ys
ffs (f:fs) ys = ff f (ffs fs ys)

ex :: F
ex = Fb (Fb Fnil 1 (Fs 2 [Fnil, Fa 3 Fnil, Fa 4 Fnil])) 5 (Fb (Fb Fnil 6 (Fs 7 []))) 8 Fnil
{-
Main> ff ex [0,0]
[1,2,3,4,5,6,7,8,0,0]
-}
```

- Prove the correctness of function 'ff' by mathematical induction. [1p]
- Implement the specification by using an imperative language:
 - Data structures [1.75p]
 - All the procedures [1.75p]

```

#include <stdio.h>
#include <stdlib.h>

typedef struct elem{
    int info;
    elem* next;
}ELEM, *LIST;

typedef enum {FA, FS, FB} SEL;

typedef struct F *F;

typedef struct elemf{
    F info;
    elemf* next;
}ELEMF, *LISTF;

typedef struct F{
    SEL sel;
    union{
        struct{
            int n;
            struct F* f;
        }fa;
        struct{
            int n;
            LISTF fs;
        }fs;
        struct{
            struct F *f1;
            int n;
            struct F *f2;
        }fb;
    }f;
}CEL, *F;

F Fnil(){
    return 0;
}

F Fa(int n, F f){
    F fans;
    fans = (F)malloc(sizeof(CEL));
    fans->sel = FA;
    fans->f.fa.n = n;
    fans->f.fa.f = f;
    return fans;
}

```

1

```

F Fs(int n, LISTF fs){
    F fans;
    fans = (F)malloc(sizeof(CEL));
    fans->sel = FS;
    fans->f.fs.n = n;
    fans->f.fs.fs = fs;
    return fans;
}

F Fb(F f1, int n, F f2){
    F fans;
    fans = (F)malloc(sizeof(CEL));
    fans->sel = FB;
    fans->f.fb.n = n;
    fans->f.fb.f1 = f1;
    fans->f.fb.f2 = f2;
    return fans;
}

F ex(){
    LISTF listfaux = (LISTF)malloc(sizeof(ELEMF));
    listfaux->info = Fnil();
    listfaux->next = (LISTF)malloc(sizeof(ELEMF));
    listfaux->next->info = Fa(3, Fnil());
    listfaux->next->next = (LISTF)malloc(sizeof(ELEMF));
    listfaux->next->next->info = Fa(4, Fnil());
    listfaux->next->next->next = 0;

    return Fb( Fb( Fnil(), 1, Fs(2, listfaux)), 5, (Fb( Fb(Fnil(), 6, Fs( 7, 0 )), 8, Fnil() ) ) );
}

void showFs(LISTF fs);

void show(F f){
    if(f == 0) printf("NIL");
    else if(f->sel == FA ){
        printf("( %d, ", f->f.fa.n);
        show(f->f.fa.f);
        printf(" )");
    }
    else if( f->sel == FS ){
        printf("( %d, ", f->f.fs.n);
        printf("[ ");
        showFs(f->f.fs.fs);
        printf(" ], ");
        printf(" )");
    }
    else if( f->sel == FB ){
        printf("( ");
        show(f->f.fb.f1);
        printf(", %d, ", f->f.fb.n);
        show(f->f.fb.f2);
        printf(" )");
    }
    else{
        printf("UNKNOWN");
    }
}

void showFs(LISTF fs){
    if( fs == 0 ) {
        printf("NIL");
        return;
    }
    show( fs->info );
    showFs(fs->next);
}

void showLs(LIST ns){
    if( ns == 0 ) {
        printf("NIL");
        return;
    }
    printf("%d, ", ns->info);
    showLs(ns->next);
}

LIST ff(F f, LIST ys);

LIST ffs(LISTF fs, LIST ys){
    if( fs == 0 ) return ys;

    ff( fs->info, ffs(fs->next, ys) );
}

LIST ff(F f, LIST ys){
    LIST ans, ansaux;

    if( f == 0 ) return ys;
    else if(f->sel == FA ){
        ans = (LIST)malloc(sizeof(ELEM));
        ans->info = f->f.fa.n;
        ans->next = ff(f->f.fa.f, ys);
        return ans;
    }
    else if(f->sel == FS ){
        ans = (LIST)malloc(sizeof(ELEM));
        ans->info = f->f.fs.n;
        ans->next = ffs(f->f.fs.fs, ys);
        return ans;
    }
    else if(f->sel == FB && f->f.fb.f1 == 0 ){
        ans = (LIST)malloc(sizeof(ELEM));
        ans->info = f->f.fb.n;
        ans->next = ff(f->f.fb.f2, ys);
        return ans;
    }
    else if(f->sel == FB && f->f.fb.f1->sel == FA ){
        ans = (LIST)malloc(sizeof(ELEM));
        ans->info = f->f.fb.f1->f.fa.n;
        ansaux = (LIST)malloc(sizeof(ELEM));
        ansaux->info = f->f.fb.n;
        ansaux->next = ff(f->f.fb.f2, ys);
        ans->next = ff(f->f.fb.f1->f.fa.f, ansaux);
        return ans;
    }
    else if(f->sel == FB && f->f.fb.f1->sel == FS ){
        ans = (LIST)malloc(sizeof(ELEM));
        ans->info = f->f.fb.f1->f.fs.n;
        ansaux = (LIST)malloc(sizeof(ELEM));
        ansaux->info = f->f.fb.n;
        ansaux->next = ff(f->f.fb.f2, ys);
        ans->next = ffs(f->f.fb.f1->f.fs.fs, ansaux);
        return ans;
    }
    else if(f->sel == FB && f->f.fb.f1->sel == FB ){
        return ff(Fb( f->f.fb.f1->f.fb.f1, f->f.fb.f1->f.fb.f2 ), ys);
    }
}

```

2

```

LIST ff(F f, LIST ys){
    LIST ans, ansaux;

    if( f == 0 ) return ys;
    else if(f->sel == FA ){
        ans = (LIST)malloc(sizeof(ELEM));
        ans->info = f->f.fa.n;
        ans->next = ff(f->f.fa.f, ys);
        return ans;
    }
    else if(f->sel == FS ){
        ans = (LIST)malloc(sizeof(ELEM));
        ans->info = f->f.fs.n;
        ans->next = ffs(f->f.fs.fs, ys);
        return ans;
    }
    else if(f->sel == FB && f->f.fb.f1 == 0 ){
        ans = (LIST)malloc(sizeof(ELEM));
        ans->info = f->f.fb.n;
        ans->next = ff(f->f.fb.f2, ys);
        return ans;
    }
    else if(f->sel == FB && f->f.fb.f1->sel == FA ){
        ans = (LIST)malloc(sizeof(ELEM));
        ans->info = f->f.fb.f1->f.fa.n;
        ansaux = (LIST)malloc(sizeof(ELEM));
        ansaux->info = f->f.fb.n;
        ansaux->next = ff(f->f.fb.f2, ys);
        ans->next = ff(f->f.fb.f1->f.fa.f, ansaux);
        return ans;
    }
    else if(f->sel == FB && f->f.fb.f1->sel == FS ){
        ans = (LIST)malloc(sizeof(ELEM));
        ans->info = f->f.fb.f1->f.fs.n;
        ansaux = (LIST)malloc(sizeof(ELEM));
        ansaux->info = f->f.fb.n;
        ansaux->next = ff(f->f.fb.f2, ys);
        ans->next = ffs(f->f.fb.f1->f.fs.fs, ansaux);
        return ans;
    }
    else if(f->sel == FB && f->f.fb.f1->sel == FB ){
        return ff(Fb( f->f.fb.f1->f.fb.f1, f->f.fb.f1->f.fb.f2 ), ys);
    }
}

```

3

Neta

```
data A = Nil : B & Int A : N Int [A]

meta :: A -> [Int] -> [Int]
meta Nil ys = ys
meta (N m ls) ys = n:meta ls ys
meta (B Nil m r) ys = m:meta r ys
meta (B (N lm lls) n r) ys = lm:meta lls (n:meta r ys)
meta (B (B ll lm lr) n r) ys = meta (B ll lm (B lr n r)) ys

metla :: [A] -> [Int] -> [Int]
metla [] ys = ys
metla (a:as) ys = meta a (metla as ys)

a :: A
a = B (B (B (N 1 [N 2 [Nil,Nil]])) 3 Nil) 5 (B Nil 7 (N 8 [])))
      10
      (B (N 11 []) 14 (N 17 []))

{-
Main> meta a [0,0,0]
[1,2,3,5,7,8,10,11,14,17,0,0,0]
-}
```

- (a) Prove the correctness of 'meta' by mathematical induction. [2p]
- (b) Implement the specification by using an imperative language:
 - i. Data structures + picture [2.5p]
 - ii. All the procedures [4.5p]

```

#include <stdio.h>
#include <stdlib.h>

typedef struct elem{
    int info;
    elem *next;
}ELEM, *LIST;

typedef struct A *A;

typedef struct list{
    A info;
    elemf *next;
}ELEMf, *LISTf;

typedef enum {B,A,M} SEL;

typedef struct A{
    SEL sel;
    union {
        struct{
            struct A *a1;
            int n;
            struct A *a2;
        }b;
        struct{
            int n;
            LIST *a;
        }m;
        }f;
}CEL,*A;

A Anil(){
    return 0;
}

A B(A a1,int n, A a2)
{
    A ans;
    ans = (F) malloc (sizeof(CEL));
    ans -> sel =B;
    ans ->f.b.a1=a1;
    ans ->f.b.n=n;
    ans ->f.b.a2=a2;
    return ans;
}

A M(int n, LISTf a)
{
    A ans;
    ans = (F) malloc (sizeof(CEL));
    ans -> sel =M;
    ans ->f.m.n=n;
    ans ->f.m.a=a;
    return ans;
}

A a(){
    LISTf listfauxNil = (LISTf)malloc(sizeof(ELEMf));
    listfauxNil->info = 0;
    listfauxNil->next = (LISTf)malloc(sizeof(ELEMf));
    listfauxNil->info = 0;

    LISTf listfaux = (LISTf)malloc(sizeof(ELEMf));
    listfaux->info = M(2,listfauxNil);

    return B(B(B(M 1 listfaux) 3 Nil ) 5 (8 Nil 7 (M 8 [])));
}

LIST netla(A a, List ys)
{
    if(a == 0) return ys;
    else if (a->sel == M)
    {
        ans = (LIST)malloc(sizeof(ELEM));
        ans->info = a ->f.m.n;
        ans->next =netla(a->f.m.a,ys);
        return ans;
    }
    else if (a->sel == B && a->f.b.a1== 0)
    {
        ans = (LIST)malloc(sizeof(ELEM));
        ans->info = a ->f.b.n;
        ans->next =netla(a->f.b.a2,ys);
        return ans;
    }
    else if(a ->sel == B && a->f.b.a1->sel == M)
    {
        aux = (LIST)malloc(sizeof(ELEM));
        aux->info = a ->f.b.n;
        aux->next =netla(a->f.b.a2,ys);

        ans = (LIST)malloc(sizeof(ELEM));
        ans->info = a->f.b.a1->f.m.n;
        ans->next =netla((a->f.b.a1->f.m.a),(aux)) ;
        return ans;
    }
    else if(a ->sel == B && a->f.b.a1->sel == B)
    {
        B b = (B)malloc(sizeof(B));
        b.a1 = a->f.b.a1->f.b.a2;
        b.n= a->f.b.n;
        b.a2 = a->f.b.a2;

        ans = (LIST)malloc(sizeof(ELEM));
        ans->info = a ->f.b.n;
        ans->next =netla(a->f.b.a2,ys);
        return ans;
    }
}

LIST netla(LISTf a, List ys)
{
    if(a == 0) return ys;
    return netla(a->value,netla(a->next,ys));
}

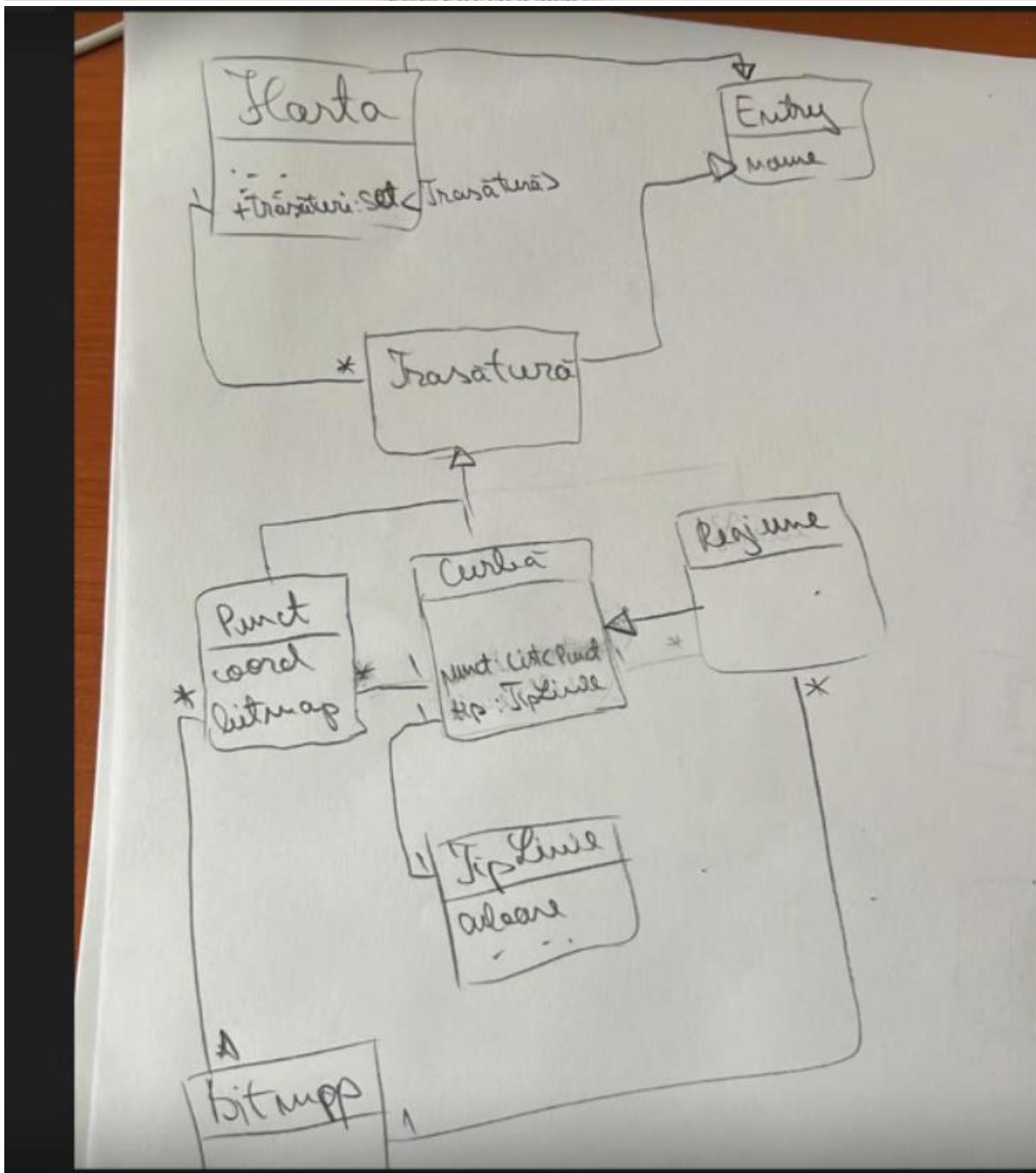
```

2. (a) Mai jos este data descrierea unui sistem informatic pentru proiectare harti (sursa: T. Lethbridge, R. Laganiere, "Object-Oriented Software Engineering", 2nd edition, 2005). Sa se analizeze si sa se proiecteze programul utilizand diagrame de clase UML (se va aplica metodologia data in capitolul "Modelare cu clase"). [5p]

Un sistem utilizat de proiectantii de harti pentru guvernul din Ootumlia contine informatii despre diverse harti. Fiecare hartă are o scară, un nume si un set de trasaturi. De asemenea, fiecare hartă are informatie ce definește latitudinea si longitudinea colțului din stanga-sus al harti, inaltimea si latimea harti in metri.

Trasaturile pot fi de trei tipuri: puncte, curbe si regiuni. In general, fiecare trasatura are un nume care este afisat pe harta langa trasatura. O trasatura de tip punct are o singura coordonata, locatia la care trebuie sa apară pe harta. Exemple de trasaturi punct includ: turnuri de transmisii, varfură (piscuri) de munte, etc. Fiecare tip de trasatura punct are asociat un simbol special; acest simbol este o simplă hartă de biti (bitmap). O trasatura de tip curba contine o lista de puncte care o definesc. Exemple de trasaturi curba includ: drumuri, cai ferate, si rauri. Fiecarui tip de (trasatura) curba ii este asociat un tip de linie. Tipul specifica culoarea liniei, grosimea liniei, lungimea liniutelor si a spatiilor dintre liniute la o linie intrerupta. O trasatura de tip regiune este foarte asemănatoare cu o trasatura de tip curba, exceptand faptul ca descrie o zona completa / inchisa (ultimul punct coincide cu primul punct). In plus, poate avea un sablon optional de umplere, care este o simplă hartă de biti (bitmap) sau o culoare.

DRISM a unui sistem software modelat ca DTMC (Discrete-



UML->Java

WhatsApp Image 2023-01-...

3. Forward engineering (UML → Java) [1.25p]

```

classDiagram
    class F {
        i: int
        +F()
    }
    class A {
        -d: double
        +A()
    }
    class D {
        -m: int
        #n: long
        +D()
    }
    class E {
        -j: int
        +E()
    }
    class C {
        +cs
        +op()
    }
    class B {
        -bs
        *x: char
        +B()
    }

    F --> A : Use
    A *--> D : +pd
    A *--> E : #as
    A *--> C : +cs
    D --> B : -bs
    C --> B : -c
    B --> B : 1
    E --> B : -e
    K --> I
    L --> I
    I --> J
    J --> B
  
```

untitled - Sublime Text (UNREGISTERED)

```

4
5
6 interface I extends K,L(){}
7 interface K {}
8 interface L {}
9 interface J {}
10
11
12 class F implements I
13 {
14     int i;
15     private E e;
16     public F(){ A a = new A();}
17 }
18
19 class E
20 {
21     int j
22     public E(){}
23 }
24
25
26 class A implements J
27 {
28     public D pd;
29     public C [] cs;
30     private double d;
31     public A(){}
32 }
33
34
35 class D
36 {
37     private int m;
38     protected long n;
39     public D(){}
40 }
41
42 class C
43 {
44     protected A []as;
45     private B [] bs;
46     public C(){}
47     void op(){}
48 }
49
50 class B extends E{
51     private C c;
52     private B b;
53     public char x;
54     public B(){}
55 }
56
57
58
59
60
61 
```

Line 23, Column 1

Java ->UML

Forward and reverse engineering

Reverse Engineering (Java -> UML) – Exercise

```

class F extends G implements I3 {
    private K[] ks;
    int i;
    public K k;
    protected H h;
    F() {}
}

class G {
    public G() {
    }
    public void f(H h) {}
}

class H {
    public H() {
    }
}

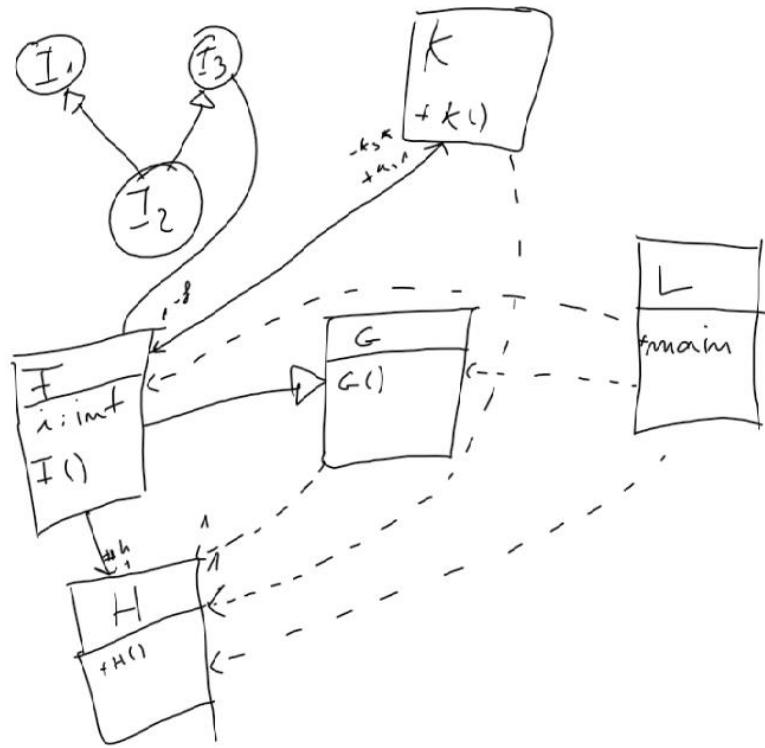
interface I1 {
}

interface I2 extends I1, I3 {
}

interface I3 {
}

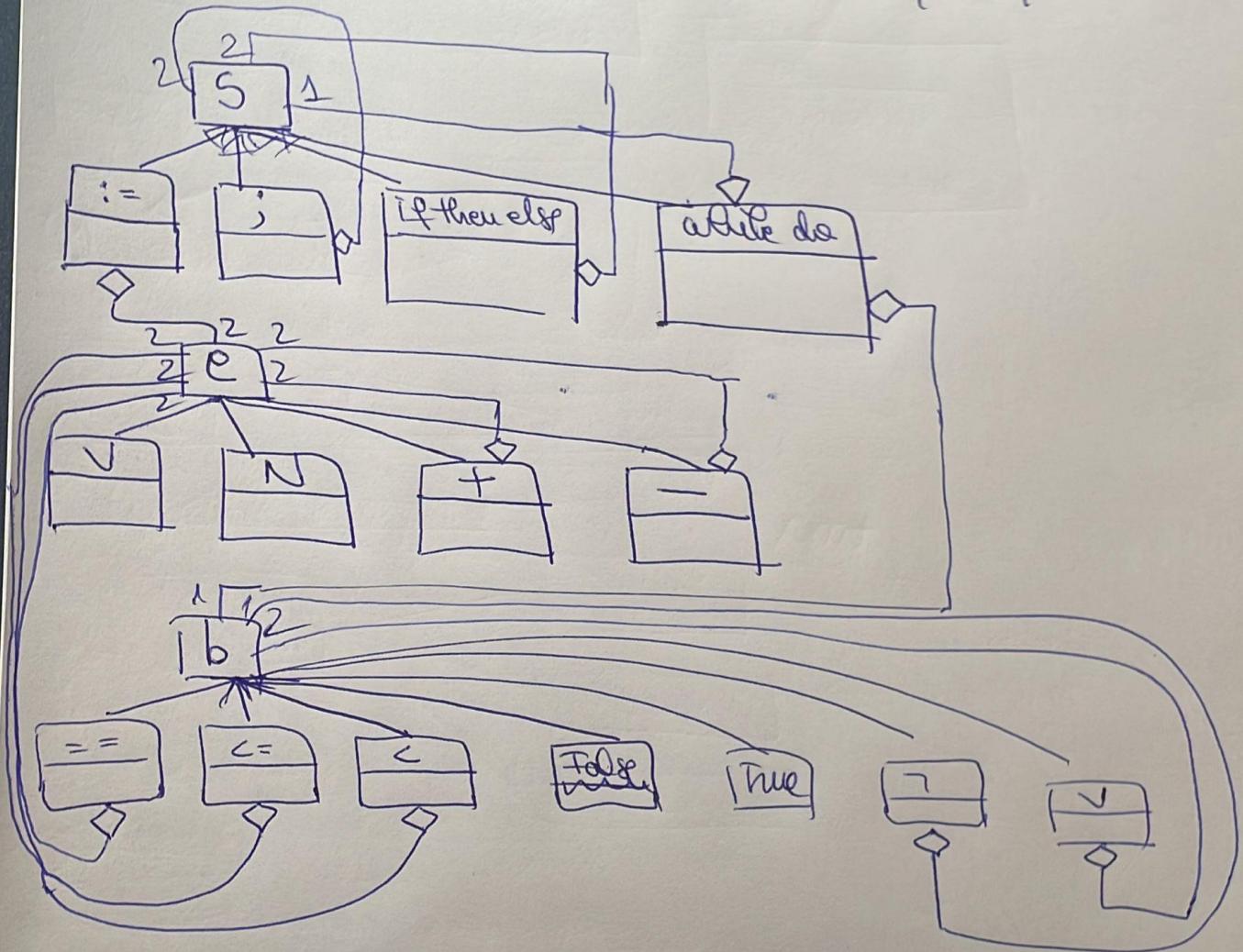
```

Software Engineering 2022

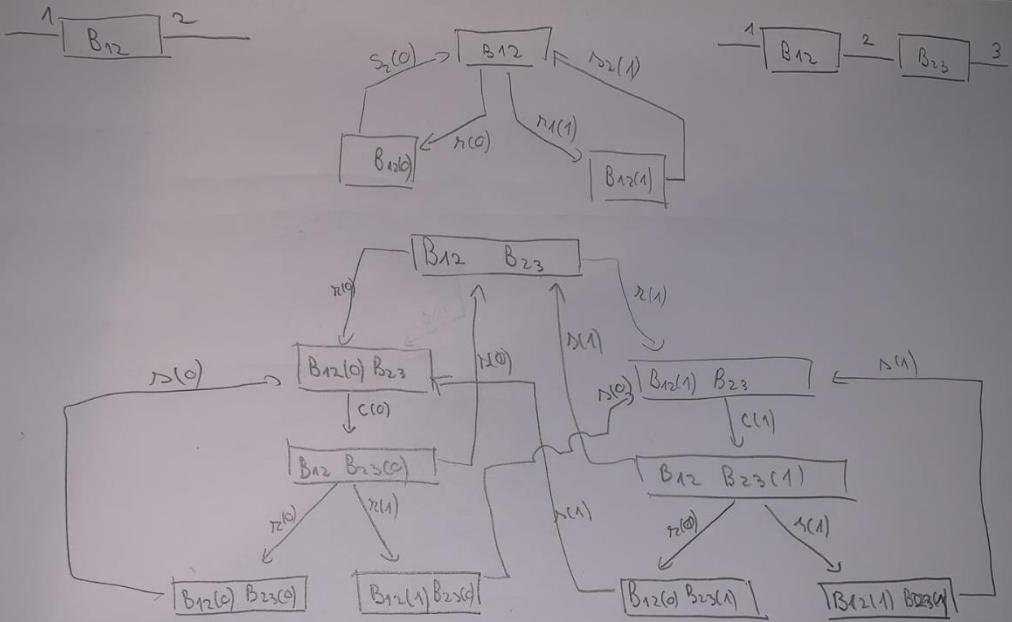


BNF

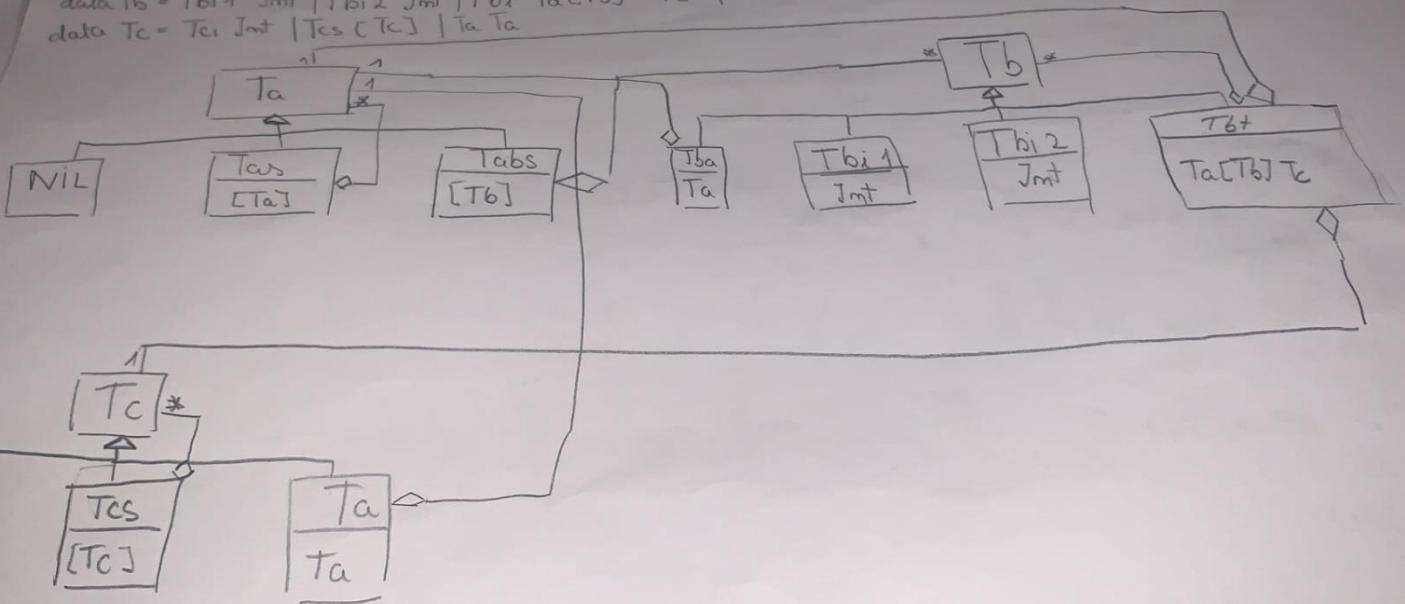
$S ::= v := e \mid s; s \mid \text{if } b \text{ then } s \text{ else } s \mid \text{while } b \text{ do } s$
 $e ::= v \mid n \mid e + e \mid e - e$
 $b ::= (e == e) \mid (e <= e) \mid (e < e) \mid \text{false} \mid \text{true} \mid \neg b \mid b \vee b$



B12



$\text{data } Ta = \text{Nil} \mid \text{Tabs}[Tb] \mid \text{Tas}[Tc]$
 $\text{data } Tb = Tb[1] \text{ Jmt } Tb[2] \text{ Jmt } Tb[Tb] \text{ Ta } | Tba \text{ Ta}$
 $\text{data } Tc = Tc \text{ Jmt } Tcs[Tc] \mid Ta \text{ Ta}$

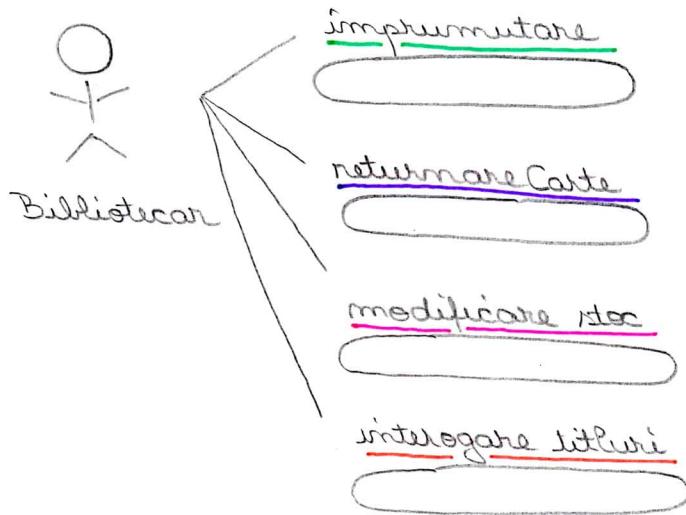


- (d) Se considera urmatorea specificare PRISM a unui sistem software modelat ca DTMC (Discrete-Time Markov Chain)

```
dtmc
module M1
    var1 : [0..3] init 0;
    []      var2<2          -> 0.25:(var1'=var2+1)+0.75:true;
    [interact1] var1=0 & var2<2 -> 0.55:(var1'=var2+1)+0.45:(var1'=var1+1);
    [interact2] var1>0        -> 0.43:(var1'=0)+0.57:(var1'=var1-1);
endmodule
module M2
    var2 : [0..3] init 0;
    [interact1] var2=0 & var1<2 -> 0.05:(var2'=var1+1)+0.95:(var2'=var2+1);
    [comm]   var1<2          -> 0.25:(var2'=var1+1) + 0.75:true;
    [interact2] var2>0        -> 0.33:(var2'=0)+0.33:(var2'=var2-1)+0.34:true;
endmodule
```

- i. Sa se proiecteze o structura de recompensa numita "str", care atribuie recompensa 1 oricarei stari in care $\text{var1}=0$ si tot recompensa 1 oricarei tranzitii care are eticheta de tranzitie "interact1" dintr-o stare in care variabila var1 are valoarea 0. [0.5p]
- ii. Sa se elaboreze o proprietate care calculeaza valoarea medie (engl. expected value) a recompensei cumulata (in raport cu "str") pana cand sistemul ajunge eventual intr-o stare in care $\text{var1}=2$. [0.25p]

1. Specificare (Model cazuri de utilizare).

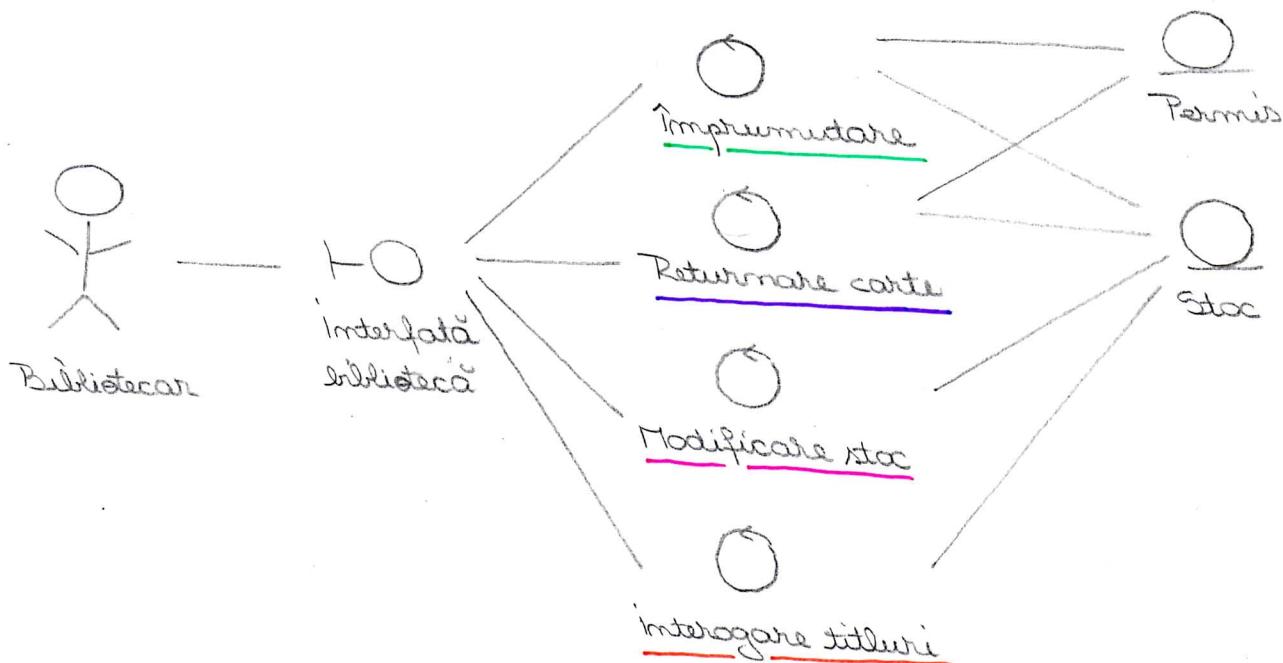


Descriere în limbaj natural :

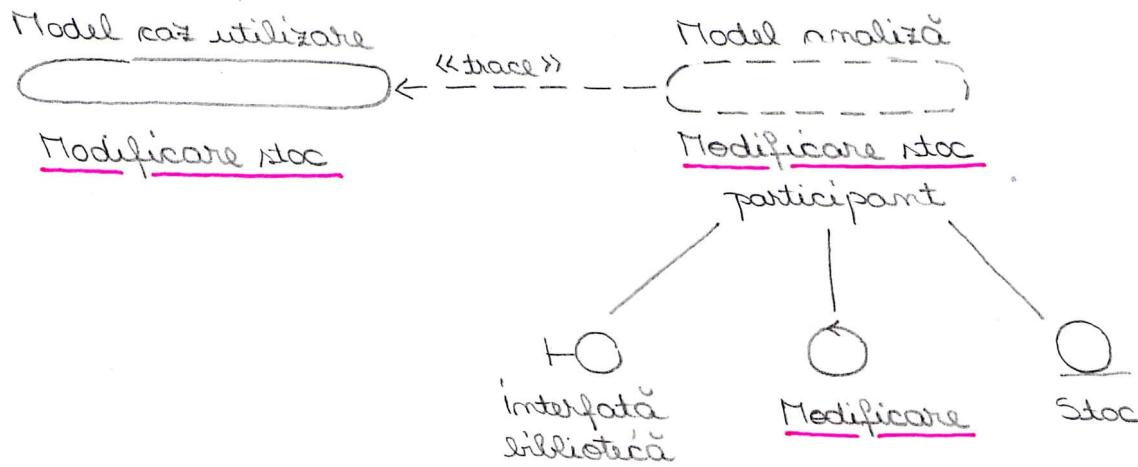
- pentru modificare stoc : 1. Verificare titlu dat
2. Modifica stocul dacă titlul există.
- împrumutare : 1. Verifică stoc pentru titlu cerut.
2. Bibliotecarul verifică permisul și cările disponibile
3. Bibliotecarul scade stocul și înregistrează împrumutul.

2. Analiza - refinează cerințele sistem și prezintă o primă iteratie în procesul de dezvoltare.

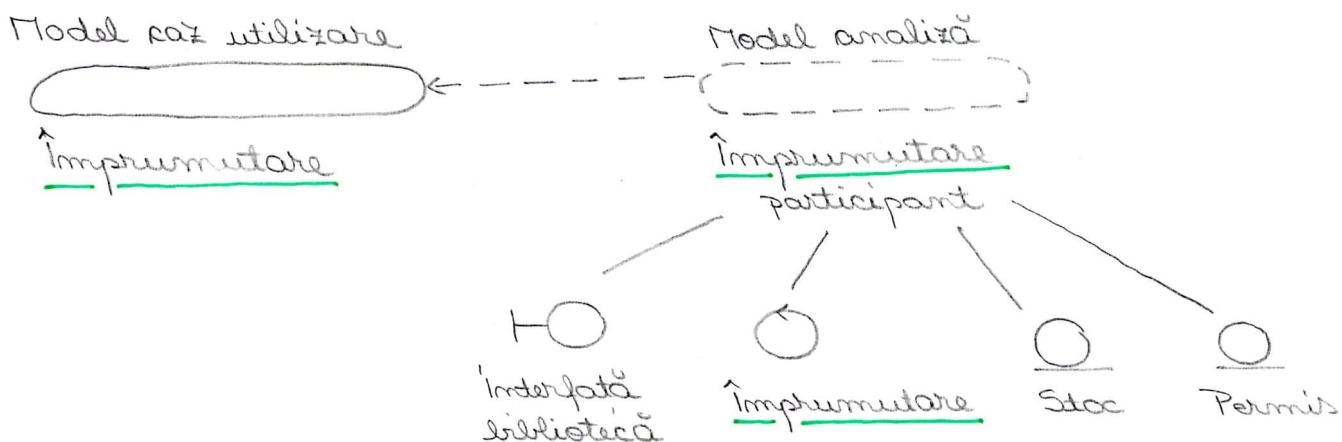
- Structura modelului de analiză



• Clasele modelului de analiză



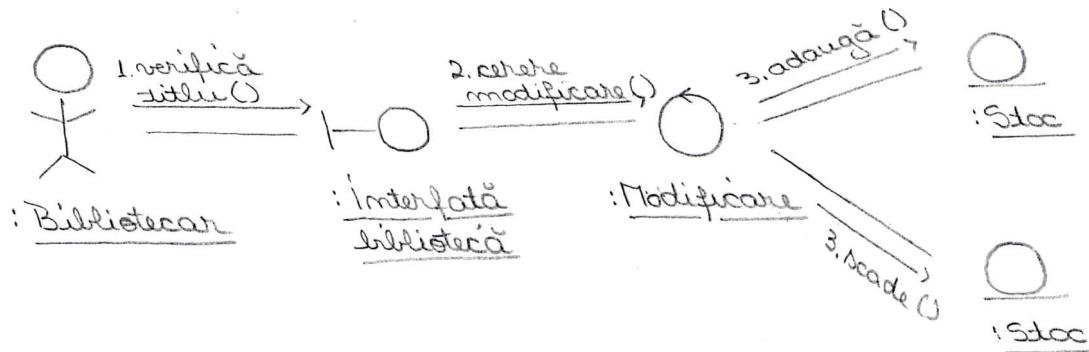
Clasele modelului de analiză interrogare titluri sunt la fel cu cele de mai sus, singura diferență: în loc de Modificare stoc, scriem Interrogare titluri. În loc de Modificare, scriem Interrogare.



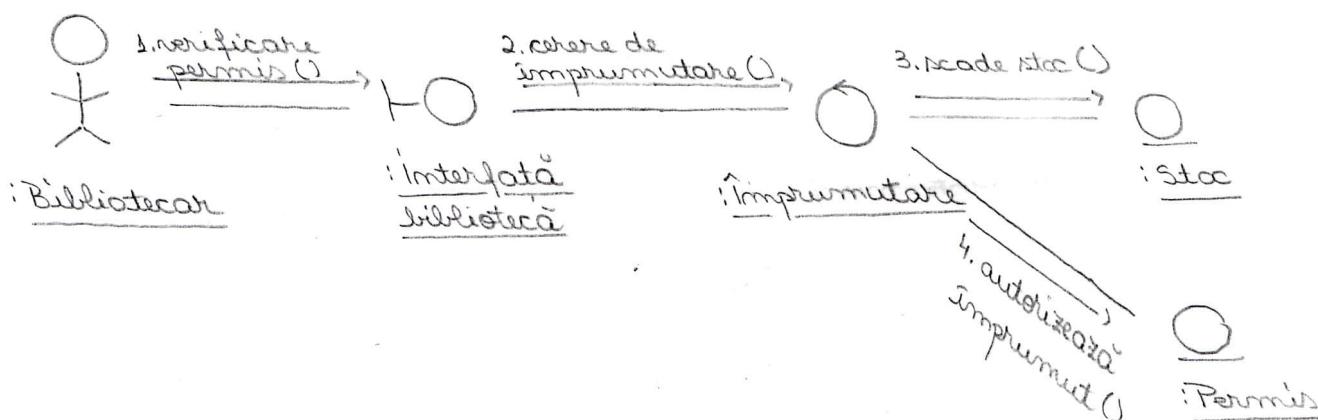
Clasele modelului de analiză returnare carte sunt la fel cu cele de mai sus, diferența este: în loc de Împrumutare, avem Returnare carte. În loc de Împrumutare, avem Returnare.

Structura de analiză trebuie să fie completată cu diagrame de interacțiune care arată aspectele dinamice ale realizării cazurilor de utilizare.

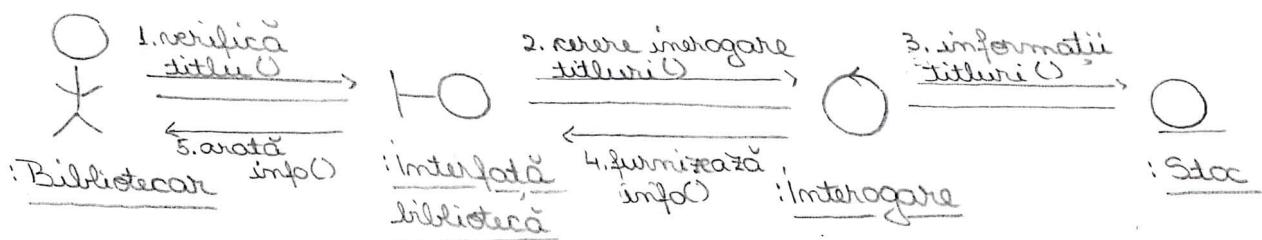
• Diagramme de comunicare pentru realizarea în analiză a flexibilității de bază pentru cazul de utilizare: - modificare stoc



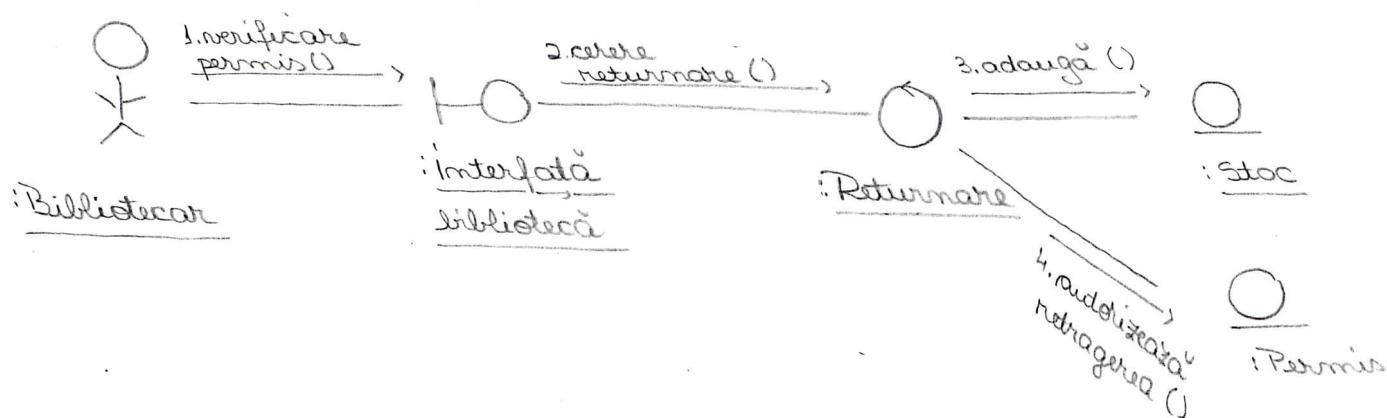
- împrumutare



- interrogare titluri



- returnare carte



3. Proiectare / Design

Modelul de proiectare trebuie să reprezinte o schită pentru reprezentare.

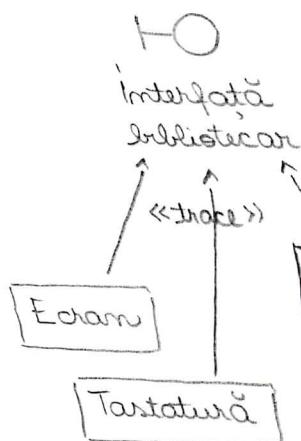
Principala intrare e modelul de analiză. Modelul de design e adaptat împotriva mediului de adaptare.

O clasă de analiză e o abstractizare a unui număr de clase de design aşa cum se vede în figurile de mai jos.

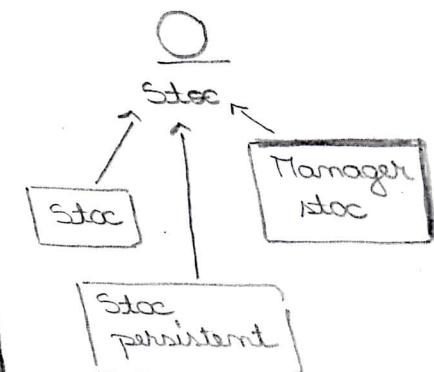
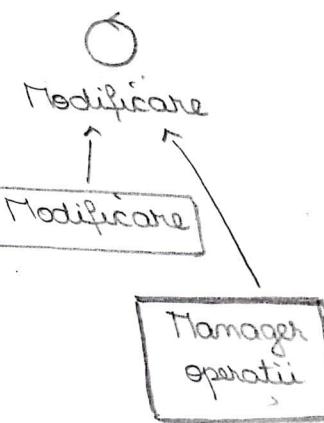
Toate clasele de analiză duc către clase de design coresp. prin hafizare.

Modificare stoc

Model analiză



Model design

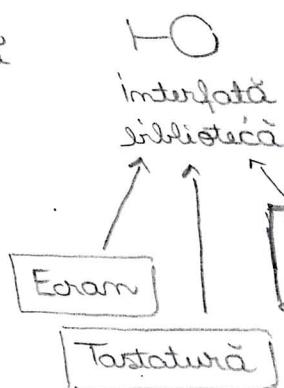


Clasele Manager bibliotecă, Manager operatii, Manager stoc sunt clase active proiectate să organizeze activitatea celorlalte clase.

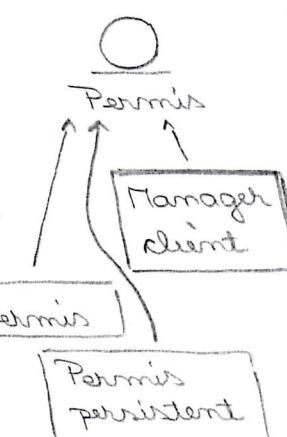
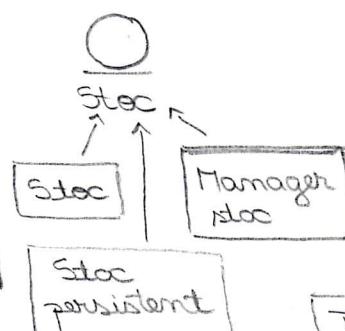
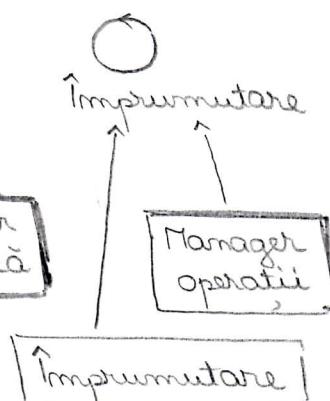
În cazul unei implementări distribuite, instanțele claselor active sunt procese sau fișe de execuție. Ele se reprezintă ca un dreptunghi cu linie îngroșată.

Imprumutare

Model analiză

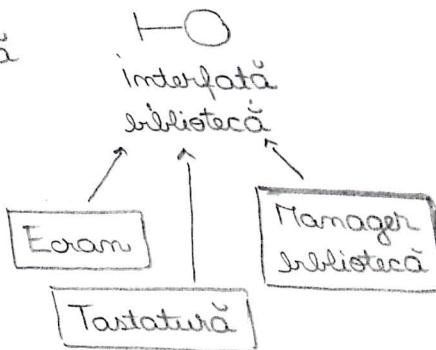


Model design

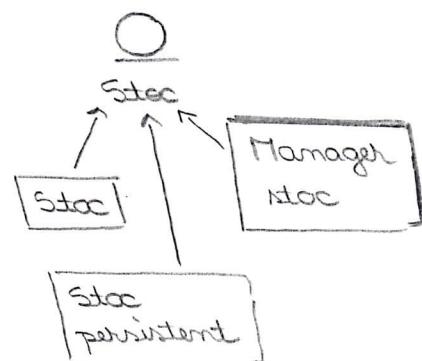
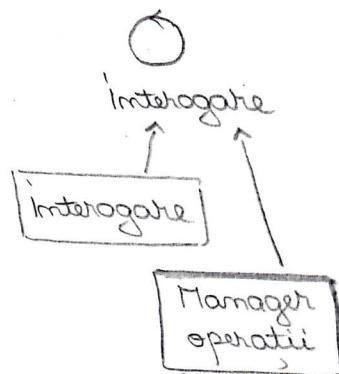


Interogare titluri

Model analiză

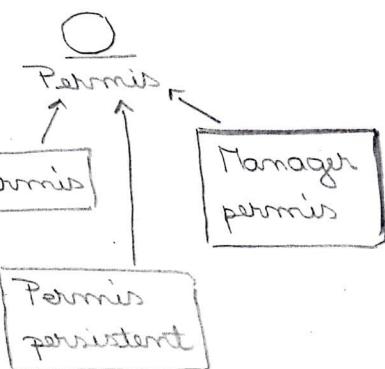
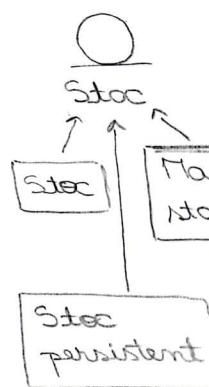
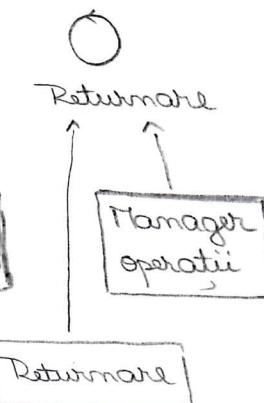
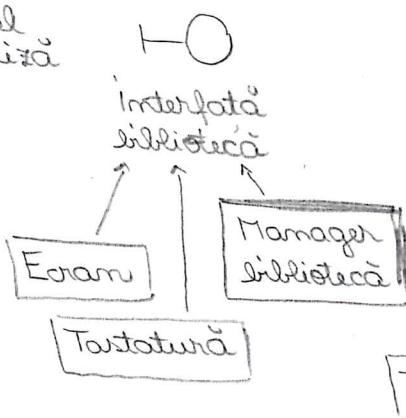


Model design



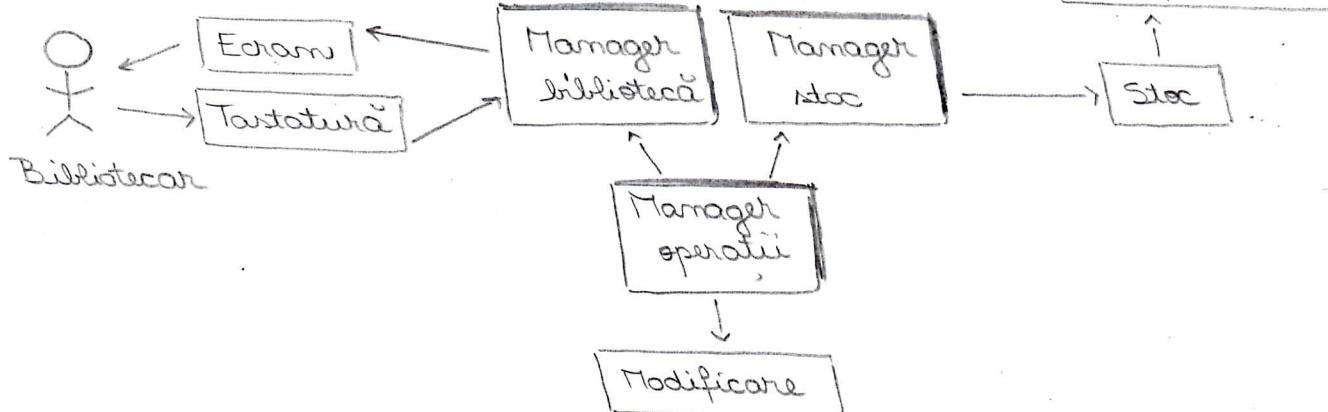
Returnare carte

Model analiză



- Diagrama de clase - prezintă relațiile dintre clasele de design

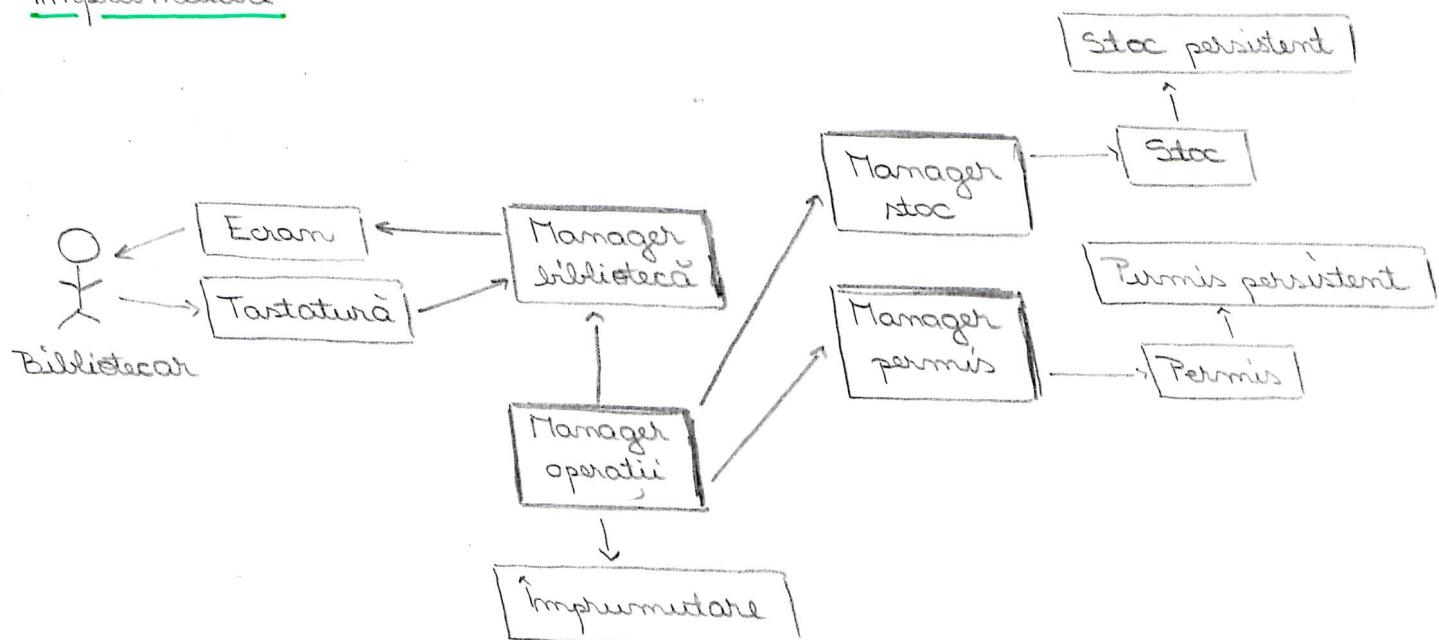
Modificare stoc



Interogare titluri

La fel ca la Modificare stoc, singura diferență: în loc de **Modificare** avem **Interogare**.

Împrumutare

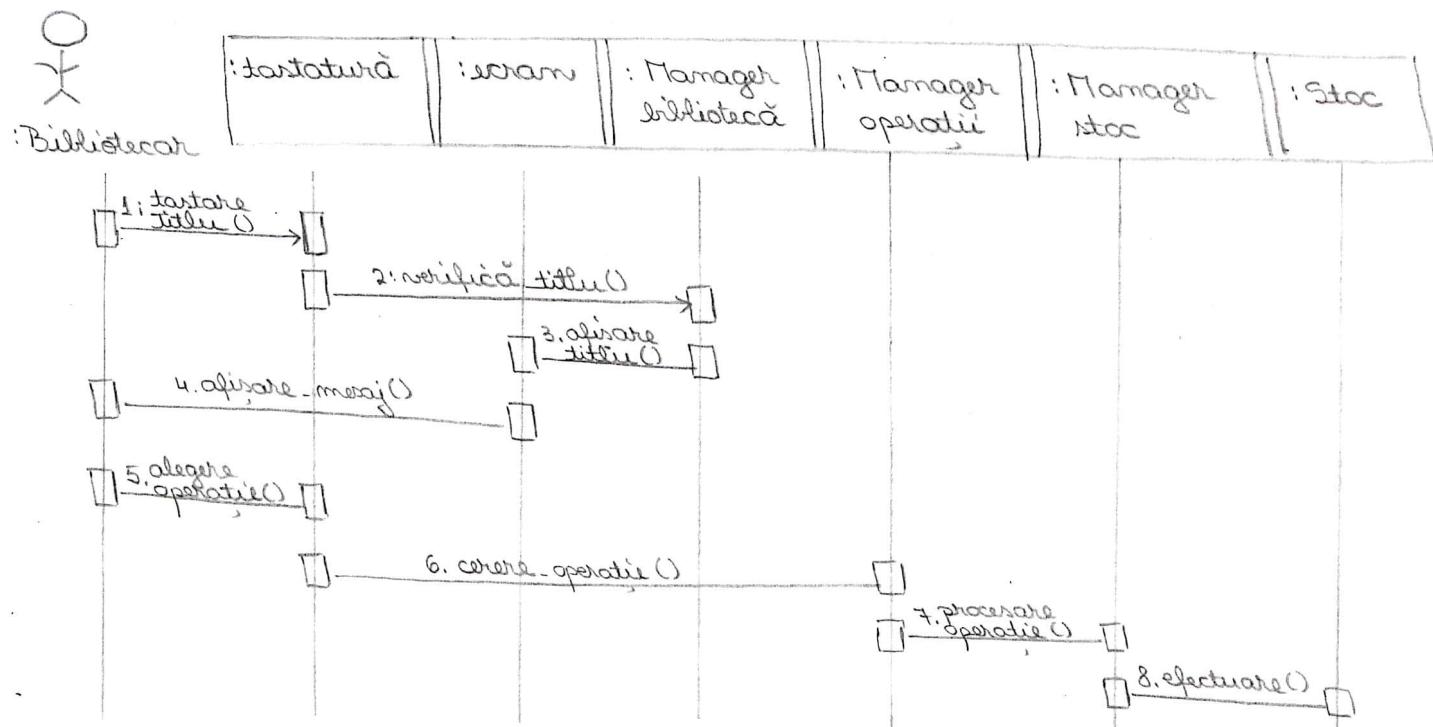


Returnare carte

La fel ca la Împrumutare, singura diferență: în loc de **Împrumutare**, avem **Returnare**.

• Diagrama de secențiere pentru fluxul de bază:

Pentru **Modificare stoc** și pentru **Interoagere titlu**



Pentru Împrumutare și pentru Returnare carte

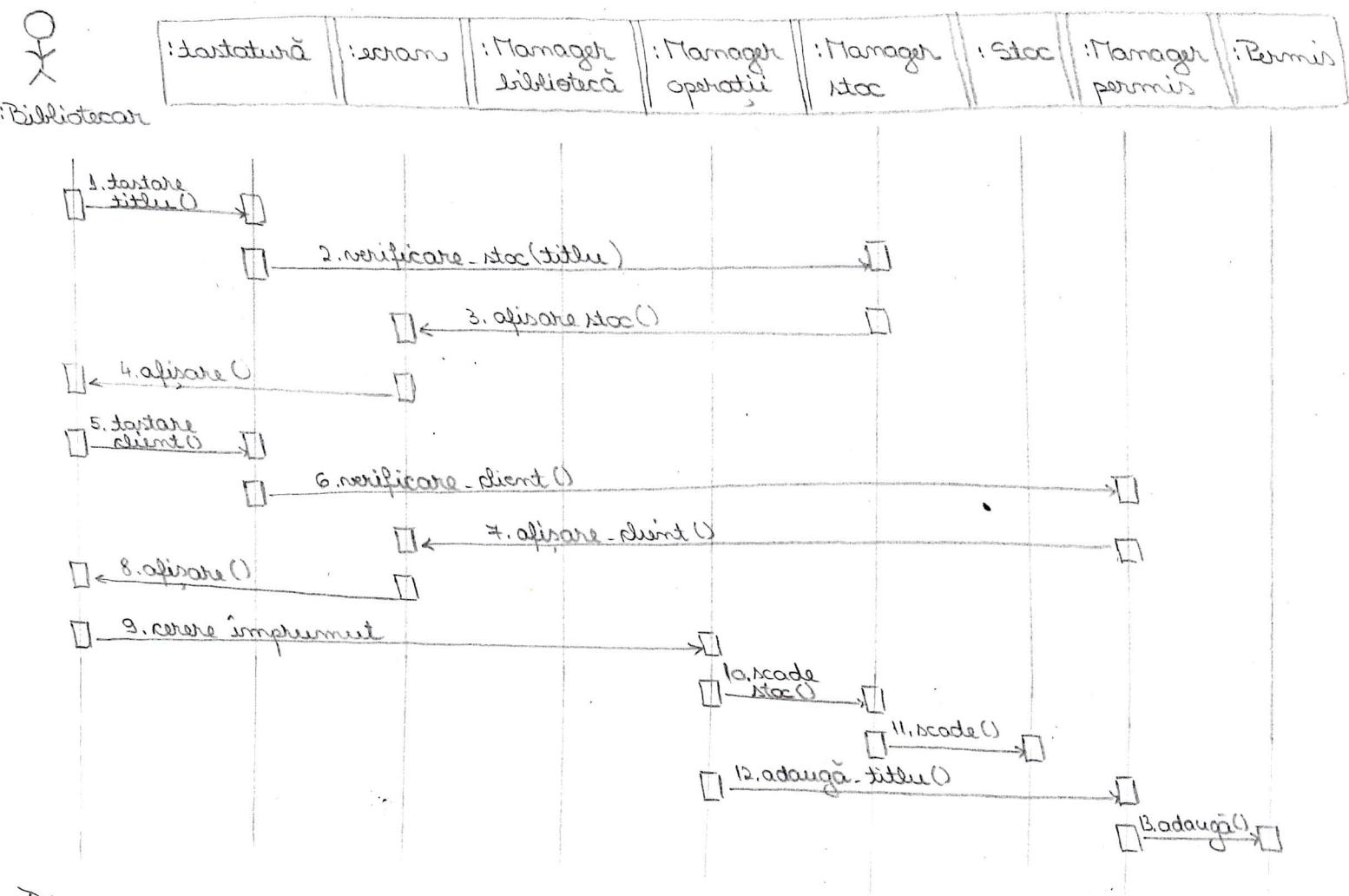
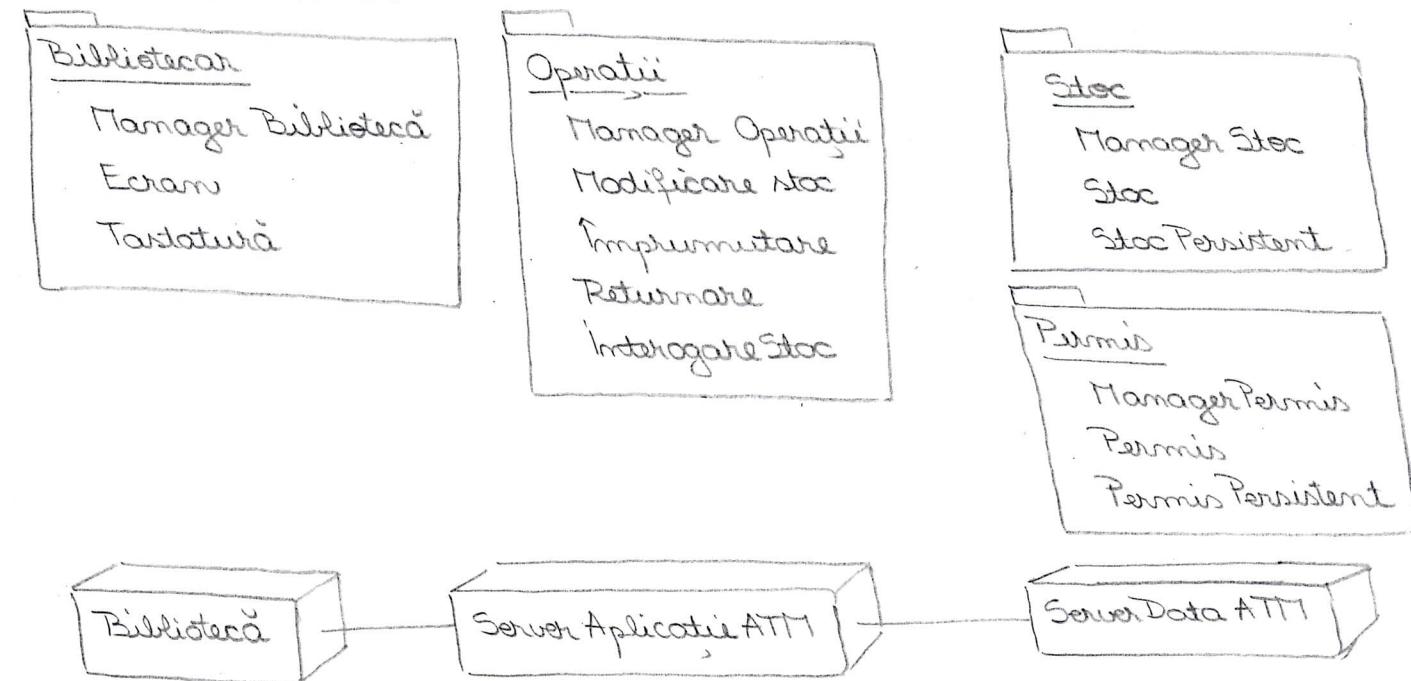
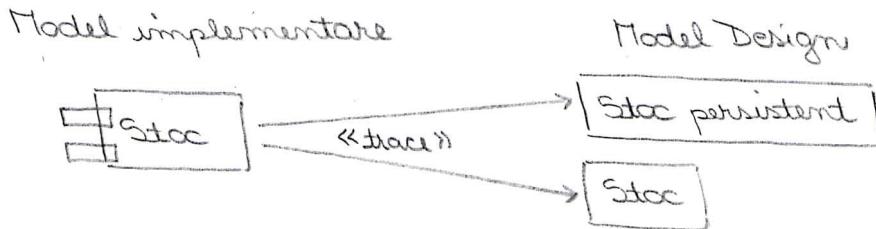


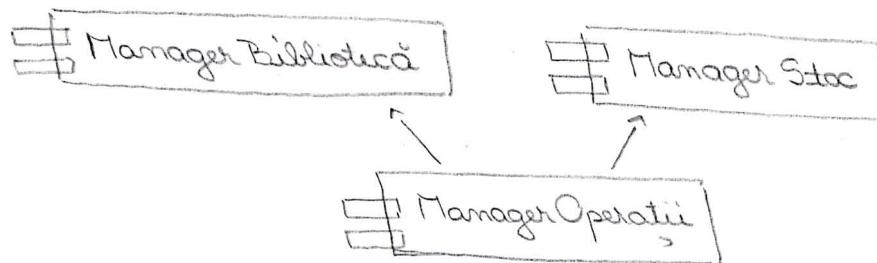
Diagrama de pachete



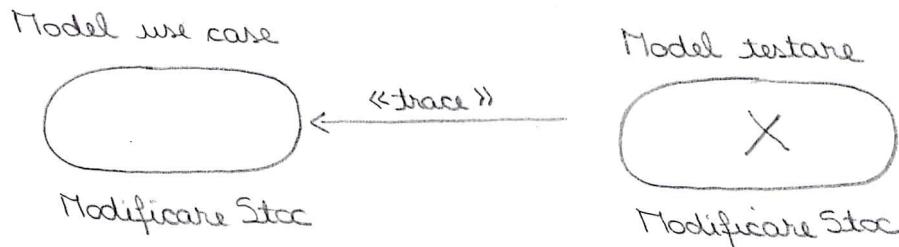
4. Implementare



Dependențe de compilare



5. Testare



Cazuri de testare: Modificare Stoc

Intrare: Titlul "Abcd" există în bibliotecă, cu stocul de 32 buc.

Bibliotecarul cere creșterea stocului pentru titlul "Abcd" cu 5 buc.

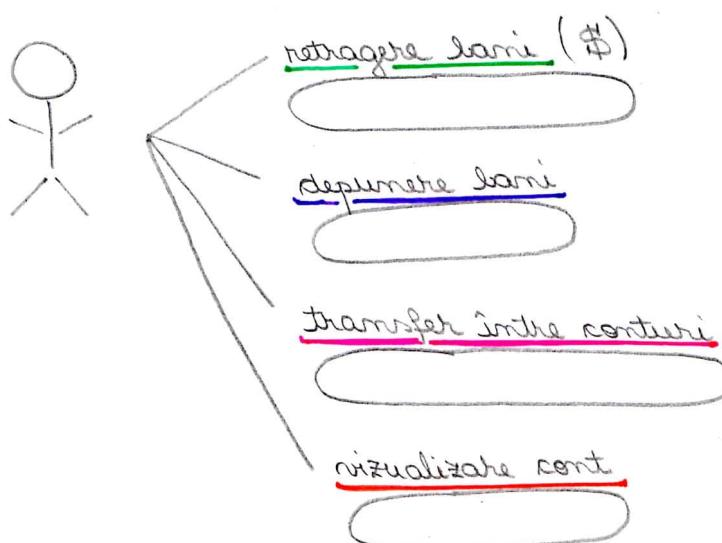
Rezultat: Stocul pentru carte "Abcd" este de 37 buc.

2. Aplicați metodologia de dezvoltare ghidată de căzurile de utilizare din Procesul Unificat în construirea de modele UML pentru un sistem automat bancar, ce permite utilizatorului (1) să retragă bani din cont, (2) să depună bani în cont, (3) să transfere bani între conturi și (4) să vizualizeze sume existente în cont.

Să se dezvolte modelele UML complete (specificare, analiză, proiectare, implementare, testare) pentru serviciul de retragere bani,

depunere bani
vizualizare sumă
transfer între conturi

1. Specificare (Model căzuri de utilizare).



Se poate răsări o descriere în limbaj natural fiecărui caș de utilizare!

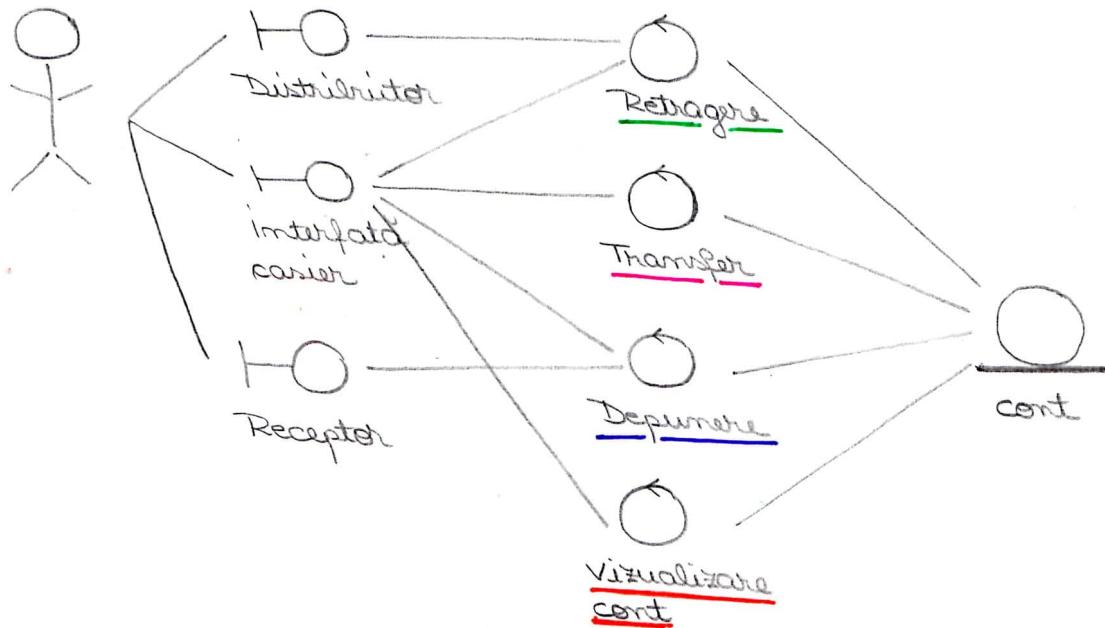
ex. pentru transfer bancar: 1. Clientul se identifică

2. Clientul alege cont surșă, contul destinatie, suma de bani
3. Sistemul transferă bani

2. Analiza - nașterează cerințele sistem și reprezintă o primă iteratie în procesul de dezvoltare

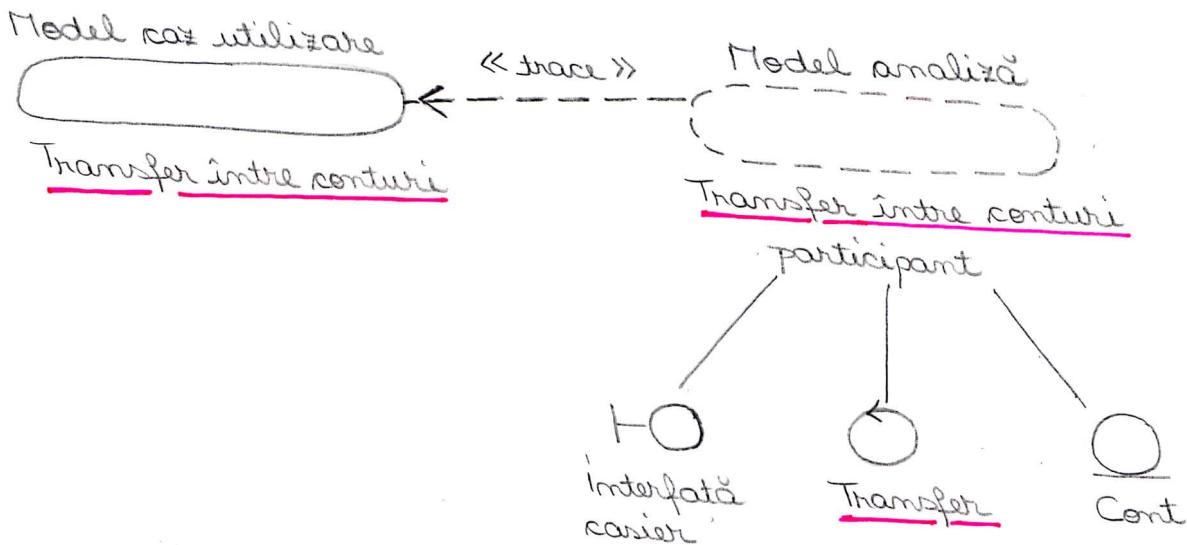
- Structura modelului de analiză

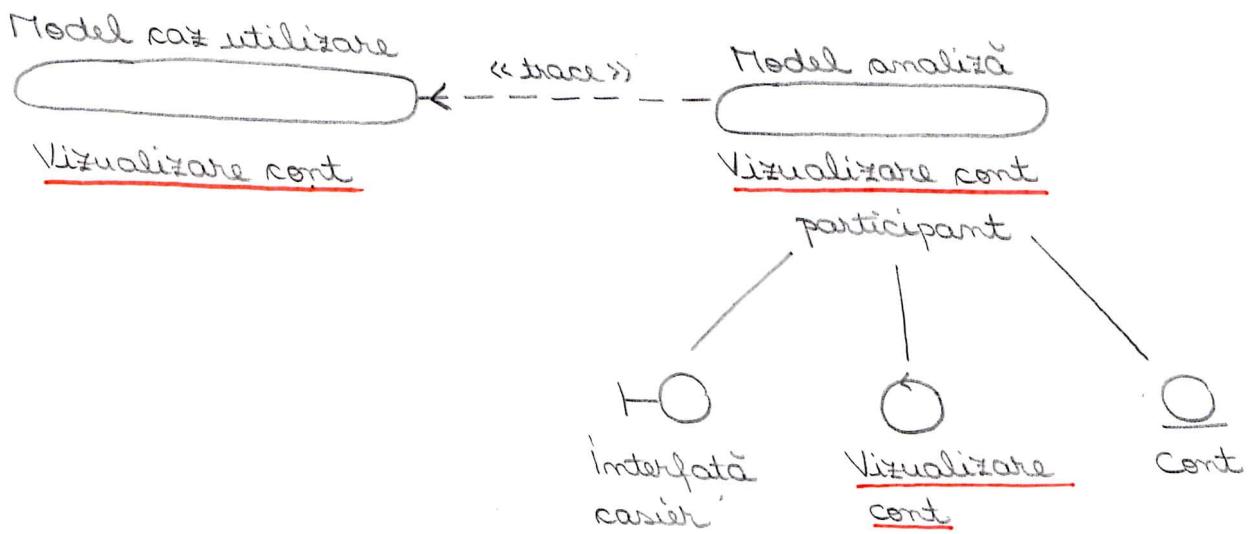
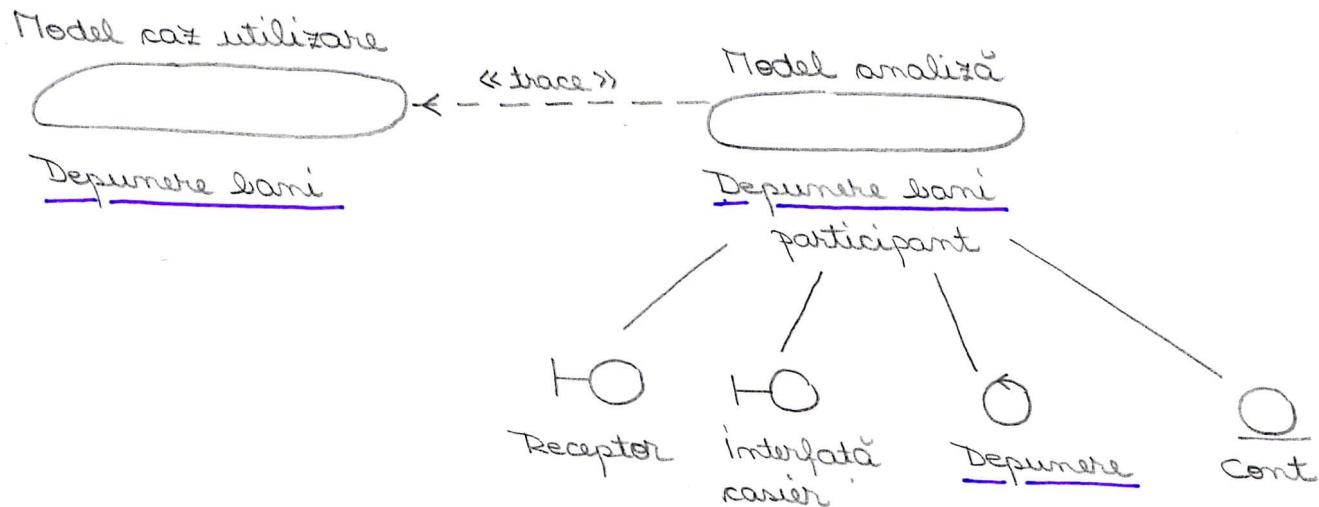
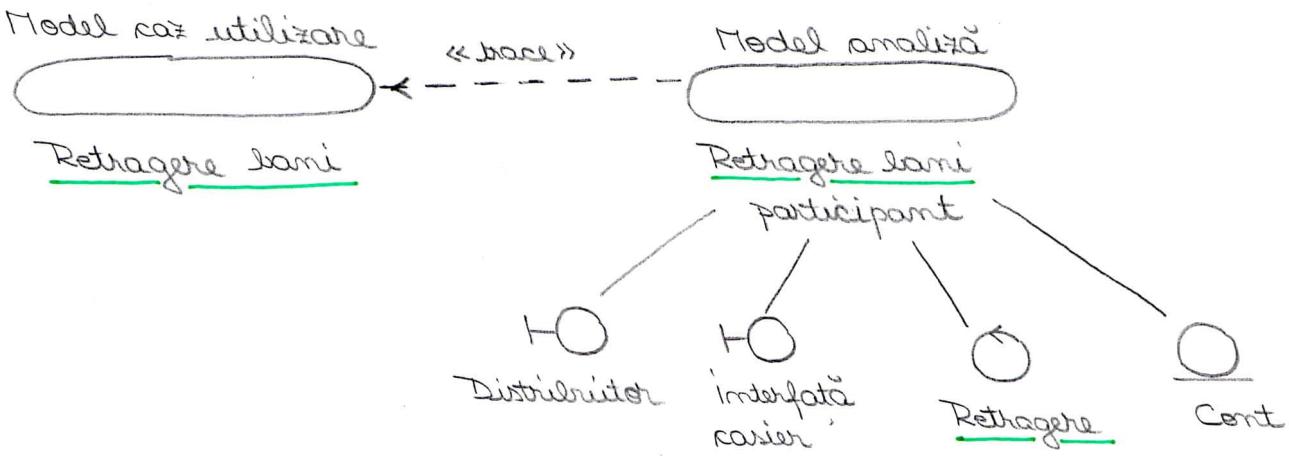
Există o clasă de control pentru fiecare rază de utilizare



Obs! Fiecare clasă de utilizare poate participa (și poate juca roluri diferite) în mai multe realizări de căsuțe de utilizare.
 ex: clasa cont participă la realizarea tuturor căsuțelor de utilizare
 Fiecare rază de utilizare e realizat printr-o coloană de clase analiză.

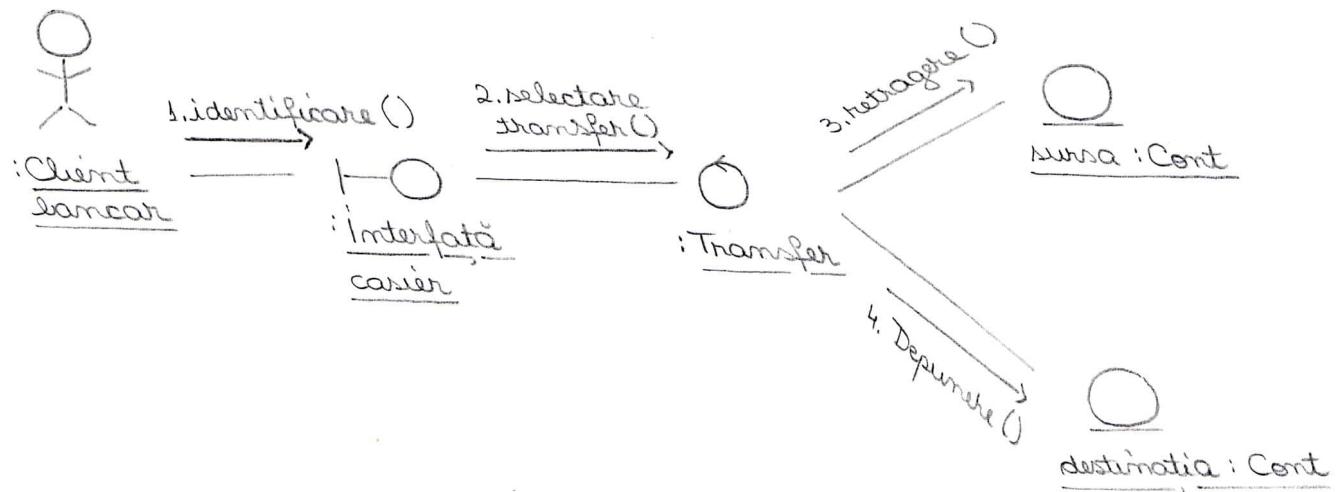
- Clasele modelului de analiză



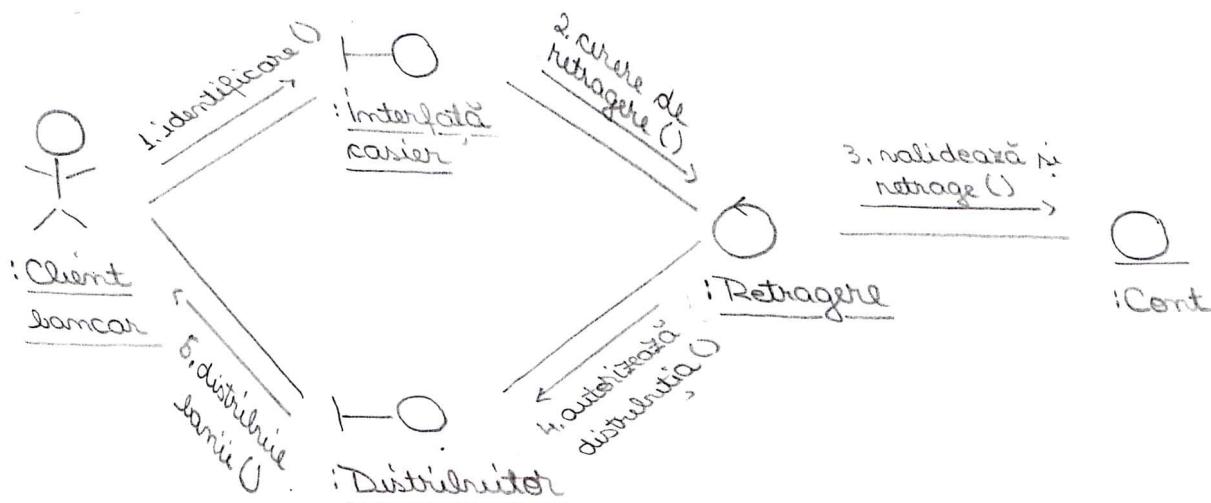


Structura de analiză trebuie să fie completată cu diagrame de interacțiune care arată aspectele dinamice ale realizării cazurilor de utilizare.

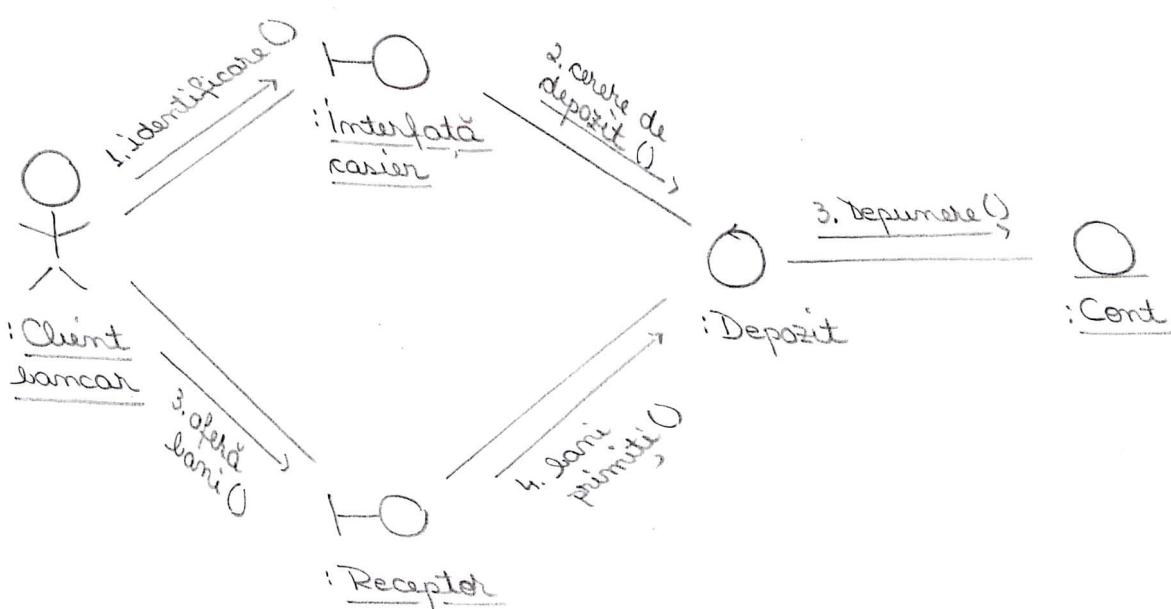
- Diagrama de comunicare pentru realizarea în analiză a fluxului de bani
pentru cazul de utilizare - transfer între conturi



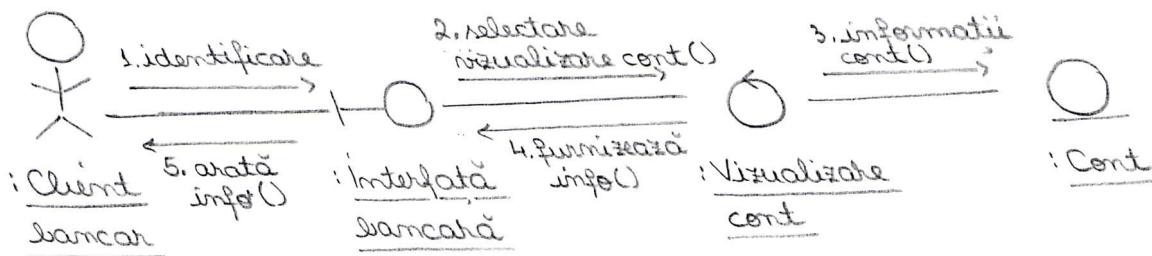
- retragere bani



- depunere bani



- vizualizare cont



Fiecare clasă trebuie să îndeplinească rolurile de colaborare.

Prin compilarea acestor roluri pentru toate cazurile de utilizare se obține o specificare a responsabilităților și atributelor fiecărei clase de utilizare.

3. Proiectare / Design

Modelul de proiectare trebuie să reprezinte o schită pentru reprezentare.

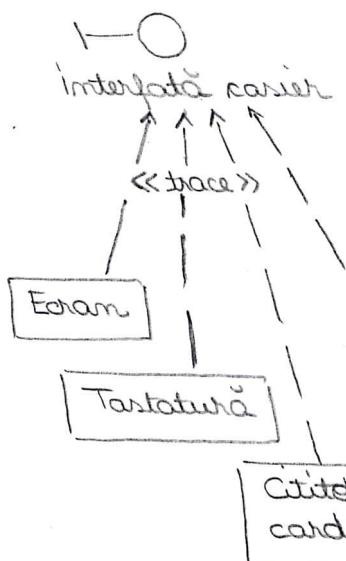
Principala intrare e modelul de analiză. Modelul de design e adaptat și la mediul de adaptare.

O clasă de analiză e o abstractizare a unei mulțimi de clase de design așa cum se vede în figurile de mai jos.

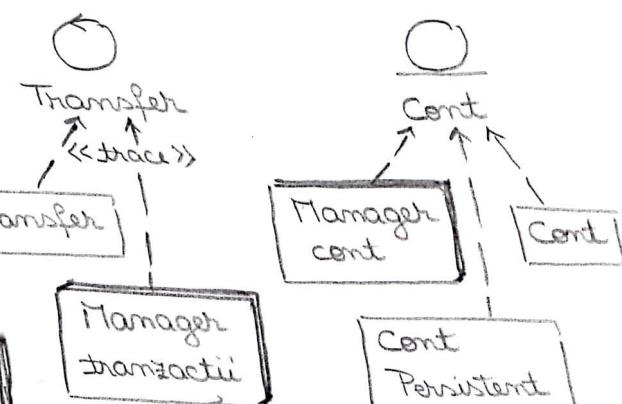
Toate clasele de analiză dă măște la clase de design coresp. primă rafinare.

Transfer între conțuri

Model analiză



Model design

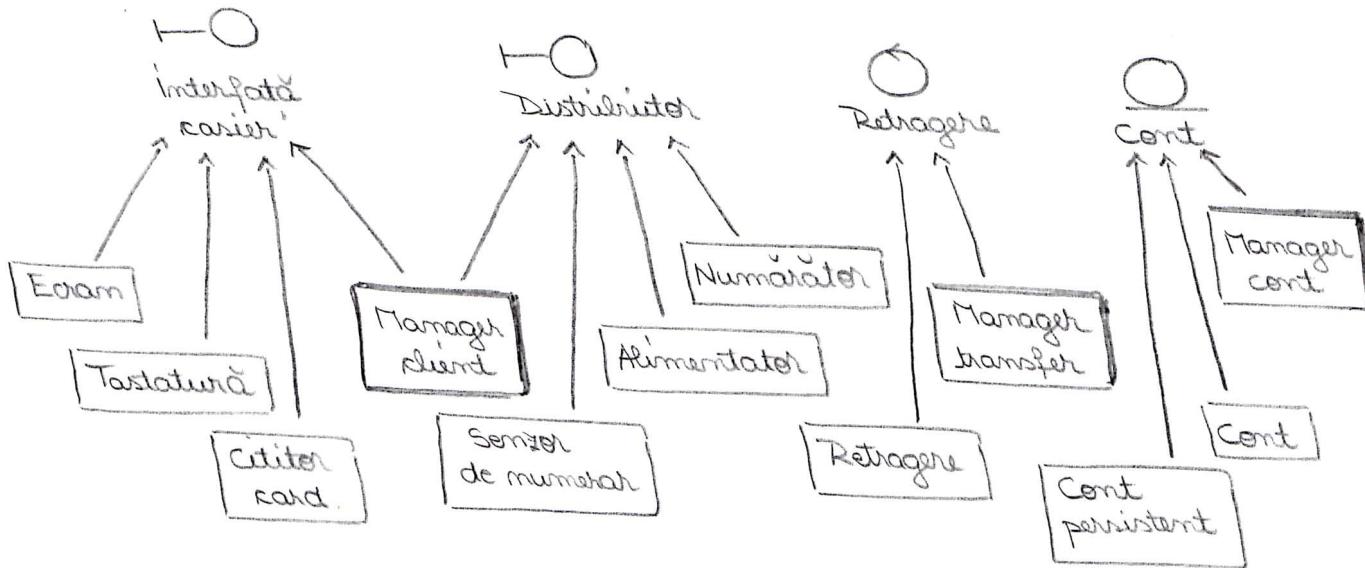


Clasele Manager client, Manager tranzacții, Manager cont sunt clase active proiectate să organizeze activitatea celorlalte clase.

În cazul unei implementări distribuite, instancele claselor active sunt procese sau fizice de execuție. Ele ne reprezintă ca un dreptunghi cu linie îngroșată.

Retragere bani

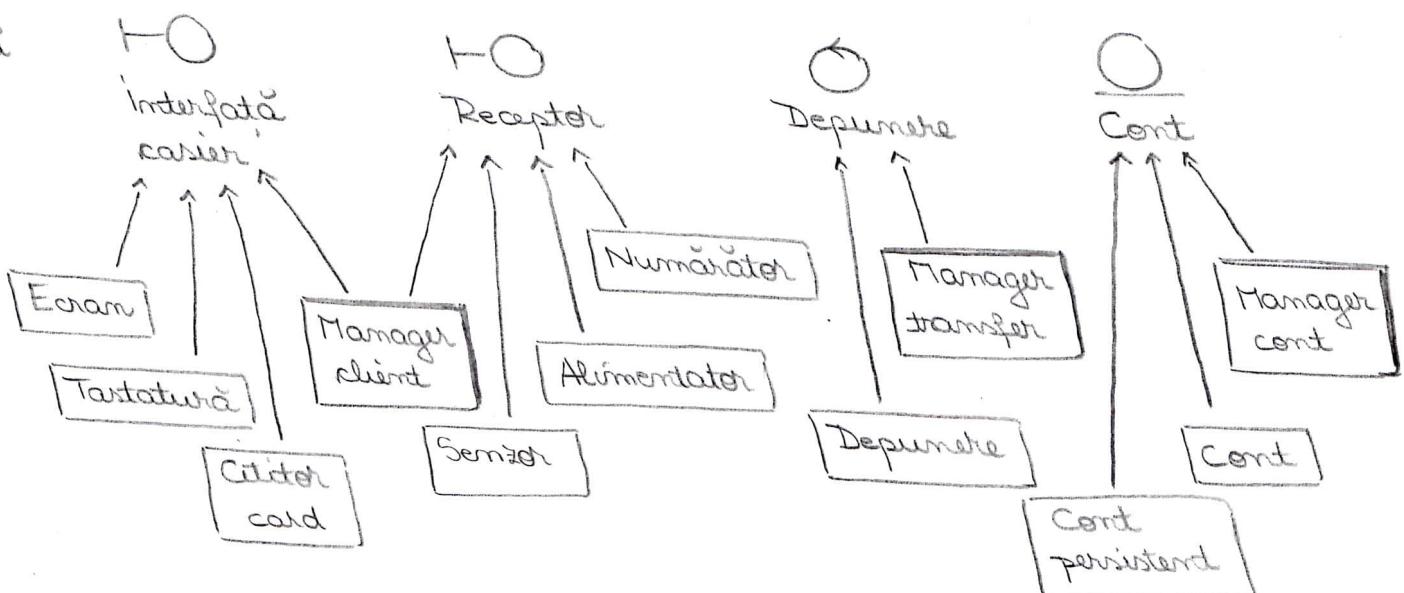
Model analiză



Model design

Depunere bani

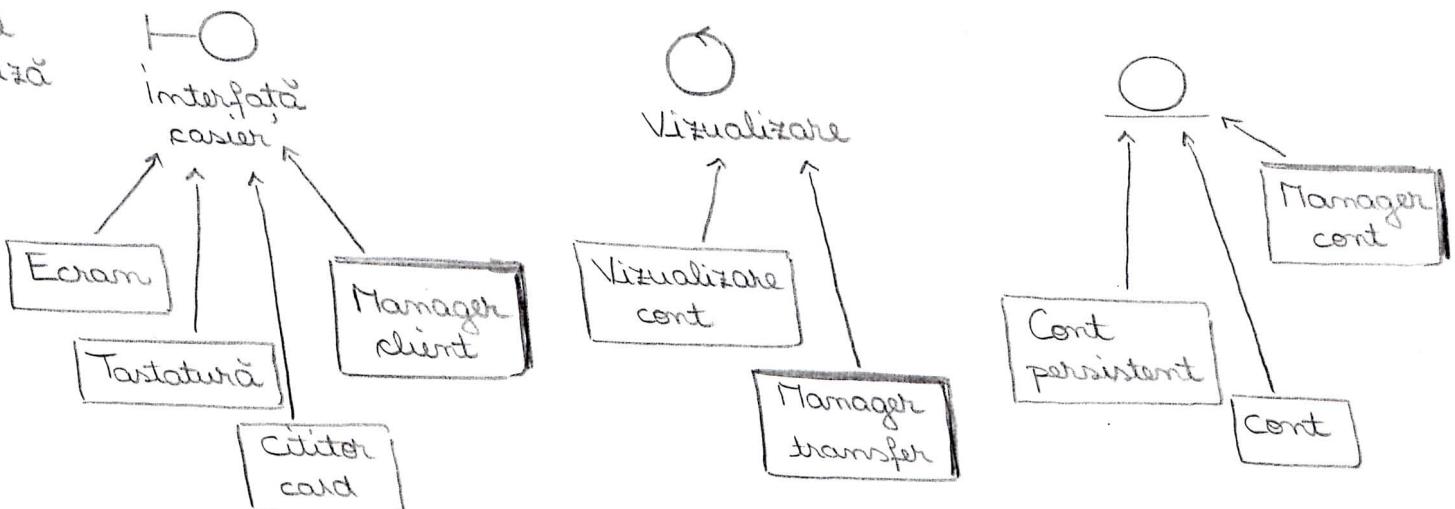
Model analiză



Model design

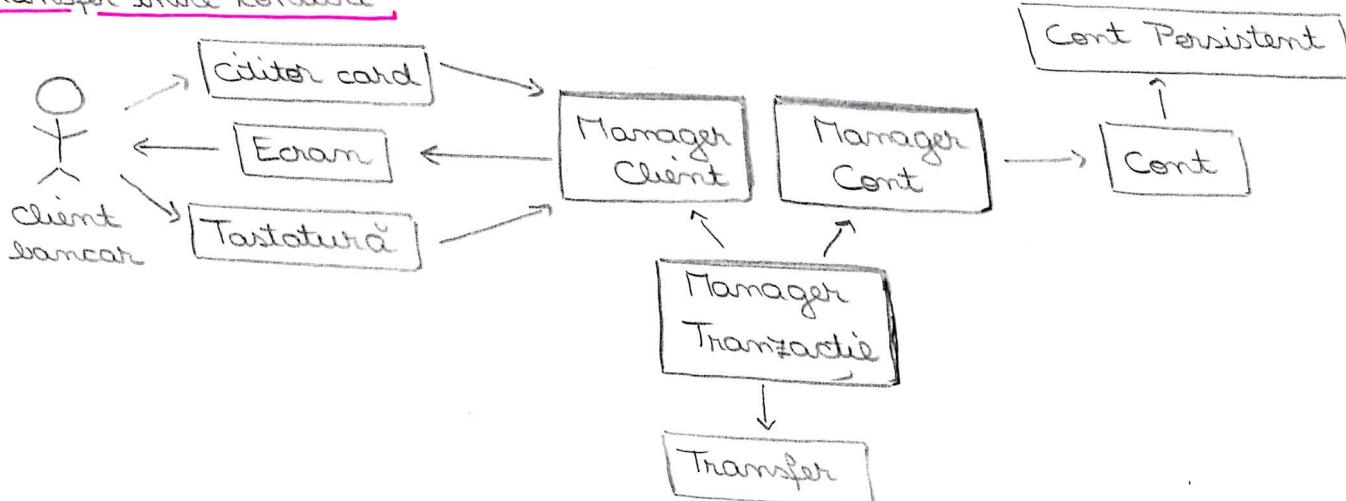
Vizualizare cont

Model analiză



- Diagrama de clase - prezintă relațiile dintre clasele de design ce participă în realizarea cazului de utilizare transfer între conturi

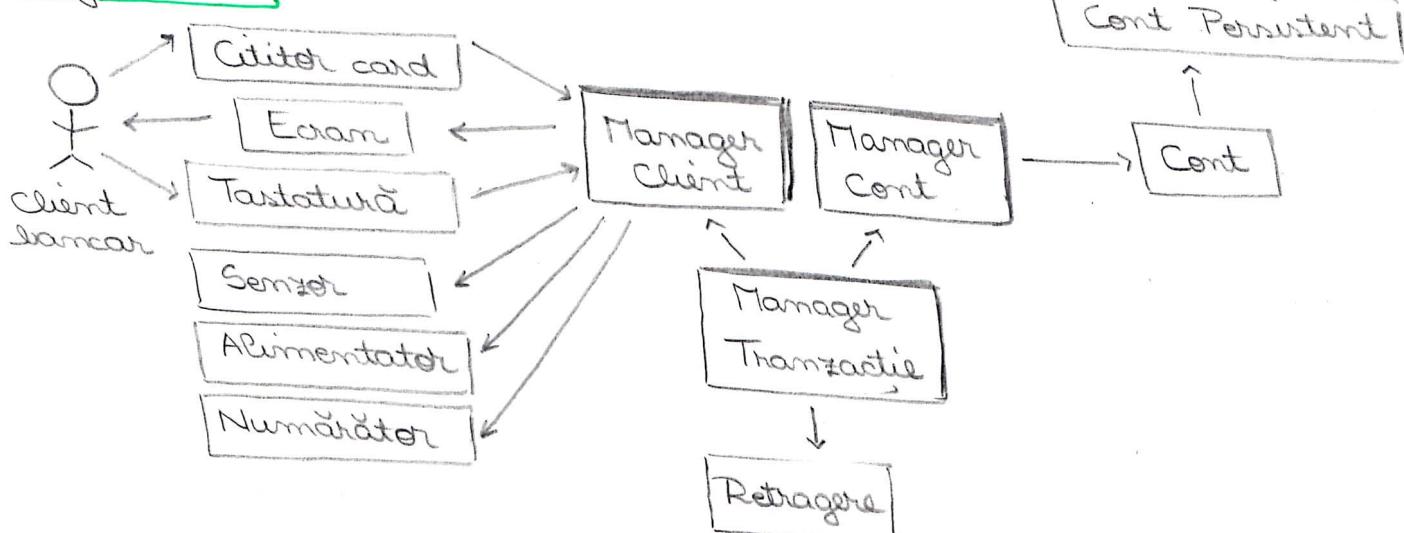
Transfer între conturi



Vizualizare cont

La fel ca la Transfer, singura diferență în loc de [Transfer] avem Vizualizare cont

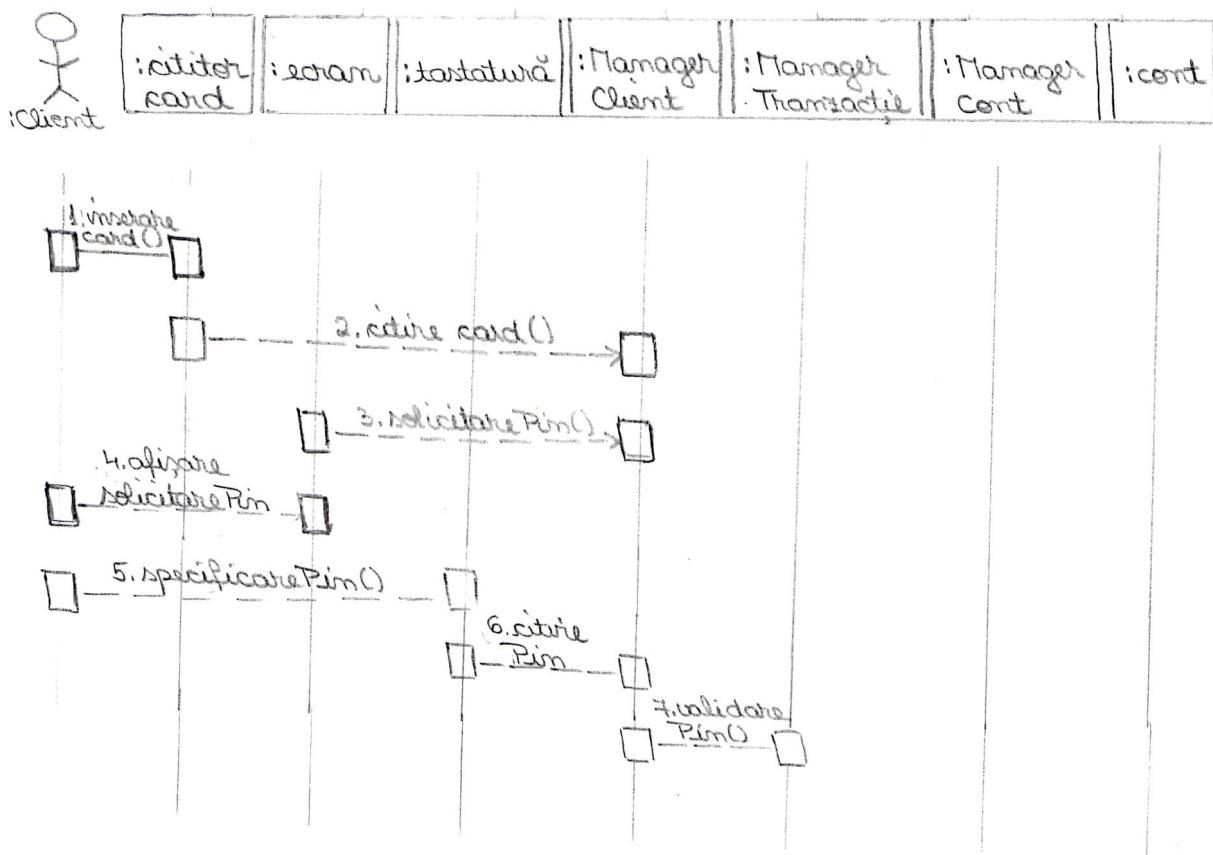
Retragere bani



Depunere bani

La fel ca la Retragere bani, singura diferență: în loc de [Retragere] avem Depozit].

- Diagramă de secvențiere pentru fluxul de bază
(la fel pentru toate 4).



Este util să grupăm clasele care au ceea ce în comun.
Considerăm o soluție de descompunere bazată pe 3 pachete: Client, transacție și cont. Există o clasă în fiecare pachet. Proiectantul decide să desfășoare componentele care formează subsistemele pe 3 moduri distincte într-o arhitectură "three-tier".

