UML (Unified Modeling Language)

Limbaj unificat de modelare

CUPRINS

1. Introducere in UML

UML reprezinta o notatie ce se foloseste cu precadere in analiza si proiectarea OO. Folosirea de ***modele*** poate înlesni abordarea problemelor complexe, facilitând înţelegerea lor. Un model este o simplificare a unui anumit sistem, care permite analiza unora dintre proprietatile acestuia, retine caracteristicile necesare. Folosirea de modele este comuna aproape tuturor domeniilor ştiinţei. Am putea da ca exemple formalismul matematic, reprezentarile din fizica, recunoasterea formelor, etc.
Pe un principiu asemănător se bazează conceptul de **UML** ("Unified Modeling Language"). Limbajul unificat de modelare UML este un limbaj pentru *specificarea, vizualizarea, construirea si documentarea* elementelor sistemelor software. Poate fi folosit si pentru alte sisteme, cum ar fi cel de modelare al afacerilor. UML reprezinta o colectie de practici ingineresti optime, care au fost încununate de succes în modelarea sistemelor mari si complexe. UML se bazeaza pe notatii si se foloseste cu precadere in analiza si proiectarea orientata pe obiecte, (OOA si OOD).
Figura urmatoare prezinta modul în care a evoluat de-a lungul timpului acest concept, detalii se pot obtine de la adresa www.omg.org.

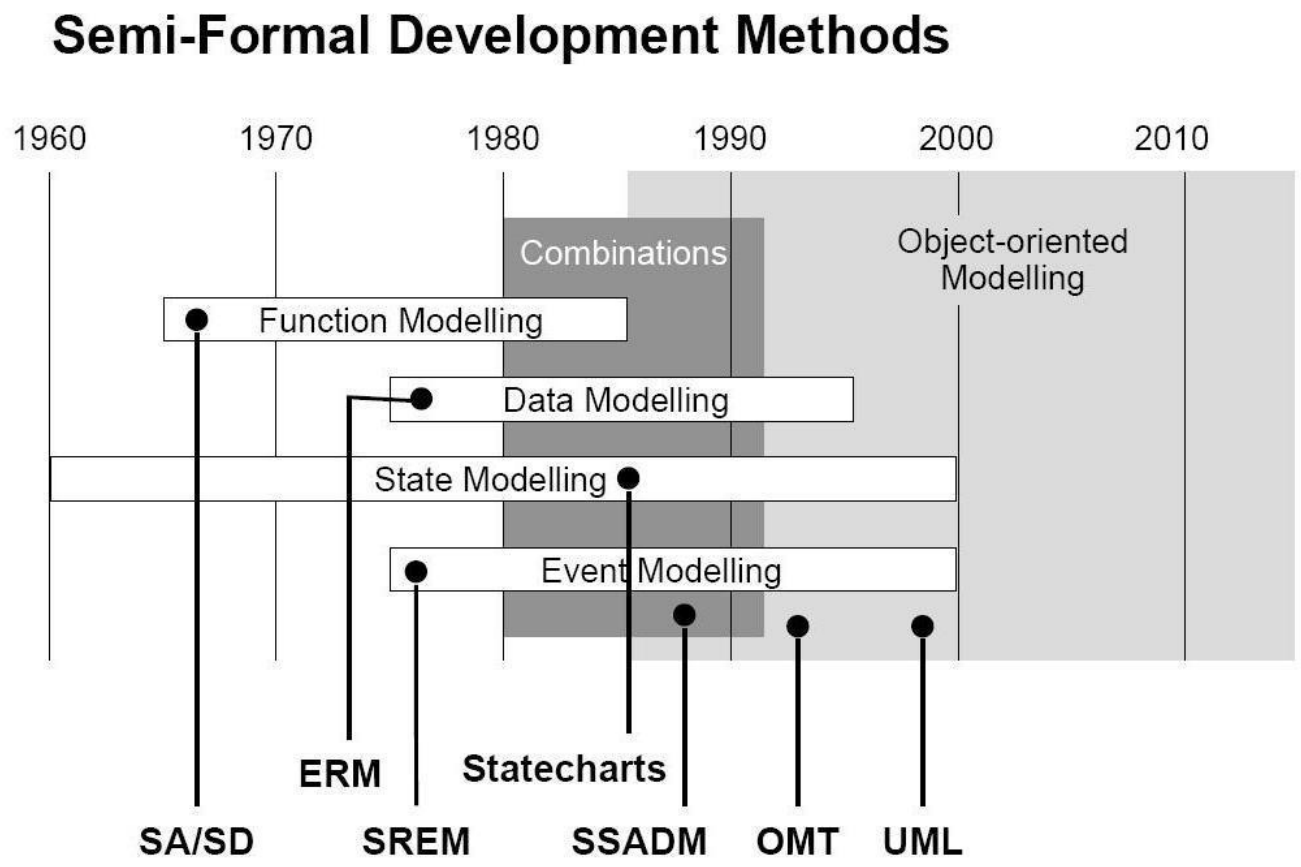


Figura 1. Dezvoltarea metodelor semi-formale

Între 1989 si 1994 erau folosite mai mult de 50 de limbaje de modelare software, fiecare cu propriile notatii. Utilizatorii doreau un limbaj standardizat, usor de utilizat in modelare.
Pe la mijlocul anilor ’90 trei metode s-au dovedit mai eficiente in acest proces de modelare:
-diagramele Booch, potrivite mai ales pentru *proiectare si implementare*, cu dezavantajul unor notatii complicate; Ele se bazeaza pe OOD (Object Oriented Design).
-OMT (Object Modeling Technique), potrivita pentru *analiza, si la sistemele informationale cu multe date*, dezvoltat de James Rumbaugh
-OOSE (Object Oriented Software Engineering) metoda dezvoltata de Ivar Jacobson care a propus asa-numitele *use cases*, care ajutau la întelegerea *comportamentului* întregului sistem.

În 1994 Jim Rumbaugh, creatorul OMT, a parasit General Electric, alaturându-se lui Grady Booch la Rational Corp. În 1995 Ivar Jacobson, creatorul OOSE, a venit la Rational, iar ideile lui (în special conceptul de use cases, cazuri de utilizare) au fost adaugate „Metodei unificate”; metoda rezultanta a fost numita „Limbajul de modelare unificata” – UML.
În 1996 a dus la formarea de catre Rational a consortiului partenerilor UML din care faceau parte si companii mari precum Hewlett-Packard, Microsoft si Oracle.
În ianuarie 1997 UML 1.0 a fost propus spre standardizare în cadrul OMG (Object Management Group). În noiembrie 1997 Versiunea UML 1.1 a fost adoptata ca si standard de catre OMG, iar în martie 2003 a fost publicata versiunea 1.5. În octombrie 2004 a fost introdusa versiunea 2.0. Figura următoare prezintă etapele aparitiei UML 1.0.

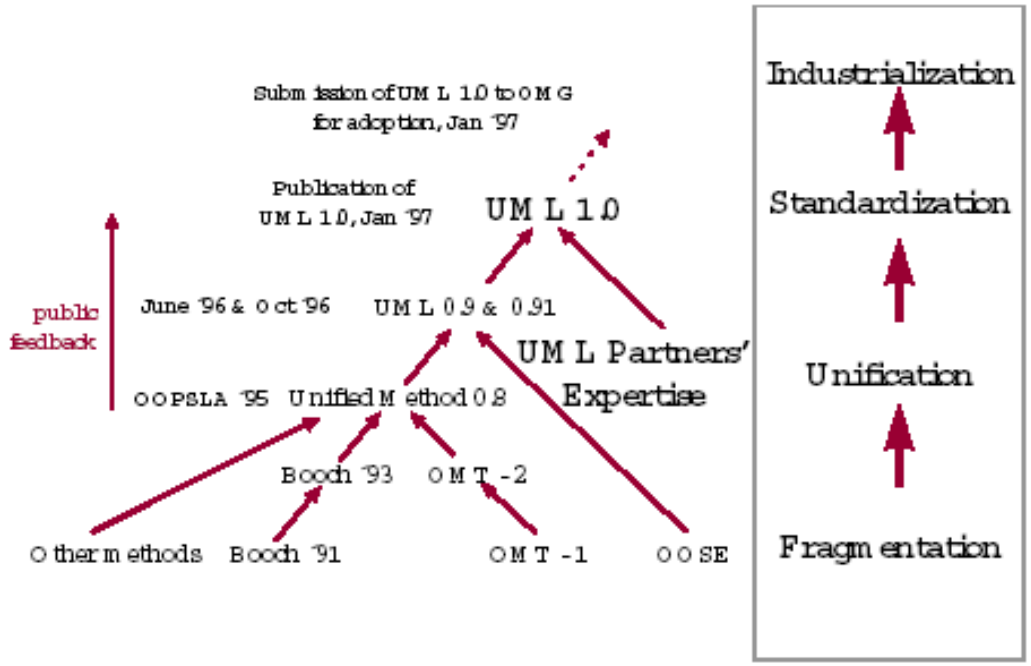


Figura 2. Etapele aparitiei UML 1.0

Limbajul UML s-a format având la bază cele trei metode amintite, la care se adaugă contribuţii notabile în diverse faze ale etapelor de analiză şi proiectare cum ar fi: clasificare Odell, hărţi de stări David Harel, ciclul de viaţă al obiectelor Shlaer-Mellor, şabloane de proiectare Gamma, etc.

Deci, limbajul de modelare modificat ofera arhitecturi de sisteme ce functioneaza pentru analiza si proiectarea obiectelor cu un limbaj corespunzator pentru specificarea, vizualizarea,

construirea si documentarea artefactelor sistemelor sofware si de asemenea pentru modelarea în întreprinderi. UML este un limbaj de modelare care ofera o exprimare grafica a structurii si comportamentului software.

UML reprezintă un standard de notaţie. Introduce initial (ver. 1.0) un număr de 9 diagrame de descriere ale unui sistem informatic şi semantica acestor diagrame. UML nu propune şi un proces de utilizare a acestor diagrame in construcţia unei aplicaţii.

Cele nouă tipuri de diagrame propuse de UML sunt:

- de clase, ***class diagram***, care prezinta structura statica în termeni de clase si asocieri (relatii)
- de obiecte, ***object diagrams***, care prezinta obiectele si legaturile lor, fiind niste diagrame de colaborare simplificate, fara reprezentarea mesajelor trimise între obiecte;
- de cazuri de utilizare, ***use case diagram***, care prezinta functiile sistemului din punct de vedere al utilizatorului abordand problema comportamentului sistemului
- de colaborare, ***collaboration diagram***, care sunt reprezentari spatiale ale obiectelor, legaturilor si interactiunilor;
- de secvenţă, ***sequence diagram***, care prezinta temporal obiectele si interactiunile lor
- de stări, ***state diagram***, care prezinta comportamentul unei clase în termeni de stari;
- de activităţi, ***activity diagram***, care reprezinta comportamentul unei operatii în termeni de actiuni.
- de implementare, ***implementation (component, deployment) diagram,*** de componente si de punere in functiune (exploatare, constructie), care prezinta constructia componentelor pe dispozitivele hardware;

Aceste diagrame pot fi impărţite in 3 categorii:

- diagrame *statice* descriu structura şi responsabilitaţile sistemului informatic (diagramele de cazuri de utilizare (use-cases), clase, obiecte)
- diagrame *dinamice* descriu comportamentul şi interacţiunile care au loc intre diverse entităţi in cadrul sistemului informatic (diagrame de activităţi, colaborare, secvenţă, stări)
- diagrame *arhitecturale* descrie componentele executabile ale sistemului şi determină locaţiile fizice de execuţie şi nodurile de stocare a datelor (diagrame de componete, exploatare)

Urmatorul paragraf arata legatura istorica intre UML si predecesorii lui:
„Use-case diagrams are similar in appearance to those in OOSE.
Class diagrams are a melding of OMT, Booch, class diagrams of most other OO methods. Process-specific extensions (e.g., stereotypes and their corresponding icons) can be defined for various diagrams to support other modeling styles.
State diagrams are substantially based on the statecharts of David Harel with minor modifications. The Activity diagram, which shares much of the same underlying semantics, is similar to the work flow diagrams developed by many sources including many pre-OO sources.
Sequence diagrams were found in a variety of OO methods under a variety of names (interaction, message trace, and event trace) and date to pre-OO days. Collaboration diagrams were adapted from Booch (object diagram), Fusion (object interaction graph), and a number of other sources.
Collaborations are now first-class modeling entities, and often form the basis of patterns.
The implementation diagrams (component and deployment diagrams) are derived from

Booch's module and process diagrams, but they are now component-centered, rather than module-centered and are far better interconnected."

UML este in continua dezvoltare, noi concepte sunt adaugate de la o versiune la alta devenind astfel tot mai complex si greu de utilizat.

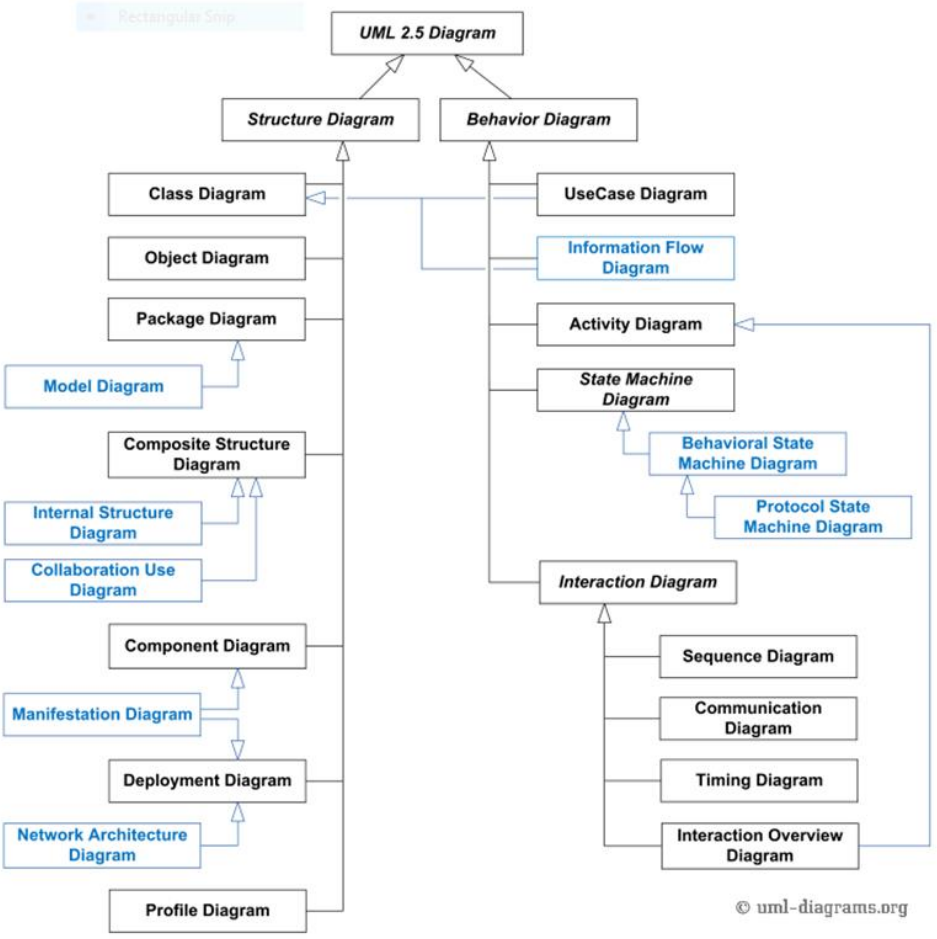
Versiunea actuala este UML 2.x care introduce o alta **classificare bazata pe**:

-Diagrame de structura (structural)

-Diagrame de comportament (behavioral), si alte sub-diagrame.

Ultima versiune (in 2020-21) este UML 2.5.x:

https://www.uml-diagrams.org/uml-25diagrams.html



UML 2.5 Diagrams Overview.
Note, items in blue are not part of official taxonomy of UML 2.5 diagrams.

2. Mecanisme comune UML

UML defineste un mic numar de mecanisme comune care asigura integritatea conceptuala a notatiilor. Aceste mecanisme comune cuprind:

-stereotipurile - specializeaza clasele metamodelului;
-etichetele - extind atributele claselor metamodelului;
-notele (comentariile);
-constrângerile - extind semantica metamodelului;
-relatia de dependenta;
-dualitatile (tip , instanta) si (tip , clasa).

Stereotipurile

Stereotipurile fac parte din mecanismele de extensibilitate prevazute de UML. Fiecare element de modelare al UML poate avea un stereotip atunci când semantica elementului este insuficienta. Stereotipul:

-permite metaclasificarea unui element al UML;
-permite utilizatorului (metodologist, constructor de utilitare, analist, proiectant) sa adauge noi clase de elemente de modelare, în plus fata de nucleul predefinit de UML;
-usureaza unificarea conceptelor apropiate, cum ar fi subsistemele sau categoriile, care sunt exprimate prin intermediul stereotipurilor de împachetare;
-permite extinderea controlata a claselor metamodelului, de catre utilizatorii UML, un element specializat printr-un stereotip S fiind semantic echivalent cu o noua clasa a metamodelului, denumita ea însasi S.

De fapt, toata notatia UML poate fi construita pornind de la clasele Entitate si Stereotip, celelalte concepte putând fi derivate (specializate) prin aplicarea mecanismului Stereotip asupra clasei Entitate. Creatorii UML au cautat un echilibru între:

-clasele, incluse de la început si
-extensiile obtinute prin stereotipizare,

astfel încât:

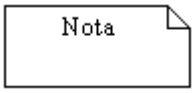
-doar conceptele fundamentale au fost exprimate sub forma de clase distincte;
-celelalte concepte, derivabile din cele de baza, au fost tratate ca stereotipuri.

Etichetele

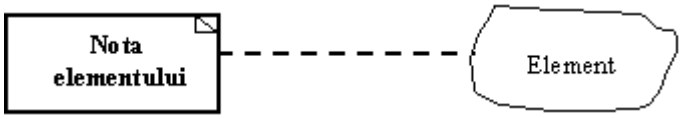
O eticheta este o pereche (nume, valoare) care descrie o proprietate a unui element de modelare. Proprietatile permit extinderea atributelor elementelor metamodelului. O eticheta modifica semantica (întelesul) elementului pe care îl califica.

Notele

O nota cuprinde ipotezele si deciziile aplicate in timpul analizei si a reprezentarii grafice. Notele pot contine orice informatie, inclusiv textul planului, fragmente de cod sau referinte la alt document. O nota contine un volum nelimitat de text. In consecinta notele pot fi dimensionate.
Notele se comporta ca niste etichete. Ele se pot folosi in orice fel de diagrama. Notele sunt doar explicatii oferite acolo unde apar in diagrama. Ele nu sunt considerate ca facand parte din model. Ele pot fi sterse ca orice alta componenta a diagramei.
Ancora notei conecteaza o nota la un element pe care il simuleaza.Aceste elemente pot lega o nota de un element sau mai multe elemente ale diagramei respective.
O nota poate sa nu fie legata, caz in care aceasta se refera la intreaga diagrama.
Forma grafica a unei note este un dreptungi care are un colt indoit:



In diagrama daca nota da explicatii asupra unor anumite elemente, atunci folosim si ancore ale notei care se reprezinta printr-o linie punctata ce face legatura intre element si nota:



Constrângerile

O constrângere este o relatie semantica între elementele de modelare. UML nu specifica o sintaxa particulara pentru constrângeri (de obicei se foloseste reprezentarea intre acolade **{..}**), care pot fi exprimate:

-în limbaj natural;
-în pseudocod;
-prin expresii de navigare;
-prin expresii matematice.

Relatia de dependenta

Relatia de dependenta defineste o relatie de utilizare unidirectionala între doua elemente de modelare, denumite sursa si tinta relatiei. Notele si constrângerile pot fi de asemenea surse ale unei relatii de dependenta.

Dualitatile (tip, instanta) si (tip, clasa)

Multe elemente de modelare prezinta dualitatea *(tip, instanta),* în care:
tipul denota esenta elementului, iar
instanta valorile sale ce corespund unei manifestari ale acelui tip.
De asemenea, dualitatea *(tip, clasa)* corespunde separarii între:
tipul, si
clasa, care furnizeaza realizarea acestei specificari.

3. Diagrame de clase

Diagramele de ***clase*** exprima la modul general structura statica a unui sistem, în termeni de clase si de relatii între aceste clase.

Asa cum o clasa descrie un ansamblu de obiecte, o asociere (connection) descrie un ansamblu de legaturi (links): obiectele sunt instante ale claselor, legaturile sunt instante ale asocierilor.

Diagramele de clase si diagramele de obiecte sunt reprezentari alternative pentru modelele obiectelor. Diagramele de clase contin clase si diagramele de obiecte contin obiecte, dar se pot mixa clasele si obiectele atunci când este de vorba de diferite tipuri de metadate.
Diagramele de clasa sunt mult mai relevante decât cele de obiecte. De obicei, se construiesc diagramele de clase si ocazional diagrame de obiecte pentru a ilustra structuri de date complicate ori transmiteri de mesaje.
Diagramele de clase contin simboluri grafice pentru clase. Se pot crea una sau mai multe clase pentru a reprezenta clasele din nivelul de vârf al modelului curent. De asemenea, se pot crea una sau mai multe diagrame de clase pentru a reprezenta fiecare pachet din model, clase care sunt , ele însele continute în pachetul ce cuprinde clasele de reprezentat.
Daca se modifica proprietatile sau relatiile unei clase prin editarea specificatiilor sale, diagramele de clase ce contin aceste clase se actualizeaza conform acestor modificari. Daca modificarile au loc în cadrul unei diagrame de clase, se vor actualiza specificatiile clasei si in celelalte diagrame de clase care contin aceasta clasa.

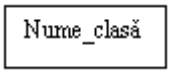
Diagramele de clase se utilizeaza pentru analiza, în care se arata rolurile si responsabilitatile comune ale entitatilor ce descriu comportamentul sistemului si pentru proiectare, unde se indica structura claselor ce formeaza arhitectura sistemului.

3.1. Clasele

O clasa contine structura si comportamentul comun unui set de obiecte. O clasa este o abstractie a entitatilor lumii reale. Cand acestea exista in lumea reala, ele sunt instante ale clasei, prin atribuite ale obiectelor concrete. Pentru fiecare clasa care are un comportament temporal semnificativ, putem crea o diagrama de stare care sa descrie acest comportament.

O clasa se reprezinta grafic printr-un dreptunghi impartit in trei compartimente , cu numele clasei in compartimentul de sus, o lista de atribute (cu tipuri optionale si valori) in cel din mijloc, si o lista de operatii (cu lista de argumente optionale si tipuri de returnare) in ultimul.

Compartimentele in care gasim atributele si operatiile pot fi ascunse (in cazul in care continutul acestora nu este semnificativ pentru contextul diagramei) pentru a reduce detaliile. Ascunderea compartimentelor nu face nici o declaratie despre absenta sau prezenta atributelor sau operatiilor, dar desenand compartimentele goale se exprima explicit ca nu exista elemente in acele parti (atribute si operatii).



Fiecare clasa trebuie sa aiba un nume. In diagramele de clase, toate simbolurile de clase cu acelasi nume se considera a reprezenta aceeasi clasa, indiferent de diagrama de clase in care apare.
Daca numele unei clase este scris cu italice, atunci acea clasa se considera a fi *abstracta*. In acest caz cel putin o metoda a clasei va fi abstracta.
O clasa poate contine si un **stereotip** cu proprietatile sale specifice. UML defineste urmatoarele stereotipuri de clase:
<<signal>>, o situatie speciala care declanseaza o tranzitie într-un automat;
<<interface>>, o descriere doar a operatiilor vizibile, practic o interfata soft;
<<metaclass>>, clasa de clase (ca în Smalltalk);
<<utilitare>>, o clasa redusa la conceptul de modul si care nu poate fi instantiata.

3.2. Structura si comportamentul claselor

Atribute si operatii

Atributele si operatiile (metodele) pot fi prezentate complet sau nu în compartimentele claselor.
Prin conventie, din cele doua compartimente suplimentare ale clasei, primul compartiment contine atributele iar al doilea compartiment contine operatiile.
Sintaxa utilizata pentru *descrierea atributelor* este:

Indic. Vizibilitate Nume_Atribut: Tip_Atribut[= Valoare_Initiala]

În faza analizei cerintelor se pot specifica de catre utilizator proprietati redundante, care pot fi eliminate prin utilizarea atributelor derivate, indicând clar faptul ca aceste proprietati sunt derivate din alte proprietati deja atribuite.
Sintaxa utilizata pentru *descrierea operatiilor* este:

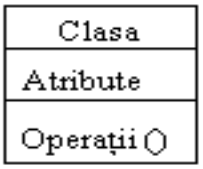
Indic. Vizibilitate NumeOperatie(NumeArgument: TipArg.[=ValImplicita], ...):TipReturnat

Dat fiind ca lungimea specificatiei poate fi mare, argumentele operatiilor pot fi suprimate în forma grafica. De obicei cand valoarea de retur e omisa se omit si parametrii formali.
Metodele pot fi precedate si ele de stereotipuri in acelas mod ca si la clase. Astfel *<<constructor>>*, *<<misc>>* indica ca am un constructor sau o metoda normala.
Atat in compartimentul atributelor cat si in cel al metodelor putem avea … (ellipsis) ce specifica ca mai sunt atribute sau metode nespecificate.

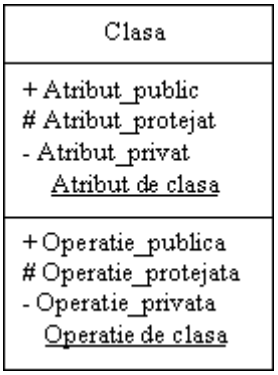
Vizibilitatea atributelor si a operatiilor

UML defineste 3 niveluri de vizibilitate pentru atribute si operatii:
public - element vizibil tuturor clientilor clasei (ofera partea de interfata pura) (simbol +);
protected - element vizibil subclaselor clasei, dar dependent de limbaj (simbol #);
private - element vizibil în interiorul clasei (ofera partea de implementare pura) (simbol -).
Informatia privind vizibilitatea, chiar daca este definita în model, poate sa nu fie întotdeauna

figurata în mod explicit. Explicit, nivelul de vizibilitate este simbolizat prin caracterele: +, #, -. Lipsa simbolului nu implica o anumita vizibilitate implicita.
Anumite atribute si operatii pot fi vizibile global, în toate expresiile lexicale ale clasei. Aceste elemente denumite variabile si operatii de clasa, sunt reprezentate ca si obiectele prin *sublinierea numelor*. Notatia se justifica prin faptul ca o variabila de clasa apare ca un obiect partajat de instantele clasei. Prin extensie, operatiile de clasa sunt de asemenea subliniate.
Ele se intalnesc in programare ca si atribute si operatii *statice.*



Amplasarea atributelor si operatiilor in reprezentarea unei clase

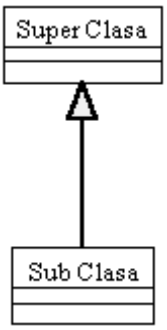


Reprezentarea atributelor si operatiilor din punct de vedere al vizibilitatii acestora

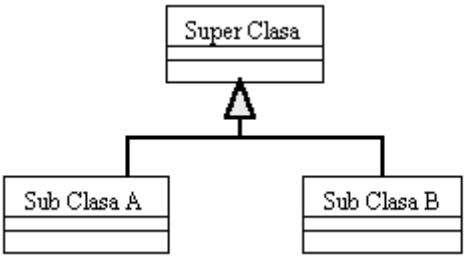
Generalizarea

UML utilizeaza termenul *generalizare* pentru a desemna relatia de clasificare intre un element mai *general* si un element *specific*. De fapt, termenul de generalizare reprezinta un punct de vedere bazat pe un arbore de clasificare.
De exemplu, un animal e un concept mai general decat o pisica, un caine sau un tigru, si reciproc, o pisica este un concept mai specializat decat un animal.
Elementul specific poate contine informatii proprii lui, cu conditia de a ramane complet coerent.
Generalizarea se aplica in primul rand claselor, pachetelor sau cazurilor de utilizare.
In cazul claselor, relatia de generalizare exprima faptul ca elementele unei clase sunt descrise de asemenea si de alta clasa (de fapt prin titlul unei alte clase). Relatia de generalizare este specificata prin *este un* ***(is a)***sau *este un tip de* ***(is a kind of)***. O pisica ***este un*** animal, ceea ce este o generalizare. O pisica are doua urechi, ceea ce nu e o generalizare, ci o *compunere*.

Relatia de generalizare se *reprezinta* prin intermediul unei sageti care indica (atinge) clasa mai generala pornind de la clasa specializata. Varful sagetii este un triunghi gol, ceea ce permite deosebirea fata de triunghiul deschis (unghiul) caracteristic navigarii unidirectionale a asocierilor.

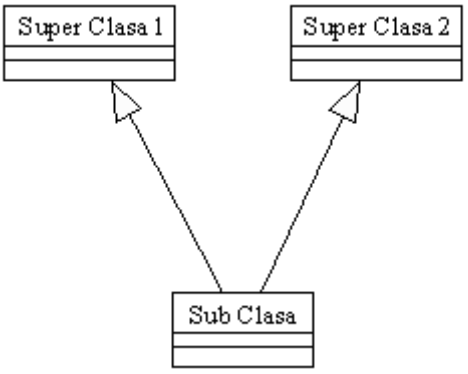


In cazul subclaselor multiple, sagetile pot fi unite intr-una singura, sau pot fi reprezentate independent, fara a exista un inteles diferit sau particular ale vreuneia dintre cele doua forme.



Atributele, operatiile, relatiile si constrangerile definite in superclase sunt mostenite in subclase tinand cont de specificatorii asociati.
In programare, relatia de generalizare este adesea implementata utilizand relatia de *mostenire* intre clase, propusa de limbajele OO. Mostenirea este o modalitate de realizare a clasificarii dar nu este unica.
In cazul in care o clasa are un numar mare de subclase, atunci se pot specifica doar cateva din ele, celelalte fiind indicate prin … (ellipsis).
Relatia de generalizare definita de UML e mai abstracta decat relatia de mostenire care exista in limbajele de programare ca C++ sau Java. Mostenirea e o relatie statica, ceea ce induce un cuplaj foarte puternic intre clase, ceea ce e nepotrivit pentru notiunea de clasificare dinamica. In analiza, este mai bine sa se vorbeasca despre generalizare sau despre clasificare, si sa se trateze mai tarziu, in faza de proiectare, modalitatea de implementare a generalizarii.

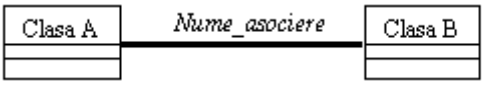
Clasele pot avea mai multe superclase, in acest caz generalizarea poarta numele de *mostenire multipla*, si mai multe sageti pleaca de la o subclasa catre diferite superclase. Generalizarea multipla consta din regruparea mai multor clase intr-una singura. Superclasele nu au in mod obligatoriu stramosi comuni (ascendenti comuni).



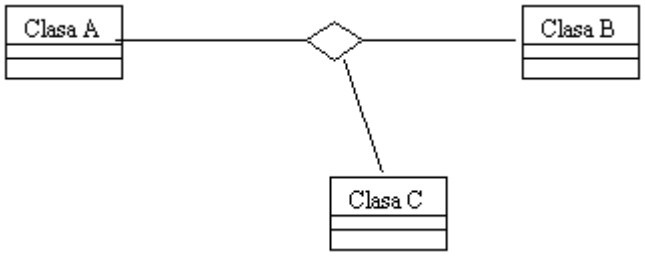
O clasa poate fi specializata in functie de mai multe criterii simultan. Fiecare criteriu al generalizarii este indicat in diagrama prin asocierea unui simbol al relatiei de generalizare, atunci cand sagetile sunt agregate simbolul aparand o singura data.

Asocierile

Asocierile sunt *relatii structurale* intre clase de obiecte. O asociere simbolizeaza o informatie a carei durata de viata nu este neglijabila in raport cu dinamica generala a obiectelor instantiate ale claselor asociate.



Majoritatea asocierilor sunt *binare,* conectand 2 clase. Asocierile sunt reprezentate prin linii care unesc clasele asociate. Asocierile pot fi reprezentate prin trasee rectilinii sau oblice, in functie de preferintele utilizatorului. Experienta recomanda utilizarea unui singur stil de reprezentare a traseelor pentru simplificarea citirii diagramelor unui proiect.
Pe langa asocierile binare pot exista asocieri *multiple* reprezentate prin intermediul unui romb catre care vin trasee de la diferitele componente ale asocierii.



Asocierile multiple pot fi reprezentate in general avansand asocierea la rang de clasa si adaugand constrangeri catre bratele multiple ale asocierii care se instantiaza toate simultan. Ca si la asocierile binare, extremitatile unei asocieri multiple sunt denumite *roluri* si pot purta un nume. Dificultatea de a gasi un nume diferit fiecarui rol al unei asocieri multiple este adesea semnul unei asocieri de multiplicitate inferioara.

Asocierile pot avea un nume, ***numele asocierii,*** el fiind figurat cu litere italice (prima fiind litera mare) la mijlocul liniei care simbolizeaza asocierea. Sensul citirii numelui poate fi precizat prin intermediul unui mic triunghi cu un varf de unghi indreptat catre clasa indicata si plasat in apropierea asocierii. Pentru simplitate triunghiul poate fi inlocuit cu semnele < sau > disponibile tuturor font-urilor.
Asocierile intre clase exprima in primul rand *structura statica,* fapt pentru care numele asocierii sub forma verbala, care evoca mai degraba comportamentul, este in afara spiritului general al diagramelor de clase. De aceea, numirea extremitatilor relatiilor permite clasificarea diagramelor ca si numirea asocierilor, dar intr-o forma pasiva, potrivita orientarii statice a diagramelor de clase.

O extremitate a unei asocieri poarta ***numele de rol***, astfel incat asocierile binare cu 2 roluri, au la fiecare extremitate cate unul. Rolul descrie modul in care o clasa vede o alta clasa dincolo de o asociere. Rolul este numit prin intermediul unei forme nominale, vizual

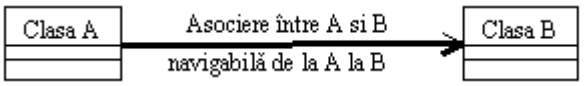
deosebindu-se de asociere prin faptul ca este figurat cu litere obisnuite si este plasat la una din extremitatile unei asocieri.
Numirea asocierilor si a rolurilor nu se exclud reciproc, desi in practica rar se combina cele doua constructii. De obicei se incepe prin completarea unei asocieri printr-un verb, care va servi mai tarziu pentru a construi un substantiv verbal care denumeste rolul corespunzator. Atunci cand doua clase sunt legate printr-o singura asociere, numele claselor este adesea suficient pentru a caracteriza rolul, numele rolurilor avand eficienta atunci cand mai multe asocieri leaga 2 clase, fiecare exprimand un concept distinct.
Prezenta unui numar mare de asocieri intre 2 clase poate fi suspecta. Fiecare asociere adauga un cuplaj intre cele 2 clase, iar cuplajul puternic indica de obicei o descompunere gresita. Pe de alta parte, acesta poate fi sensul figurarii de mai multe ori a aceleiasi asocieri, numind fiecare asociere cu numele mesajelor care circula intre obiectele instanta ale claselor asociate.

Fiecare rol al unei asocieri detine o indicatie asupra ***multiplicitatii*** care arata cate dintre obiectele clasei plasate la extremitatea respectiva a asocierii, pot fi legate de un obiect al clasei aflate la celalalt rol al asocierii. Multiplicitatea este o informatie detinuta de rol sub forma unei expresii naturale limitate.

Simbol	Multiplicitate (nr. obiecte)	Multime valori
1	Unu si numai unu (exact unu)	{ 1 }
0..1	Zero sau unu	{ 0 , 1 }
M..N	De la M la N (intregi naturali)	{ M, M+1 ,……, N-1 , N }
*	De la 0 la mai multe (neprecizat)	{ 0 , 1 , ….. , k }
0..*	De la 0 la mai multe (neprecizat)	{ 0 , 1 , ….. , k }
1..*	De la 1 la mai multe (neprecizat)	{ 1 , 2 , ….. , k }
		(M , N , k valori naturale)

O valoarea de multiplicitate mai mare decat 1 implica o multime de obiecte, multime nelimitata in cazul unei valori *, simbol care arata ca mai multe obiecte participa la o relatie, fara a preciza numarul de obiecte. Valorile multiplicitatii exprima constrangeri legate de domeniul aplicatiei, valabile pe intreaga durata de viata a obiectelor.
Multiplicitatile nu trebuie sa tina seama de regimurile tranzitorii, cum ar fi crearea sau distrugerea obiectelor, multiplicitatea 1 indicand faptul ca in regim permanent, un obiect are obligatoriu o legatura catre un alt obiect.
Determinarea valorilor optime ale multiplicitatii este foarte importanta pentru gasirea echilibrului intre:
-suplete si posibilitatea extinderii;
-complexitate si eficacitate.
Pentru *analiza* este importanta doar valoarea multiplicitatii, pe cand *proiectarea* trebuie sa selecteze structurile de date (stive, cozi, fisiere, tablouri) pentru implementarea multimilor care corespund valorilor multiplicitatii de tipul 1..*, 0..* sau *. Supraestimarea valorilor multiplicitatii conduce la cresterea cerintelor de stocare si a timpului de cautare.
Valoarea 0 de inceput a multiplicitatii pune pe de alta parte problema necesitatii existentei codului pentru testarea prezentei/absentei legaturilor intre obiecte. Valorile multiplicitatii sunt adesea utilizate pentru a deservi in mod generic asocierile. Formele cele mai intilnite sunt asocierile 1 la 1, 1 la M, si N la M.

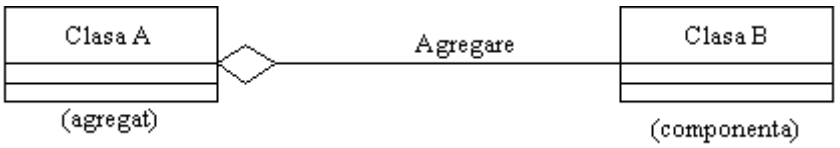
Deci asocierile descriu reteaua de relatii structurale care exista intre clase si care dau nastere legaturilor intre obiecte, instante ale acestor clase. Legaturile pot fi vazute ca niste ***canale (sageti) de navigatie*** intre obiecte. Aceste canale permit deplasarea in modele si realizarea formelor de colaborare care corespund diferitelor scenarii. Implicit, asocierile sunt navigabile in doua directii. In anumite cazuri, doar o directie de navigatie este utila, ceea ce se reprezinta printr-o sageata orientata catre rolul care este posibil. Absenta sagetii inseamna ca asocierea este navigabila in ambele sensuri. Mai jos, obiectele instantiate ale clasei A permit deplasarea catre obiectele instantiate ale clasei B, dar obiectele instantiate ale clasei B nu permit deplasarea catre obiectele instantiate ale clasei A.



O asociere navigabila doar intr-un singur sens poate fi vazuta ca o semi-asociere, distinctie care apare in special in faza de proiectare, dar care poate fi sesizata si in faza de analiza, atunci cand studierea domeniului releva o asimetrie de cerinte de comunicatie.
O asociere intre clase sau interfete implica o dependenta ce poate implica o referinta la unul din cele doua obiecte. Daca dependenta este considerata foarte generala sageata de navigatie va fi reprezentata printr-o linie intrerupta (ea specifica o interfata).

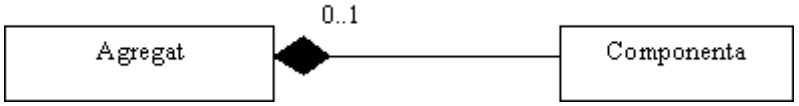
Agregarea

Relatia de *agregare* se foloseste pentru a indica o relatie de tip parte - intreg intre doua clase. Astfel, o agregare este o asociere asimetrica in care una dintre extremitati joaca un rol predominant (*clasa agregata*) in raport cu cealalta extremitate (*clasa componenta*). Oricare ar fi multiplicitatea, agregarea nu priveste decat un singur rol al unei asocieri.
Urmatoarele *criterii* permit indentificare agregarii:
-clasa face parte dintr-o alta clasa;
-valorile atributelor unei clase se propaga in valorile atributelor altei clase;
-o actiune asupra unei clase implica o actiune asupra altei clase.
Reciproca nu este intodeauna adevarata: agregarea nu implica in mod obligatoriu toate criteriile evocate mai sus. In cazurile in care exista vreo indoiala, asocierile simple sunt preferabile. Ca regula generala, trebuie aleasa intotdeauna solutia care implica un cuplaj cat mai slab.
O agregare se *reprezinta* printr-o linie intre doua clase avand un mic romb gol, la capatul in care se afla clasa de tip agregat:



Notiunea de agregare nu face nici o supozitie privind forma particulara de implementare. Contineree fizica este un caz particular de agregare, denumita ***compunere*** sau agregare *compusa*.
Atributele constituie un caz particular de agregare realizata prin valoare, ele sunt fizic continute de agregat. Notatia prin compunere e utilizata prin diagrame de clase in care un atribut participa la alte relatii in cadrul modelului.

Clasele obtinute prin compunere sunt denumite si ***clase compozite***. Ele ofera o abstractie a componentelor lor. In diagrame, compunerea se reprezinta printr-un ***romb plin***.



Agregarea *compusa* indeplineste doua conditii:

-instantele agregate trebuie sa apartina doar unui compus la un moment dat

-anumite operatii trebuie propagate de la compus la instantele agregate.

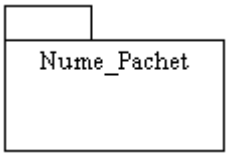
Unele asocieri (agregari) sunt *indirecte* caz in care ele sunt realizate prin intermediul unei alte clase, lucru ce se specifica realizand o legatura cu acea clasa implicata.

Pachetele

Entitatile UML pot fi grupate în pachete. Pachetele sunt containere logice în care pot fi plasate elemente înrudite, ca si directoarele din sistemele de operare. Desi orice entitate UML poate fi introdusa într-un pachet, de obicei rolul pachetelor este de a grupa clase si uneori cazuri de utilizare înrudite.

Un pachet trebuie sa aiba o functionalitate bine precizata, el nu trebuie sa îndeplineasca functii multiple, deoarece devine greu de înteles. Dependentele dintre pachete trebuie sa fie minime.

Împachetarea ofera un mecanism general pentru partitia modelelor si regruparea elementelor de modelare. Fiecare pachet este reprezentat ca un dosar (director, repertoar).



Fiecare pachet corespunde unui subansamblu al modelului si contine, în functie de model, clase, interfete, obiecte, relatii, componente sau noduri.

Descompunerea în pachete nu este legata de descompunerea functionala, fiecare pachet fiind o regrupare de elemente conform unui criteriu pur logic. Forma generala a sistemului (arhitectura sistemului) este exprimata prin ierarhia de pachete si prin reteaua de relatii de dependenta între pachete.

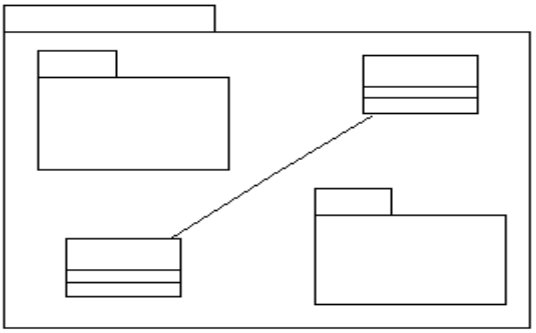
Un pachet defineste un *spatiu de nume*, astfel încât doua elemente diferite, continute în doua pachete diferite, pot purta acelasi nume.

Deci, intr-un pachet UML numele elementelor trebuie sa fie unice, dar un avantaj important al pachetelor este ca mai multe clase pot avea acelasi nume daca apartin unor pachete diferite.

Daca doua echipe A si B lucreaza în paralel, echipa A nu va trebui sa se preocupe de continutul pachetului echipei B, cel putin din punctul de vedere al denumirilor.

Un alt avantaj al pachetelor este că elementele sistemelor mari pot fi grupate în subsisteme mai mici sau faptul că este permisa dezvoltarea iterativa în paralel. Un pachet poate contine alte pachete, fara limitarea spatiului de încapsulare. Un nivel dat poate contine un amestec de pachete si de alte elemente de modelare, ca si directoarele care pot contine atât directoare cât

si fisiere.

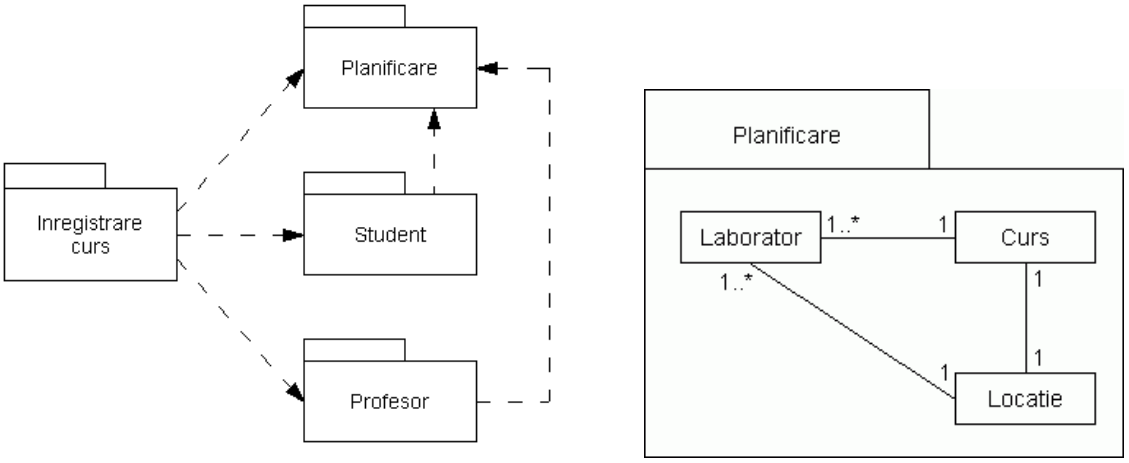


Pachetul de cel mai mare nivel este radacina împachetarii ansamblului unui model.
O clasa continuta într-un pachet poate sa apara si într-un alt pachet sub forma de element importat, de-a lungul unei relatii de dependenta între pachete. Importurile între pachete se reprezinta în diagramele de clase,diagramele de caz utilizare, diagramele de componente, prin intermediul unei relatii de dependenta stereotipizata orientata de la client catre furnizor.
O relatie de *dependenta între doua pachete* arata ca cel putin una dintre clasele pachetului client utilizeaza serviciile oferite de cel putin o clasa a pachetului furnizor.
Nu toate clasele continute într-un pachet trebuie sa fie vizibile în exteriorul pachetului.
Un pachet este o regrupare de elemente de modelare, dar si o încapsulare de elemente. La nivelul de clase, pachetele poseda o interfata si o implementare. Fiecare element al unui pachet poseda un specificator care arata daca elementul este sau nu accesibil din exteriorul pachetului. Valorile pe care le poate lua parametrul sunt:

-public, + (accesibil din exterior, interfatare)
-implementation, -, (privat, inaccesibil)

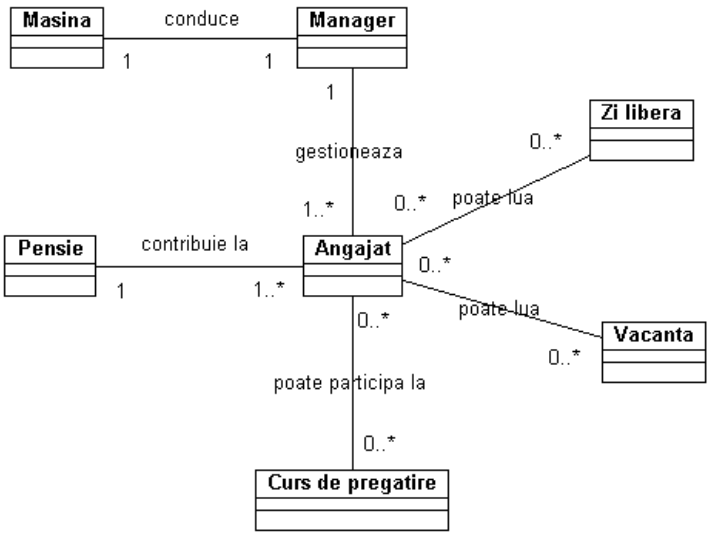
În cazul claselor, doar cele indicate ca publice apar în interfata pachetului care le contine, putând fi utilizate de clasele membre ale pachetelor client. Clasele de implementare, private nu sunt utilizabile decât în interiorul pachetului caruia îi apartin.
Relatiile de dependenta între pachete antreneaza relatii de agregare intre elementele de modelare continute în aceste pachete. Este important de a evita orice dependenta între pachete care formeaza grafuri ciclice, din motive de compilare a implementarii.
Ca regula generala, dependentele circulare pot fi reduse separând unul dintre pachetele respective în doua pachete mai mici, sau introducând un alt pachet intremediar. Anumite pachete sunt utilizate de catre toate celelalte pachete. Aceste pachete regrupeaza de exemplu clasele de baza ca multimi, liste, cozi, sau clase de tratare a erorilor. Aceste pachete poseda o proprietate care le defineste ca *pachete globale*. De aceea, nu este necesara trasarea relatiilor de dependenta între aceste pachete si utilizatorii lor pentru a nu satura încarcarea grafica a diagramelor.

Exemplu de utilizare a pachetelor:

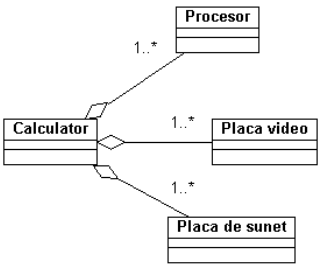


3.3. Exemple simple legate de clase

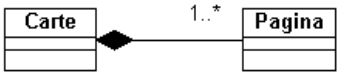
Relatii intre clase



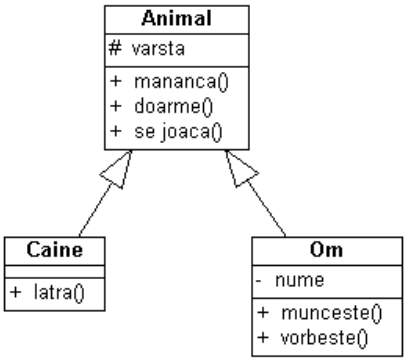
Agregarea este un concept care permite unui obiect să fie construit din altele.



Compunerea este un concept similar cu agregarea, însa mai puternic deoarece implica faptul ca un întregul nu poate exista fara parti



Mostenirea permite unei clase să primească caracteristicile altei clase, pe lângă cele specifice ei.

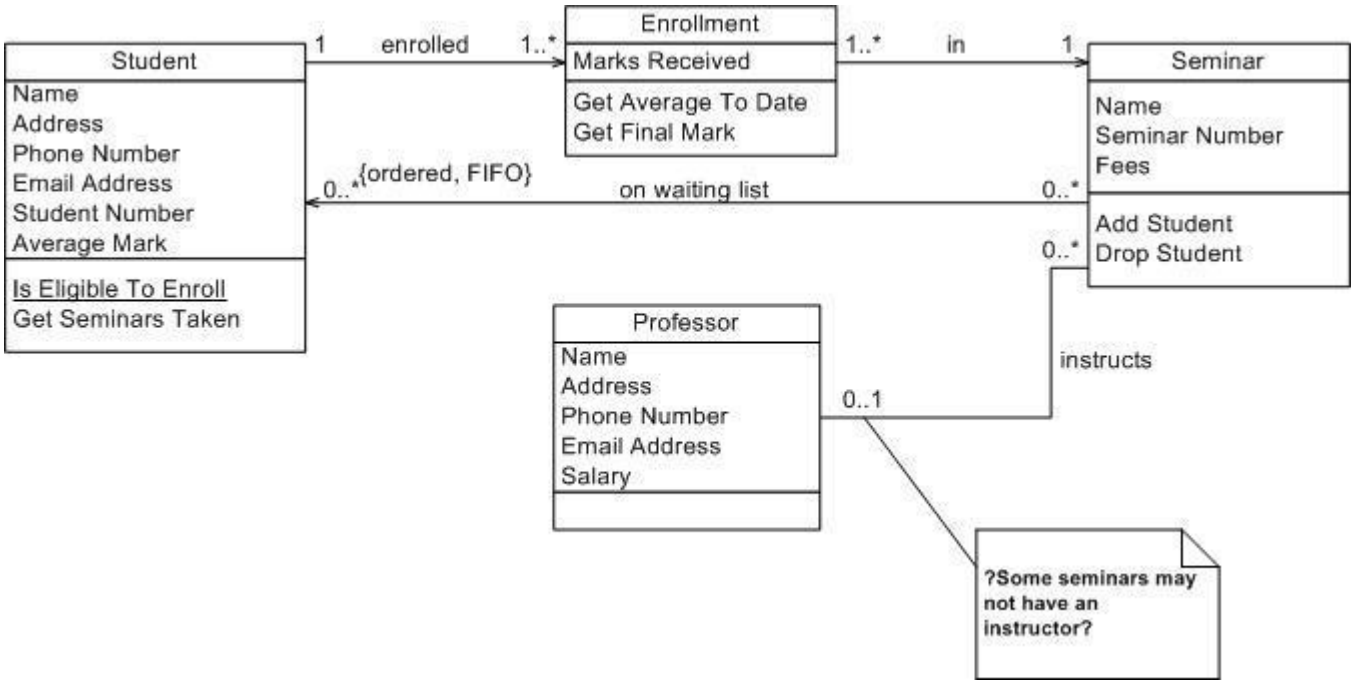


Atributele protejate sunt mostenite în clasele derivate dar sunt inaccesibile din exterior. Utilizarea abuziva a mostenirii conduce la dificultati în întretinerea programului si la urmarirea functionalitatilor (proliferarea claselor).

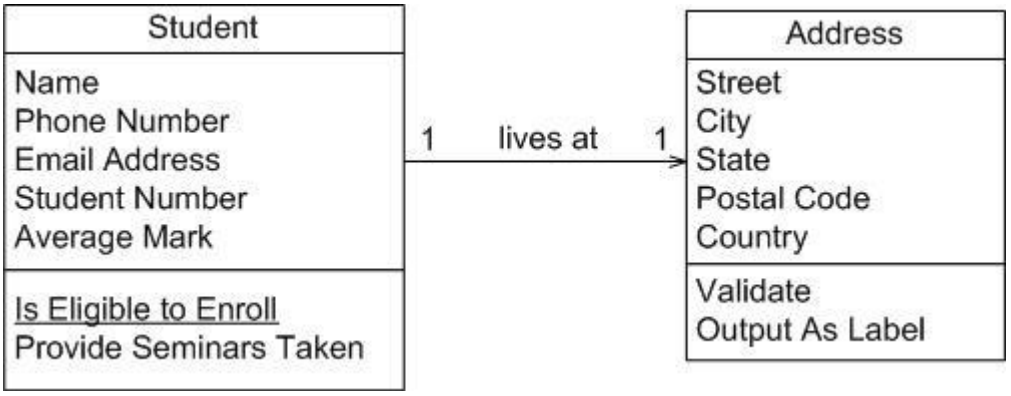
Mostenirea nu trebuie folosita decât ca mecanism de generalizare, adica se foloseste numai daca clasele derivate sunt specializari ale clasei de baza (Exista si alte tipuri de mosteniri nu doar cea prin specializare). Toate definitiile clasei de baza trebuie sa se aplice tuturor claselor derivate. Daca nu se aplica aceasta regula, clasele derivate nu sunt specializari ale clasei de baza.

Alte exemple de utilizare ale diagramelor de clase:

Conceptual, diagrama de clasa:

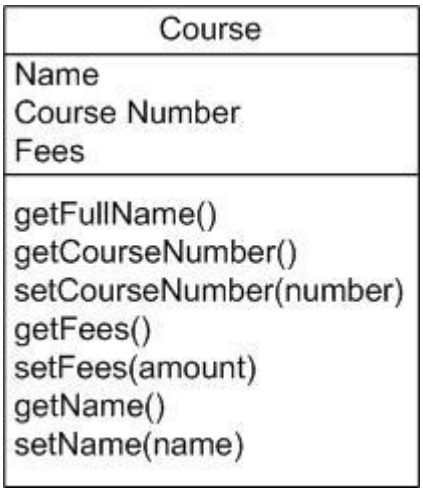


Clasele Student si Adress (diagrama conceptuala de clasa)

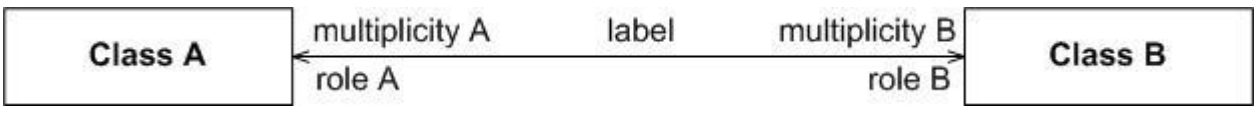


Clasa Seminar (diagrama conceptuala de clasa)

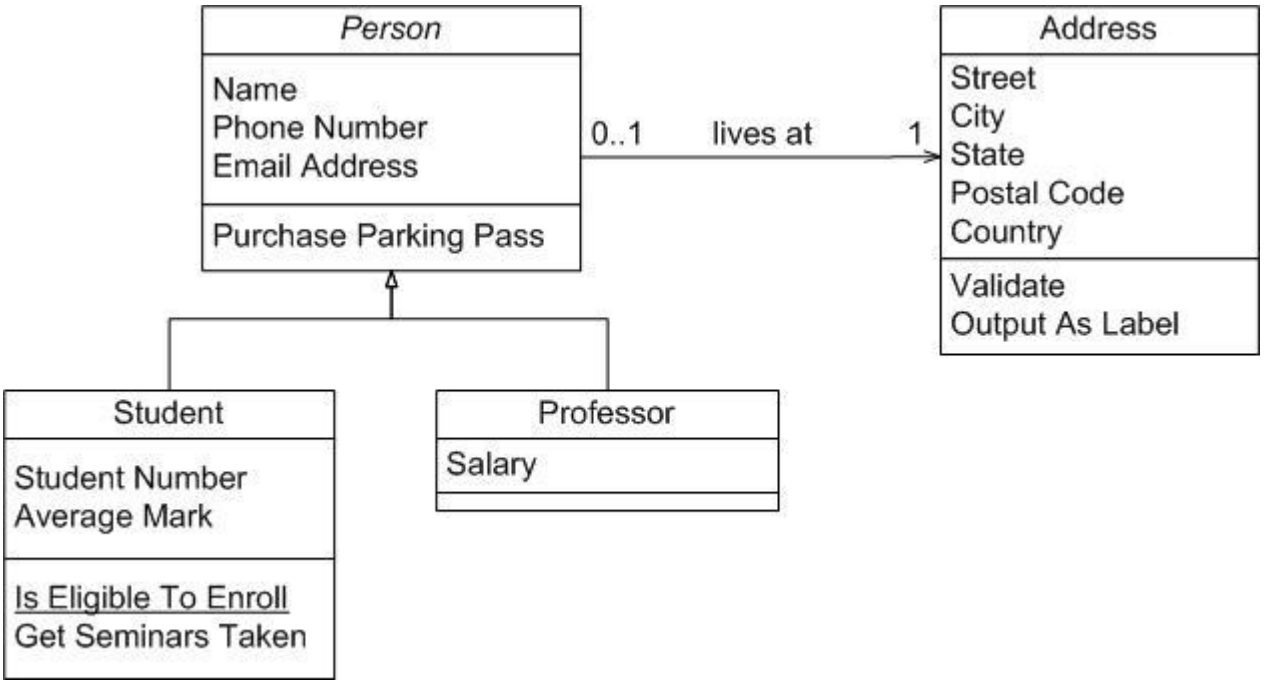
Clasa Course (cu metodele aferente)



Pentru asociatii s-au folosit urmatoarele notatii:



Ierarhia de mostenire intre clase:



4. Obiecte si diagrame de obiecte

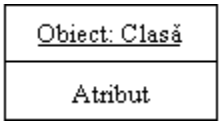
4.1. Notiunea de obiect

Un ***obiect*** este definit ca si o instanta a unei clase si este caracterizat de stari, comportament, si identitate. Structura si comportamentul unor obiecte similare este definita in clasa lor comuna. Un obiect care nu este numit, este referit ca o instanta a unei clase.

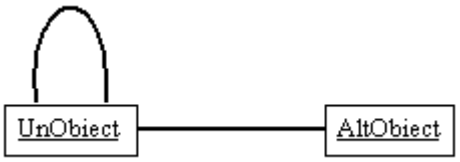
Fiecare obiect este ***reprezentat*** printr-un dreptunghi care contine fie numele obiectului, fie numele si clasa obiectului (separate prin doua puncte ":"), fie doar clasa obiectului (caz în care obiectul este denumit anonim), subliniate.

Numele singur corespunde unei modelari incomplete în care clasa obiectului nu a fost înca decisa (precizata). Clasa singura evita introducerea de nume inutile în diagrame, permitând exprimarea mecanismelor generale, valabile pentru mai multe obiecte.
De asemenea, în compartimentul obiectului poate fi inclus si stereotipul clasei, fie sub forma textuala, fie sub forma grafica, fie prin intermediul unei reprezentari grafice particulare care se substituie simbolului de obiect. Nu exista stereotipuri ale obiectelor, stereotipul care apare într-un obiect fiind întotdeauna cel al clasei obiectului.
Dreptunghiurile care simbolizeaza obiecte pot de asemenea sa detina un al doilea

compartiment, care contine valori ale atributelor acestora. Tipurile atributelor sunt deja descrise în clase, astfel încât nu mai este necesara figurarea acestora în reprezentarile obiectelor.

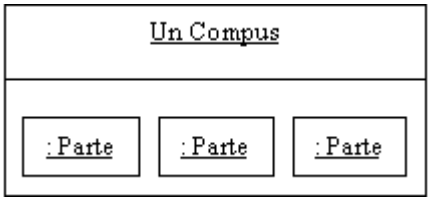


Un obiect interactioneaza prin intermediul ***legaturilor*** cu alte obiecte. O legatura este o instanta a unei asocieri, asa cum un obiect este o instanta a unei clase.
In general se afirma ca legaturile (links) specifica relatii intre obiecte iar asocierile (connections) sunt relatii intre clase.
O legatura trebuie sa existe intre doua obiecte, incluzand utilitatile clasei, doar daca este o relatie intre clasele care corespund respectivelor obiecte. Existenta unei relatii intre doua clase simbolizeaza o cale de comunicatie intre instantele claselor. Un obiect poate trimite *mesaje* spre un alt obiect.
Legaturile pot sustine mai multe mesaje in orice sens.
Obiectele sunt legate prin legaturi care sunt instante ale relatiilor între clasele obiectelor considerate.
Reprezentarea concreta a unei structuri prin obiecte este adesea mai sugestiva decât cea abstracta prin clase, în special în cazul structurilor recursive.
Intr-o diagrama de colaborare legaturile se reprezinta printr-o linie intre obiecte. Pot exista si legaturi de la un obiect la el insusi.



Majoritatea legaturilor sunt binare, dar exista si anumite legaturi multiple, de exemplu cele care corespund relatiilor ternare.
In diagrama de colaborare pe legaturile dintre obiecte se adauga mesajele corespunzatoare acestora.

Obiectele compuse din subobiecte pot fi reprezentate prin intermediul unui ***obiect compozit***, pentru reducerea complexitatii diagramelor. Obiectele compozite sunt figurate ca si obiectele clasice, cu diferenta ca atributele sunt înlocuite de obiecte, fie sub forma textuala subliniata, fie sub forma grafica.



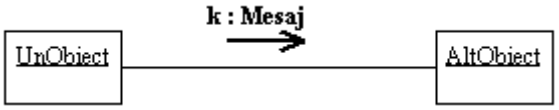
Obiectele compozite sunt instante ale claselor compozite, adica ale claselor construite pornind de la alte clase prin cea mai puternica forma de agregare.

Unitatea de comunicatie intre obiecte se numeste ***mesaj***. Mesajul este suportul unei relatii de comunicatie care leaga, in mod dinamic, obiectele care au fost separate prin procesul de

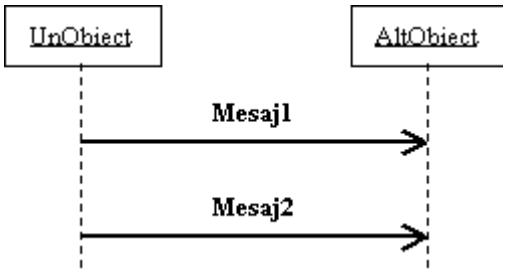
descompunere. Ele permit interactiunea flexibila, fiind in acelasi timp agent de cuplaj si agent de decuplare.
Mesajul asigura delegarea sarcinilor si garanteaza respectarea constrangerilor. Mesajul este un integrator dinamic care permite reconstituirea unei functii a aplicatiei prin punerea in colaborare a unui grup de obiecte.
Puterea sa de integrare se bazeaza pe polimorfism si pe legaturile dinamice. Un mesaj regrupeaza fluxurile de control si de date intr-o entitate unica.
Notiunea de mesaj este un concept abstract care poate fi implementat in mai multe variante, cum ar fi:

-apel de procedura;
-eveniment discret;
-intrerupere;
-cautare dinamica, etc.

In diagramele de colaborare *mesajele sunt reprezentate* prin sageti plasate in lungul legaturilor care unesc obiecte, cronologia acestora fiind figurata prin plasarea unui numar inaintea mesajului.



unde k este numarul de ordine al mesajului.
In diagramele de secventa mesajele se reprezinta prin sageti plasate intre liniile de viata ale obiectelor in ordinea lor cronologica.



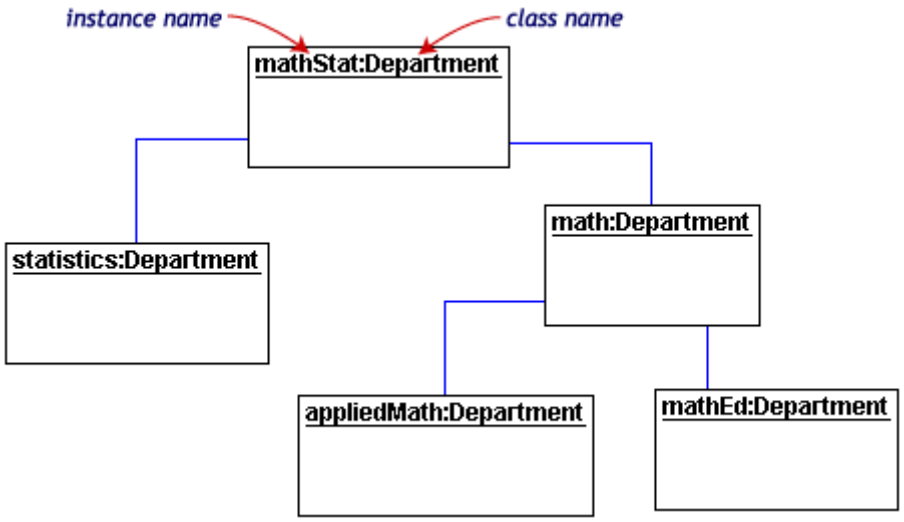
Formele de *sincronizare* ale mesajelor descriu natura mecanismelor de comunicatie care permit trecerea mesajelor de la un obiect la alt obiect.
Notiunea de sincronizare are obiect atunci cand mai multe obiecte sunt active simultan si este necesara protejarea accesului la obiecte partajat.
Notiunea de sincronizare precizeaza natura comunicatiei si regulile care conduc trimiterea mesajelor. Exista 5 mari categorii de trimiteri de mesaje:

-simplu,
-sincron,
-conditional,
-intarziat,
-asincron.

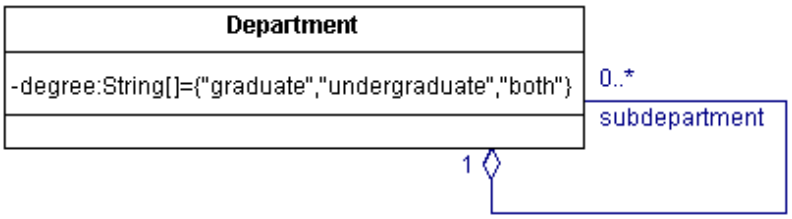
4.2. Diagrame de obiecte

Diagramele de obiecte, sau diagramele de instante, prezinta obiectele si legaturile între ele. Ca si diagramele de clase, diagramele de obiecte reprezinta structura statica. Notatia utilizata

pentru diagramele de obiecte este derivata din cea a diagramelor de clase, elementele care sunt instante fiind figurate subliniat.



Diagramele de obiecte sunt utilizate în primul rând pentru a prezenta un context, de exemplu situatia înaintea sau în urma unei interactiuni, dar pot servi si pentru a usura întelegerea structurilor de date complexe, asa cum sunt structurile recursive.



5. Diagrame de cazuri de utilizare - Use cases

Una dintre condiţiile ce trebuie indeplinite ca un proiect sa aibă succes este aceea ca cerinţele proiectului să fie definite intr-o manieră care să permită o uşoară intelegere a lui, indiferent de nivelul de pregătire informatica al celui care este implicat in proiect. De asemenea, modificările ce apar pe parcurs in cerinţe trebuie să fie cu uşurinţă asimilate de către membrii echipei de dezvoltare.
Diagramele de cazuri de utilizare au ***rolul*** de a reprezenta intr-o forma grafica functionalitatile pe care trebuie sa le indeplineasca sistemul in faza sa finala.
De aceea modelul realizat de diagramele de cazuri de utilizare alaturi de documentele de descriere succinta ale fiecarui caz de utilizare determinat poarta numele de ***model al cerinţelor***.

Diagramele de cazuri de utilizare sunt formate din entitati (actori si cazuri de utilizare) si relatii intre acestea.

Actorii sunt roluri jucate de diverse persoane sau sisteme şi care interacţionează cu sistemul aflat in dezvoltare. Este important de retinut faptul ca o persoană poate juca mai multe roluri şi un rol poate caracteriza mai multe persoane.
Reprezentare grafica a actorilor in UML este un omulet stilizat avand la subsol un text ce

reprezinta rolul jucat de actor.

Figura 5.1. Reprezentarea grafica a actorilor in UML

Determinarea actorilor se face răspunzand la intrebările:

- cine este, doreste sau e interesat de informaţiile aflate in sistem,
- cine modifica date,
- cine interactioneaza cu sistemul.

Raspunsurile concrete la aceste intrebari se introduc intr-o asa-numita **tabelă de evenimente** cu 4 coloane: Subiect(actor), Verb, Obiect, Frecvenţă. Aceasta tabela de evenimente permite de asemenea detectarea tuturor cazurilor de utilizare ale sistemului.
Cazurile de utilizare *reprezintă secvenţe de tranzacţii* ce au loc in dialog cu sistemul şi care sunt inrudite din punct de vedere comportamental. Practic, un caz de utilizare modeleaza un dialog intre un actor si sistem. Multimea de cazuri de utilizare ale unui sistem reprezinta toate modalitatile in care sistemul poate fi folosit.
Cazurile de utilizare:

- sunt unităţi de sine stătătoare, bine delimitatate (inceputul şi sfarsitul unui caz de utilizare sunt cuprinse in acesta).
- trebuie să fie iniţiate de un actor şi terminarea lor să fie *văzută* de un actor
- trebuie să indeplinească anumite scopuri de logica a problemei (dacă nu se poate găsi un astfel de obiectiv atunci cazul de utilizare trebuie regasit)
- trebuie să lase sistemul intr-o stare stabilă (nu poate fi indeplinit doar pe jumătate)

Cazurile de utilizare sunt *orientate pe scop* si reprezintă ceea ce sistemul trebuie să facă şi nu cum. Ele sunt neutre din punct de vedere tehnologic, putand fi utilizate in orice proces sau arhitectură de aplicaţie.
Reprezentarea grafica a cazurilor de utilizare in UML se realizeaza prin intermediul unui oval avand la baza numele cazului de utilizare.

Figura 5.2. Reprezentarea grafica a cazurilor de utilizare in UML

Fiecare caz de utilizare ce apare in una din diagramele ce modeleaza functionalitatea sistemului trebuie sa fie insotite de un document de descriere a sa ce va respecta urmatorul sablon:

-Nume
-Descriere
-Autori
-Stare
-Prioritate
-Precondiţii
-Postcondiţii

-Calea principală (sau BCE - Basic Course of Events)
-Căi alternative (sau ACE - Alternate Course of Events)
-Căi de excepţie

Relatii intre actori şi cazuri de utilizare:

- relatia de *asociere (comunicare)* - directia de navigare a relatiei (sageata) sugereaza cine initiaza comunicarea. In general comunicare intre actor si caz de utilizare este bi-directionala

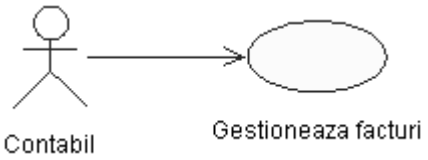


Figura 5.3. Reprezentarea grafica relatiei de comunicare intre actori si cazuri de utilizare

Relaţii intre cazuri de utilizare:

- relatia de *utilizare* are loc intre un caz de utilizare si oricare alt caz de utilizare ce utilizeaza functionalitatea acestuia. Se reprezinta grafic printr-o linie avand la capatul corespunzator cazului de utilizare folosit un triunghi si este etichetat cu stereotipul <<Uses>> (stereotipul este un concept introdus in UML care permite extinderea elementelor de modelare de baza pentru a creea noii elemente).

- relatia de *extindere* este folosita pentru a sugera un comportament optional, un comportament care are loc doar in anumite conditii sau fluxuri diferite ce pot fi selectate pe baza selectiei unui actor. Reprezentarea grafica este similara cu cea a relatiei de utilizare, dar eticheta este <<Extends>>.

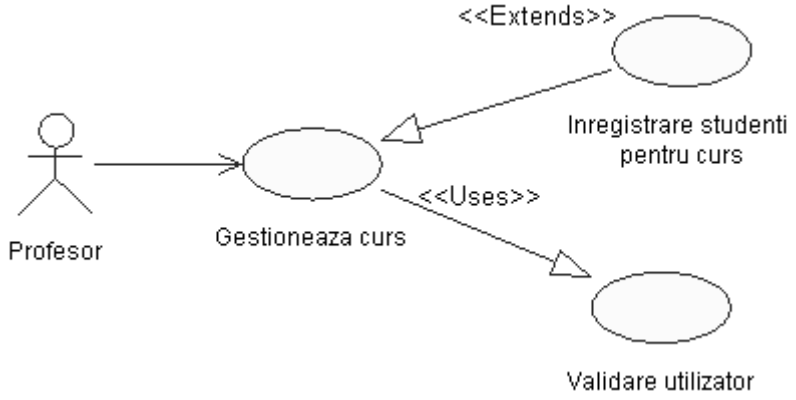


Figura 5.4. Reprezentarea grafica a relatiilor de extindere si utilizare intre cazuri de utilizare

Relaţii intre actori

- relatia de *generalizare* semnifica faptul ca un actor poate interactiona cu sistemul in toate modalitatile prin care interactioneaza un altul. Se reprezinta ca o relatie de extindere intre doua cazuri de utilizare fara a avea stereotip.

- relatia de *dependenţă* semnifica faptul ca, pentru a interactiona cu sistemul prin intermediul unui caz de utilizare, un actor depinde de alt actor. Se reprezinta printr-o linie intrerupta avand la un capat o sageata.

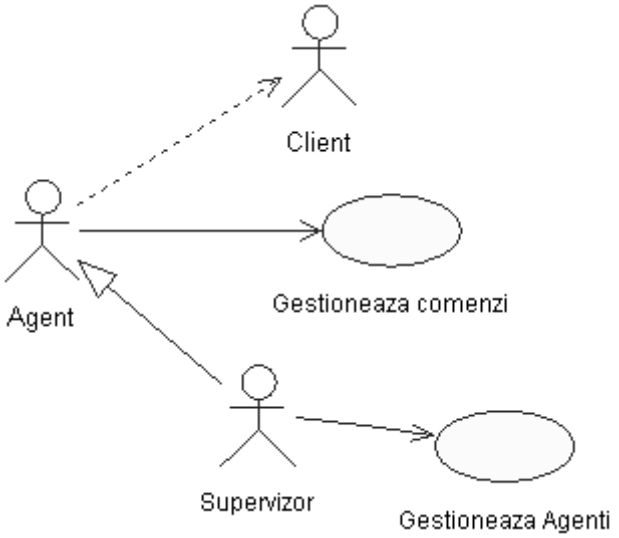


Figura 5.6. Reprezentarea grafica a relatiilor de generalizare si dependenta intre actori

Deci reprezentarea cazurilor de utilizare presupune descrierea multimii de interactiuni dintre utilizator si sistem. Cazurile de utilizare sunt denumite de obicei printr-o combinatie substantivala-verbala, de exemplu: Plateste factura, Creeaza cont etc.

Cazurile de utilizare au o importanţă deosebită in utilizarea UML. Ele definesc domeniul sistemului, permiţând vizualizarea dimensiunii si sferei de actiune a întregului proces de dezvoltare. Sunt similare cerintelor, dar cazurile de utilizare sunt mai clare si mai precise datorita structurii riguroase de notatie. Suma cazurilor de utilizare este sistemul ca întreg; ceea ce nu este acoperit de un caz de utilizare se situeaza în afara sistemului de construit. Ele permit comunicarea dintre client si dezvoltatori, de vreme ce diagrama este foarte simpla si poate fi înteleasa de oricine; ghideaza echipele de dezvoltare în procesul de dezvoltare şi ajuta echipele de testare si autorii manualelor de utilizare.
Într-un anumit scenariu, fiecare interactiune utilizator-sistem trebuie sa fie un caz de utilizare sau un singur caz de utilizare poate încapsula toate interactiunile. Un caz de utilizare trebuie sa satisfaca un scop pentru actor.

6. Diagrame de interactiune

Descrierea *comportamentului* (behavior) implica doua aspecte, descrierea structurala a participantilor şi descrierea modelelor de comunicatie (operatii + mod de reactiune). Modelul de comunicatie al instantelor care joaca un rol pentru îndeplinirea unui anumit scop se numeste ***interactiune.*** Diagramele de interactiune au doua forme bazate pe aceleasi informatii de baza, dar care se concentreaza fiecare pe un alt aspect al interactiunii. Diagramele de colaborare si diagramele de secventa sunt aceste reprezentari alternative pentru interactiunile dintre obiecte.

6.1. Diagrama de secvente

Diagramele de secventa prezinta interactiunile între obiecte din punct de vedere *temporal*, contextul obiectelor, nefiind reprezentat în mod explicit ca în diagramele de colaborare,

reprezentarea concentrându-se pe exprimarea interactiunilor.

O diagrama de secventa reprezinta o interactiune între obiecte insistând pe cronologia (ordinea temporala) a expedieri mesajelor. Notatia este derivata din diagramele MSC (Message Sequence Chart) OO ale grupului Pattern-uri Siemens (1996).

Liniile de viata sunt elemente ce caracterizeaza diagramele de secventa. O linie de viata a unui obiect poate fi considerata ca axa timpului pentru diagrama de secventa care dupa cum stim, prezinta interactiunile intre obiecte din punct de vedere temporal.
Astfel, ordinea cronologica a expedierii mesajelor este data de pozitionarea lor pe aceste axe (linii de viata).

In diagramele de secventa fiecare obiect este insotit de o linie verticala punctata. Aceste linii reprezinta de fapt liniile de viata ale obiectului.

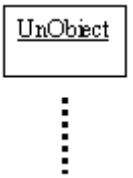
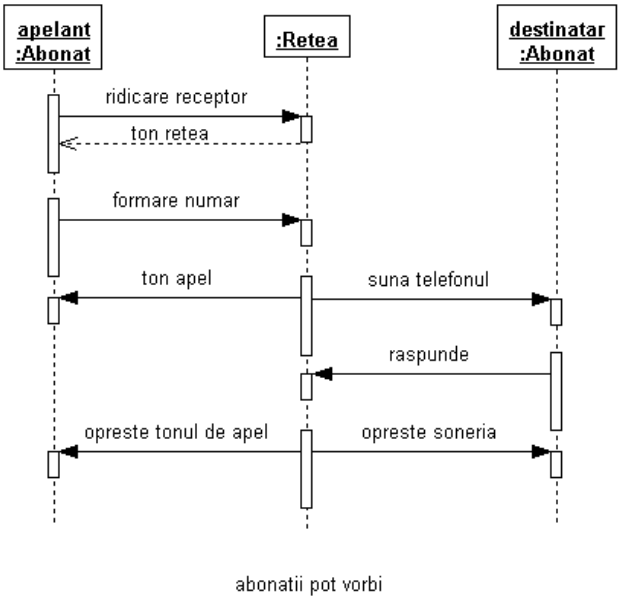


Diagrama de secvente pune accentul pe aspectul temporal (ordonarea în timp a mesajelor). Notatia grafica este un tabel care are pe *axa X obiecte,* iar pe *axa Y mesaje* ordonate crescator în timp. Axa Y arata pentru fiecare obiect timpul ca o linie verticala punctata („linia vietii" unui obiect, engl. "lifeline") si *perioada* în care obiectul preia controlul executiei (reprezentata printr-un dreptunghi) si efectueaza o actiune, direct sau prin intermediul procedurilor subordonate.

În diagrama de secvente utilizam obiecte, nu clase. Într-un program pot exista mai multe instante ale aceleiasi clase care au roluri diferite în sistem. Un obiect este identificat de numele sau si numele clasei pe care o instantiaza. Numele obiectului poate sa lipseasca daca nu este semnificativ pentru întelegerea comportamentului sistemului.
Liniile orizontale continue semnifica *mesaje initiate* de obiecte, iar *liniile orizontale punctate* reprezinta *mesaje-raspuns*.

Exemplu:



abonatii pot vorbi

6.2. Diagrama de colaborare

Diagramele de clase si obiecte arata relatiile dintre clase si obiecte. Ele ofera informatii despre interactiunile care apar clase dar nu arata succesiunea in care apar interactiunile sau concurenta ce poate sa apara.
Diagrama de colaborare se concentreaza pe rolurile instantelor si relatiile dintre ele. Ea nu contine timpul ca o dimensiune separata, de aceea secventa de comunicatii si firele de executie concurente trebuie numerotate.
O diagrama de colaborare este o diagrama de interactiuni care *arata secventele de mesaj* ce implementeaza o operatie sau o tranzactie. O diagrama de colaborare prezinta obiectele, legaturile si mesajele dintre ele. De asemenea, diagramele de colaborare pot contine simple instante de clase.
Fiecare diagrama de colaborare ofera o imagine asupra interactiunilor sau a relatiilor structurale care au loc între obiecte si a obiectelor ca entitati în modelul curent.
Diagramele de colaborare contin elemente reprezentând obiecte. Se pot crea una sau mai multe diagrame de colaborare care sa prezinte interactiunile pentru fiecare pachet logic din model; de asemenea, diagramele de colaborare sunt continute la rândul lor de pachete logice care cuprind obiectele prezente în diagrame.
O specificatie a unui obiect poate sa indice si sa modifice proprietatile si relatiile obiectului. Informatia în specificatie este prezentata textual; de asemenea, unele informatii pot fi expuse în interiorul simbolurilor grafice reprezentând obiecte în diagrama de colaborare. Daca modificam din specificatiile obiectului proprietatiile sau relatiile, diagrama de colaborare ce contine respectivul obiect se va actualiza cu noile date. Daca modificarile proprietatilor sau relatiilor unui obiect se fac în cadrul diagramei, atunci se vor actualiza specificatiile obiectului.

Deci diagrama de colaborare este construita ca o diagrama de obiecte, in care un numar de obiecte sunt prezentate impreuna cu relatiile lor, utilizind notatiile diagramelor de clase si obiecte. Sagetile mesajelor sunt trasate intre obiecte pentru a arata fluxul de mesaje intre obiecte. Pe mesaje sunt plasate etichete, care printre altele arata ordinea in care mesajele sunt trimise. Ele pot avea si alte detalii cum ar fi: conditii, iteratii, valori returnate. Dupa familiarizarea cu sintaxa etichetelor mesajelor un dezvoltator poate citi colaborarea si urma fluxul de executie al mesajelor. O diagrama de colaborare poate contine si obiecte active care sunt executate concurent (paralel) cu altele.
La o diagrama de colaborare avem:

-ori ce numar de interactiuni poate fi asociat cu o legatura

-ori ce interactiune implica apelul unei metode

-dupa fiecare interactiune sau grup de interactiuni e o sageata care puncteaza obiectul a carui metoda este apelata prin interactiune

-multimea intreaga de obiecte si interactiuni aratate intr-o diagrama de colaorare este numita in mod colectiv, colaborare

-fiecare dintre interactiuni incepe cu o secventa de numere si : (semi colon)

-secventele de numere indica ordinea in care apelurile de metode apar. O interactiune cu numarul 1 va fi inaintea uneia cu numarul 2

-secventele de numere multinivel sunt reprezentate prin 2 sau mai multe numere separate prin punct. Ele corespund apelurilor de metode de la nivele multiple. Cifrele din stanga celei mai din dreapta cifre este prefix.

Exemplu: 1.1.4, aici 1.1. e prefix pentru 4

-interactiunile numerotate cu o secventa multinivel de numere apar in timpul altei interactiuni de apel metda. Celalalt apel de metoda este determinat de prefixul interactiunii.
Exemplu: Apelurile metodei interactiunilor numerotate 1.1 si 1.2. sunt realizate in timpul apelului metodei din interactiunea 1. Intre interactiunile numerotate cu acelasi prefix metodele sunt apelate in ordinea determinata de ultimul numar in secventa lor de numere.

Intre obiecte putem avea legaturi, iar UML permite utilizarea unui simbol grafic pentru multiobiecte care permite legaturi cu un numar nedefinit de obiecte, printr-un dreptunghi in spatele altui dreptunghi.
Obiectele create ca rezultat al unei colaborari pot fi marcate cu proprietatea ***{new}***. Obiectele temporare ce exista doar in timpul unei colaborari pot fi marcate cu ***{transient}***.

.

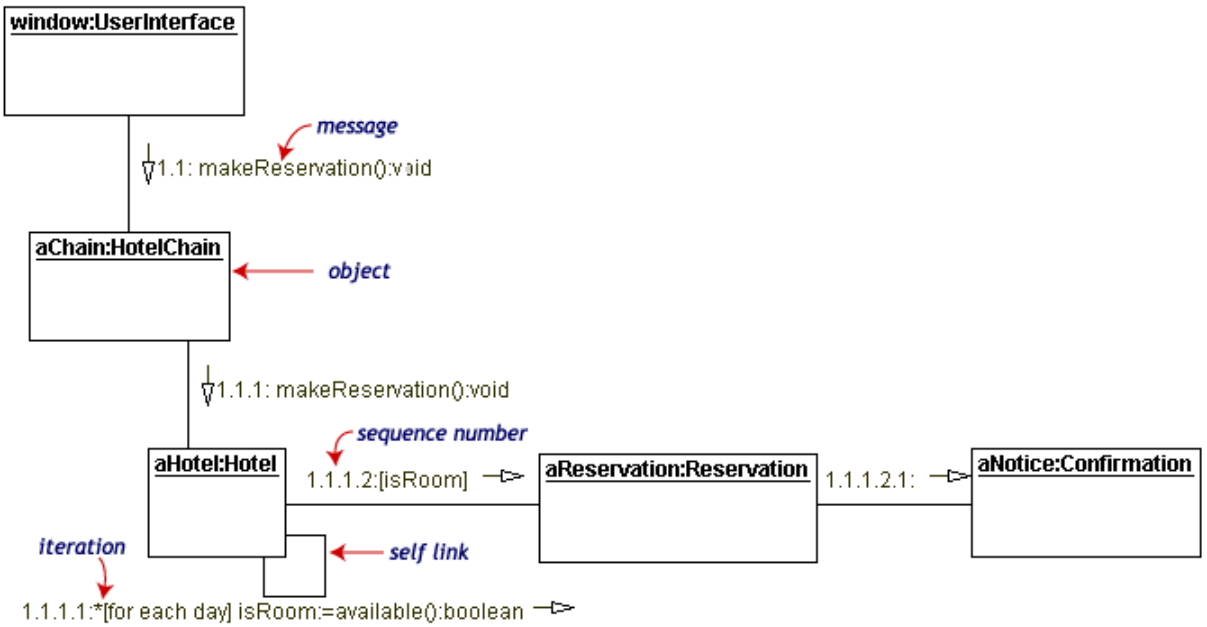


Figura 6.1. Diagrama de colaborare

Unele interacţiuni apar mai degrabă ***concurent*** decât secvenţial. In acest caz o litera la sfarsitul unui numar secvenţial indica concurenta dintre interacţiuni.
Un asterisc, *, dupa un numar secvenţial indica o interactiune ***repetata.*** De obicei in acest caz se foloseste stereotipul <<self>> pentru a arata si clarifica acest tip de interactiune.
UML permite ***asocierea unei conditii*** interactiunii repetitive prin punerea ei dupa * intre paranteze patrate. Aceasta conditie poate fi exprimata in pseudocod sau un limbaj de programare.
In cazul ***multithreadingului*** pentru a asigura excluderea mutuala UML permite urmatoarele constructii care apar dupa o metoda:
{concurrency = sequential} nu e garantata functionarea corecta la apeluri de thread-uri multiple in acelas timp
{concurrency = concurrent} apelul e facut concurent corect
{concurrency = guarded} apelul este sincronizat si corect.
Se pot introduce si unele *rafinari* (sincronizari cu alte obiecte nu cele ale metodei apelate, preconditii, etc.) care insa nu sunt standardizate in UML.
Se pot introduce prin UML si apeluri *asincrone*, *impiedecate* (balking call) indicae prin sageti

ce se intorc (bent-back arrow), etc.

7. Diagrame de stare

Pentru a întelege comportamentele complexe ale obiectelor se utilizeaza diagramele de stari, care descriu modul de functionare a instantelor. Diagramele de stari UML descriu diferitele stari în care se poate gasi un obiect si tranzitiile dintre aceste stari.
O ***stare*** reprezinta o etapa în modelul comportamental al unui obiect. O *stare initiala* este cea în care se gaseste obiectul când este creat, iar o *stare finala* este o stare din care nu mai exista tranzitii. ***Tranzitia*** reprezinta schimbarea starii, trecerea dintr-o stare în alta, si poate fi determinata de un eveniment extern sau intern.
Diagramele de stari reprezinta vizual automatele cu numar finit de stari, din punctul de vedere al starilor si tranzitiilor.
O diagrama de stare este utilizata pentru a reprezenta starile unei clase date, evenimentele ce cauzeaza tranzitiile de la o stare la alta, si actiunile care rezulta din schimbarile starilor. Fiecare diagrama este asociata unei clase sau unui nivel mai înalt de diagrama de stare.
O diagrama de stare este un graf bazat pe stari conectate prin tranzitii. O diagrama de stare descrie o istorie a vietii obiectelor sau clasei date.
O diagrama de stare prezinta o singura stare initiala, una sau mai multe stari, una sau mai multe stari finale si tranzitiile între stari. Fiecare clasa din modelul curent ce are un comportament semnificativ în ceea ce priveste evenimentele ordonate, poate contine o singura diagrama de stare pentru a descrie acest comportament.
O specificatie de stare poate indica si modifica propritatile unei stari. Informatiile privind specificatiile starii sunt prezentate textual si în plus, pot aparea în desenul respectivei stari în reprezentarea diagramei de stare. Daca modificam în cadrul specificatiilor, proprietatile starii, atunci diagramele de stare se vor actualiza reflectând aceste schimbari.

Starea unui obiect reprezinta istoria cumulativa a comportamentului respectivului obiect. Starea acopera toate proprietatile statice ale obiectului si valorile curente pentru fiecare proprietate. Toate instantele ale aceleiasi clase exista in aceeasi stare.
Numele unei stari trebuie sa fie unic in clasa pe care o descrie, sau daca este imbricata unei stari, in interiorul acesteia.
Actiunile dintr-o stare pot exista la unul din cele patru momente:

-(on) entry (actiune ce se executa la intrarea dintr-o stare);

-(on) exit (actiune ce se executa la iesirea dintr-o stare);

-on an activity (on event, on:) (actiune executata la aparitia unui eveniment);

-upon event. Actiunea "upon event" va fi similara cu o tranzitie de stare avand urmatoarea sintaxa:

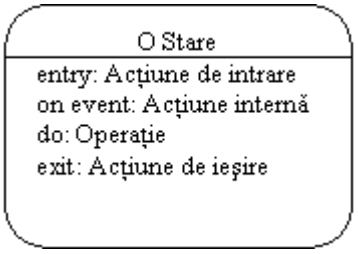
event(args)[conditie]

Actiunea upon event este diferita de *autotranzitie* (tranzitia unei stari la ea insasi). O autotranzitie executa alte actiuni "entry" si "exit", in timp ce o actiune "upon event" poate fi privita ca un eveniment intern care nu declanseaza orice alte actiuni.
Actiunile sunt de doua feluri:

-simple: sunt texte simple. Textul reprezinta tot ce dorim sa se intample cand se produce un eveniment.

-care trimit evenimente: sunt actiuni care declanseaza alt eveniment.

Actiunile trebuie sa fie introduse in forma specificatiei Starii Actiune.
Grafic, ***starea*** in diagrama de stare se reprezinta printr-un dreptunghi rotunjit in care scriem un nume si un compartiment:



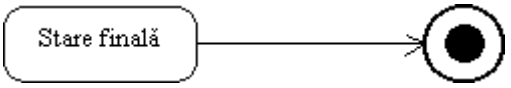
Stare initiala

O stare initiala este o stare speciala ne indica in mod explicit initializarea masinii cu stari. Starea iniala se conecteaza primei stari normale printr-o tranzitie neetichetata. Intr-o diagrama de stare putem avea exact o stare initala. Cand folosim *stari imbricate*, trebuie sa definim cate o stare initiala in fiecare context. In general, doar o tranzitie poate pleca din starea initiala. Totusi, tranzitiile multiple pot fi atasate starii initiale daca macar una dintre ele este etichetata cu o conditie. Nu se admit tranzitii care nu vin dintr-o stare. Sarile initiale se pot eticheta daca se doreste. Specificatiile stariilor sunt asociate fiecarei stari initiale.
O stare initiala se reprezinta in diagrama de stare printr-un mic cerc plin.

Stare finala

O stare finala reprezinta starea de final (terminala) a unui sistem. Aceasta se foloseste in diagrama de stare cand vrem sa aratam explicit finalul masinii cu stari. Tranzitiile pot doar sa existe in starea finala; odata ce masina cu stari se sfarseste, ea dispare. In mod normal, ne putem asigura ca masina cu stari asociata unei clase nu va mai exista atunci cand obiectul referit in aceasta este distrus si, de aceea, niciodata nu ajunge intr-o stare finala. Totusi, putem folosi o stare finala pentru a arata explicit finalul, daca este necesar. Intr-un context pot exista un numar nedefinit de stari finale.
O stare finala se reprezinta grafic printr-un cerc plin concentric altui cerc gol:

In diagrama de stare, pentru a construi o stare ca stare finala ii atasam printr-o sageata simbolul de stare finala.

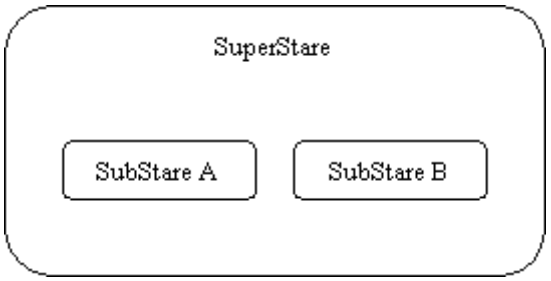


Putem eticheta starile finale daca dorim; specificatiile starii sunt asociate fiecarei stari finale.

Stari imbricate

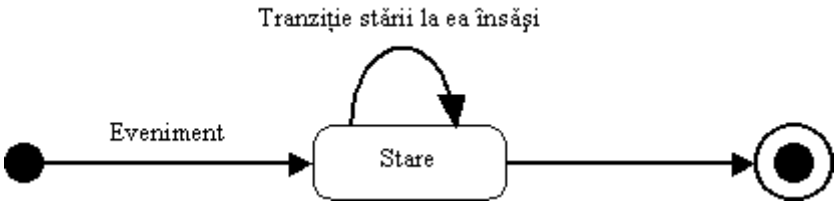
Diagramele de stari pot deveni destul de dificil de citit atunci cand, datorita exploziei combinatorii, numarul de conexiuni intre stari devine ridicat.
Solutia pentru a rezolva aceatsa situatie consta imbricarea starilor, utilizand principiul generalizarii starilor. Astfel, vom crea stari mai generale denumite superstari. Starile mai

specifice (imbricate) poarta numele de substari, iar numarul lor nu este limitat. Substarile mostenesc caracteristicile superstarilor lor, in particular variabilele de stare si tranzitiile externe.
Se poate efectua si operatia inversa de a descompune in substari, aceasta numindu-se *descompunere disjunctiva* (de tip sau - exclusiv). Aici tranzitiile pot fi mostenite cu exceptia cazului in care descompunerea in substari are ca scop definirea unei stari particulare pentru tratarea unei tranzitii interne.
Grafic, dreptunghiurile reprezentand starile imbricate (substarile) sunt introduse in cel al superstarii:



Tranzitii

O tranzitie intre stari reprezinta schimbarea unei stari cauzata de un eveniment. Intr-o diagrama de stari, tranzitiile sunt folosite pentru a lega doua stari sau indica o tranzitie a unei stari in ea insasi. Poti introduce in diagrama una sau mai multe tranzitii dintr-o stare atat timp cat fiecare tranzitie este unica. Tranzitiile dintr-o stare nu pot avea acelasi eveniment, doar in cazul cand exista conditii pentru acel eveniment.
O tranzitie se reprezinta grafic printr-o sageata orientata spre urmatoarea stare:

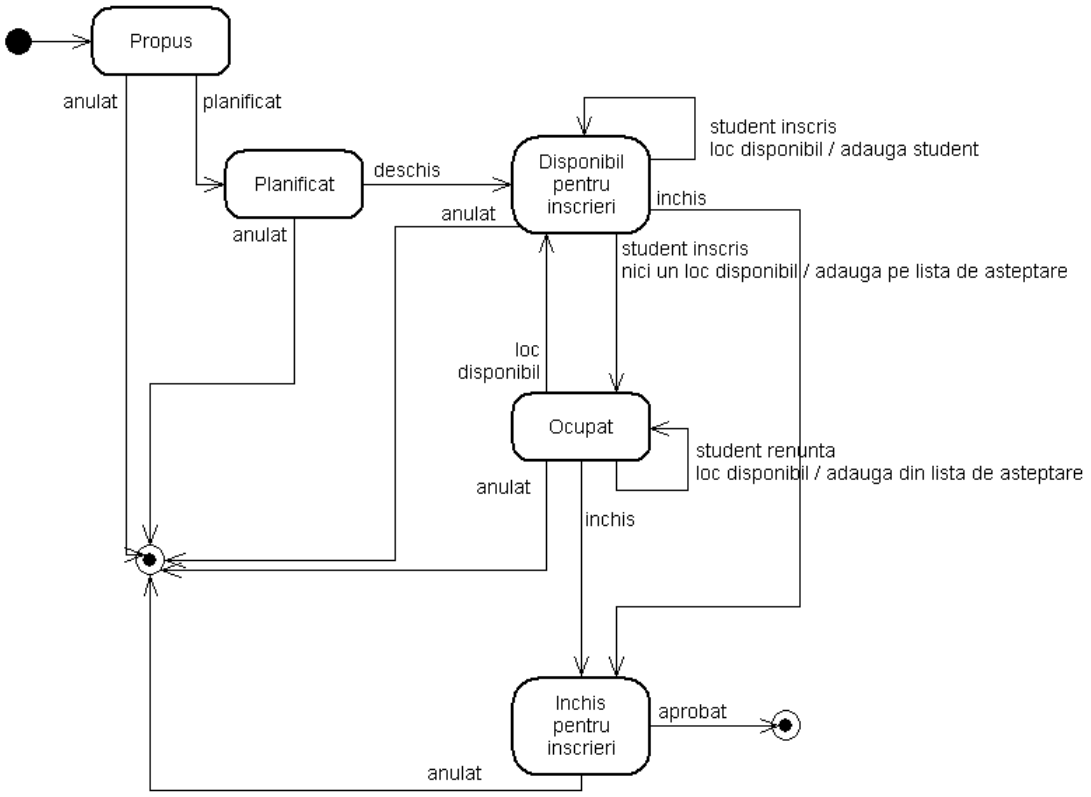


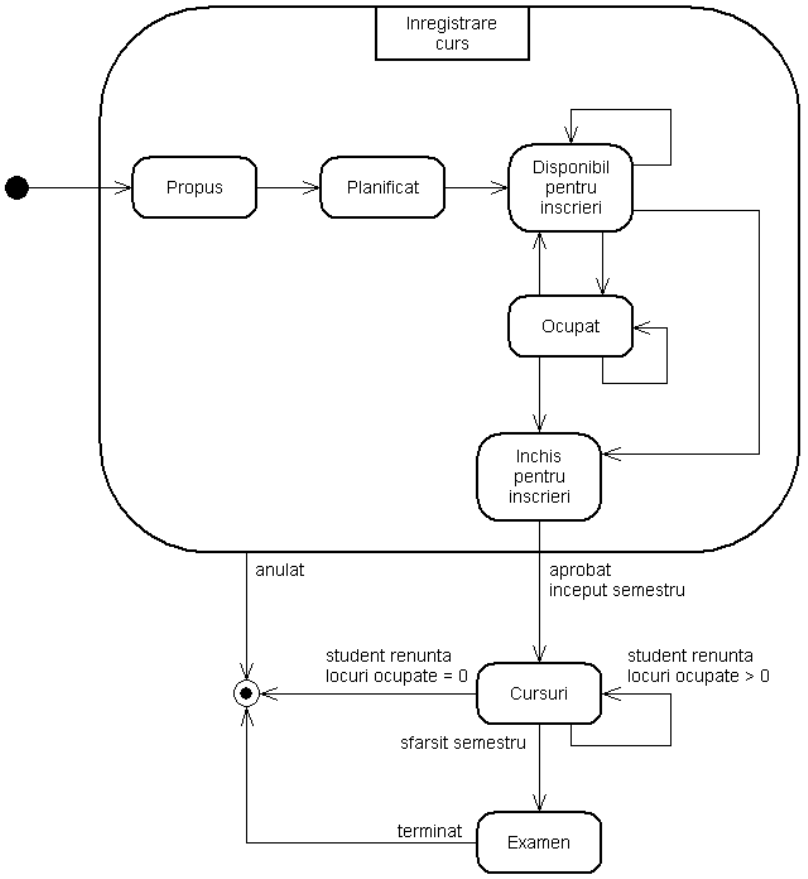
Trebuie etichetata fiecare tranzitie cu un nume, care sa reprezinte numele a cel putin un eveniment care cauzeaza tranzitia intre starile respective. Nu este obligatoriu sa se foloseasca nume unice pentru etichetarea tranzitiilor, deoarece acelasi eveniment poate declansa tranzitia intre mai multe stari diferite. O eticheta a unui eveniment este un nume simbolic.
Tranzitiile sunt etichetate cu urmatoarea sintaxa:

event (arguments)[condition] / action ^ target.sendEvent (arguments)

Doar un eveniment este admis pentru o tranzitie, si o singura actiune pentru un eveniment. Evenimentele, conditiile si actiunile trebuie sa fie incluse in specificatia tranzitiilor.

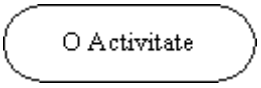
Exemple diagrame de stari:


Propus
anulat
planificat
Planificat
deschis
anulat
Disponibil pentru inscrieri
student inscris
loc disponibil / adauga student
inchis
student inscris
nici un loc disponibil / adauga pe lista de asteptare
loc disponibil
Ocupat
student renunta
loc disponibil / adauga din lista de asteptare
anulat
inchis
Inchis pentru inscrieri
aprobat
anulat

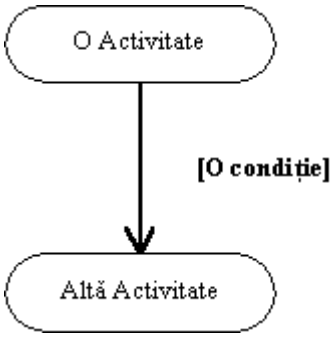

Inregistrare curs
Propus
Planificat
Disponibil pentru inscrieri
Ocupat
Inchis pentru inscrieri
anulat
aprobat
inceput semestru
student renunta
locuri ocupate = 0
Cursuri
student renunta
locuri ocupate > 0
sfarsit semestru
terminat
Examen

8. Diagramele de activitati

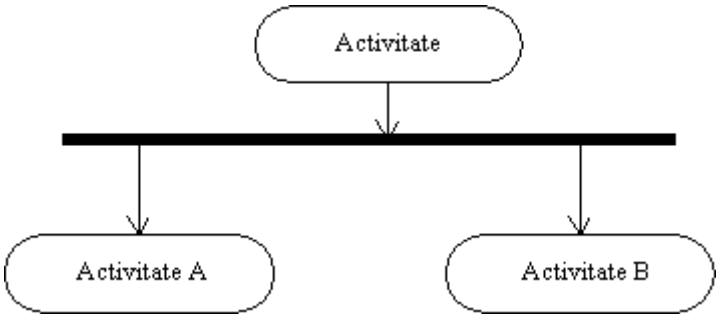
O diagrama de activitati este o varianta a diagramelor de stari organizata în raport cu actiunile, destinata în primul rând reprezentarii comportamentului intern al unei metode (implementarea unei operatii) sau a unui caz de utilizare.
Diagrama de activitati reprezinta starea de executie a unui mecanism, sub forma unei derulari de etape regrupate secvential in ramificatii paralele de fire de executie (fluxuri de control). O diagrama de stari poate reprezenta o astfel de derulare de etape, dar data fiind natura procedurala a implementarii operatiilor, în care cea mai mare parte a evenimentelor corespund pur si simplu sfârsitului activitatii precedente, nu este necesara separarea sistematica a starilor, activitatilor si evenimentelor. Este atunci interesata disponibilitatea unei reprezentari simplificate pentru vizualizarea directa a activitatilor. În acest context, o ***activitate*** apare ca un stereotip de stare. O activitate este reprezentata printr-un dreptunghi rotunjit, ca si starile, dar sub forma de semicerc în lateral.



Fiecare activitate reprezinta o etapa particulara în executia metodei care o inglobeaza. Activitatile sunt legate prin ***tranzitii automate***, reprezentate prin sageti, ca si tranzitiile în diagramele de stari. Atunci când o activitate se termina, tranzitia este declansata si activitatea următoare demareaza. Activitatile nu detin nici tranzitii interne si nici tranzitii declansate de catre evenimente. Tranzitiile între activitati pot fi decorate cu conditii logice, mutual (reciproc) exclusive, care se reprezinta în apropierea tranzitiilor carora le valideaza declansarea.

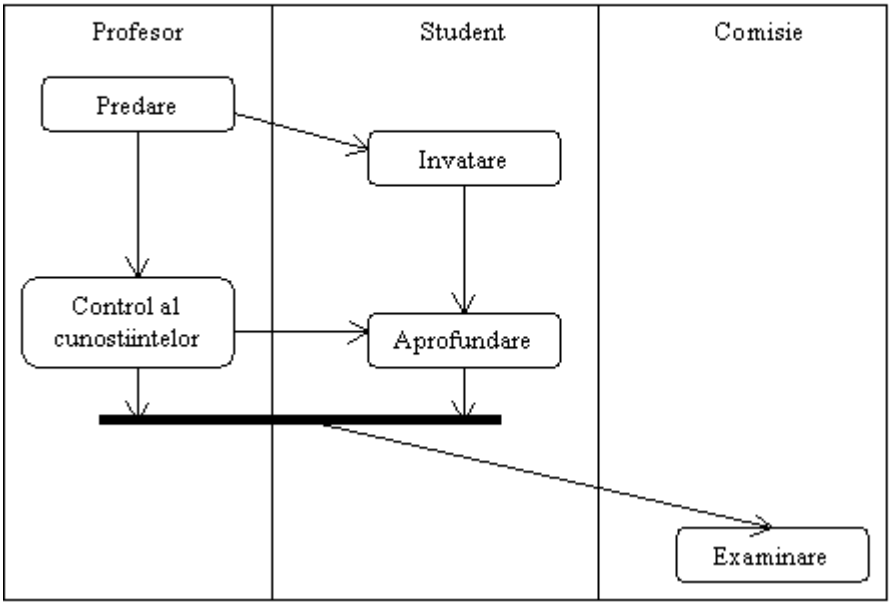


Diagramele de activitati reprezinta sincronizarile între fluxuri de control, prin intermediul ***barelor de sincronizare***. O bara de sincronizare permite deschiderea si închiderea ramificatiilor paralele în interiorul unui fir de executie a unei metode sau al unui caz de utilizare. Tranzitiile care pleaca dintr-o bara de sincronizare sunt declansate simultan. Pe de alta parte, o bara de sincronizare nu poate fi trecuta decât atunci când toate tranzitiile care intra în bara au fost declansate.



Diagramele de activitati pot fi decupate în ***culoare de activitati***, asa cum o piscina este decupata în culoare de natatie, pentru a arata diferitele responsabilitati în cadrul unui mecanism sau al unei organizatii. Fiecare responsabilitate este repartizata uneia sau mai multor obiecte si fiecare activitate este alocata unui culoar dat. Pozitia relativa a culoarelor nu are nici o semnificatie, tranzitiile fiind libere sa tranverseze culoarele la care nu se refera.

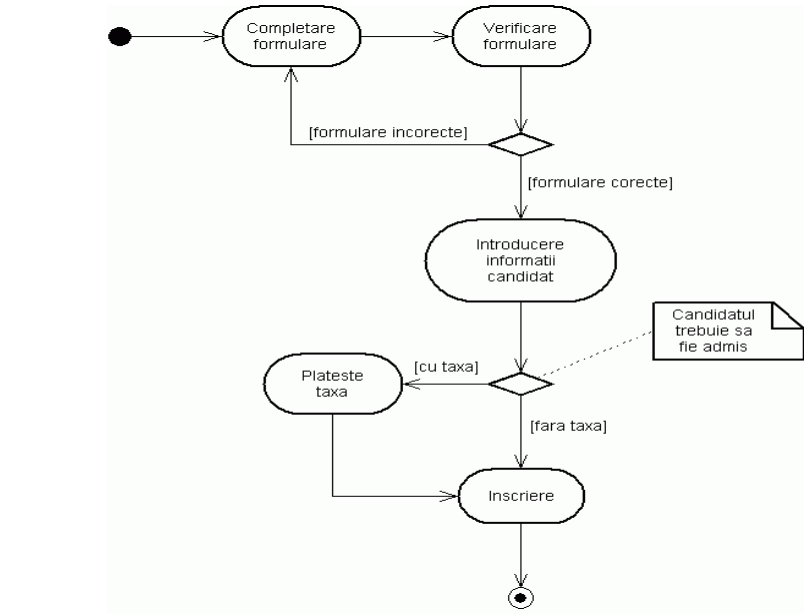
Partitia unei diagrame de activitati in culoare de activitati.

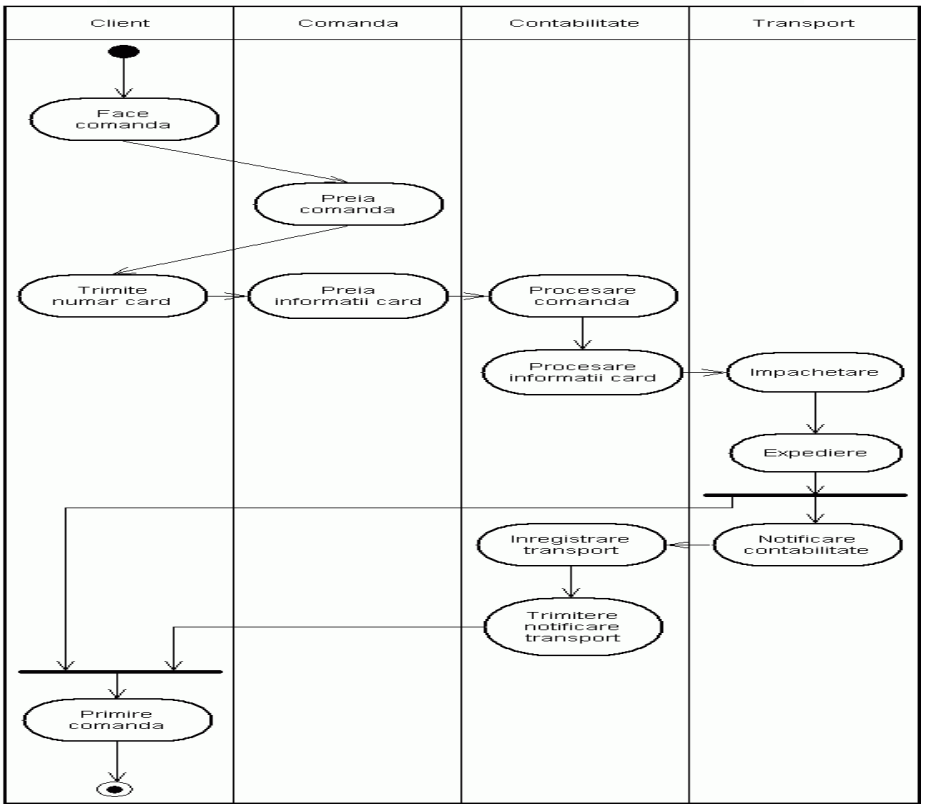


Diagramele de activităţi sunt folosite pentru modelarea proceselor sau a algoritmilor din spatele unui anumit caz de utilizare.
Se folosesc următoarele ***notaţii***: *nodul initial* este un cerc plin este punctul de start al diagramei. Desi nu este obligatoriu, prezenta sa face diagrama mai lizibila. *Nodul final* este un cerc plin înconjurat de un alt cerc. O diagrama poate avea 0, 1 sau mai multe noduri finale. *Activitaţile* sunt reprezentate prin dreptunghiuri rotunjite, iar *fluxurile* prin sagetile diagramei. *Punctul final* al fluxului este reprazentat printr-un cerc cu un X în interior; acesta indica faptul ca procesul se opreste în acest punct. O *ramificatie* (“fork”) se modelează printr-o bara de sincronizare cu un flux de intrare si mai multe fluxuri de iesire. Ea denota începutul unor activitati desfasurate în paralel, iar o *reunire* (“join”) printr-o bara de sincronizare cu mai multe fluxuri de intrare si un flux de iesire, denota sfârsitul prelucrarilor paralele.
O *condiţie* se modelează prin text asociat unui flux, care defineste o conditie care trebuie sa fie adevarata pentru traversarea nodului. *Deciziile* se reprezintă prin romburi cu un flux de

intrare si mai multe fluxuri de iesire; fluxurile de iesire includ conditii. Pentru *îmbinări* (“merge”) se folosesc romburi cu mai multe fluxuri de intrare si un flux de iesire; toate fluxurile de intrare trebuie sa atinga acest punct pentru ca procesul sa continue. O *partitie* (“swimlanes”) e modelată prin o parte a diagramei care indica cine/ce îndeplineste activitatile.

Exemple diagrame de activitati:





9. Diagrame de implementare (componente, deployment)

9.1. Diagrame de componente

Diagramele de componente descriu elementele fizice (hardware) si relatiile lor in mediul de implementare. Diagramele de componente arata optiunile privind implementarea.

Componentele sunt derivate din urmatoarele elemente Booch: programe principale, module, subprograme si procese. Fiecare diagrama de componente descrie modelul din punct de vedere fizic.

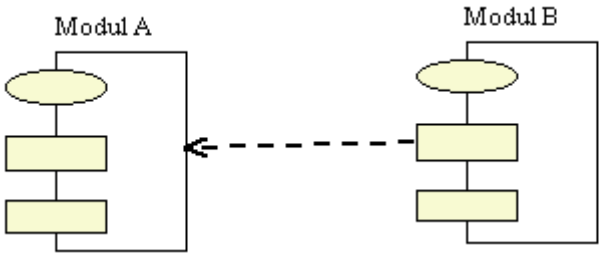
Diagramele de componente contin urmatoarele elemente:
-subsisteme;
-componente:
    -programe principale,
    -module,
    -subprograme,
    -procese;
-dependente.

Se pot creea una sau mai multe diagrame de componente pentru a descrie ***subsistemele*** si ***componentele*** la nivelul de top al modelului curent; unele diagrame de componente sunt continute chiar ele in nivelul de varf (top) al modelului. De asemenea, se pot creea diagrame de componente pentru a arata subsistemele si componentele continute in fiecare subsistem al modelului curent; unele diagrame de componente pot fi chiar ele continute de subsistemele care cuprind subsistemele si componentele care le descriu.
Specificatiile subsistemelor si cele ale componentelor permit prezentarea si modificarea proprietatilor acestora. Informatia in aceste specificatii este prezentata textual, o parte din aceasta informatie putand aparea in simbolurile grafice ale subsistemelor sau componentelor. Daca modificam proprietatile subsistemelor sau componentelor in cadrul specificatiilor acestora, se vor actualiza toate diagramele de componente care includ aceste elemente. Daca modificarile au loc in cadrul unei diagrame, actualizarea se va face in specificatiile subsistemelor sau componentelor sau orice alta diagrama care contine diagrama modificata.

Dependente intre componente

Relatiile de dependenta sunt utilizate in diagramele de componente pentru a indica faptul ca o componenta foloseste serviciile sau facilitatile oferite de alta componenta. Acest tip de dependenta este reflectarea optiunilor de implementare.
Relatia de dependenta poate fi specializata printr-un stereotip pentru a preciza natura optiunilor de implementare care conduc la relatia de dependenta.

Relatiile de dependenta sunt *reprezentate* prin intermediul unei sageti punctate de la utilizator catre furnizor:



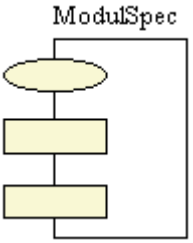
In diagrama de componente, relatiile de dependenta reprezinta in general dependente de compilare, ordinea compilarii fiind data de graful relatiilor de dependenta.

Module

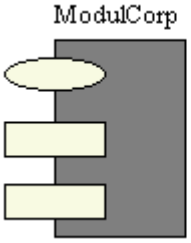
Modulele reprezinta toate tipurile de elemente fizice care intra in constructia aplicatiilor informatice. Ele sunt constituite dintr-un *modul specificatie* (interfata) si un *modul implementare*. Modulul implementare este adesea referit ca un *corp*.
O clasa este declarata in pachet.
Fiecare modul trebuie sa aiba un *nume*. De obicei, numele modulului este un nume simplu de fisier. Modulele cu acelasi nume si de acelasi tip reprezinta aceeasi componenta a modulului, indiferent de diagrama de componente in care apare. De exemplu, doua module specificatie cu acelasi nume reprezinta defapt acelasi modul, in timp ce un modul corp avand acelasi nume cu cele doua de specificatie reprezinta o componenta a modelului separata.

Reprezentarea modulelor

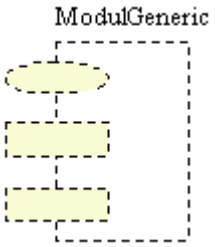
Module specificatie: In C++, se pot folosi pachetul specificatie pentru a reprezenta un fisier *.h:



Module corp: In C++ se pot folosi pachetele corp pentru a reprezenta un fisier *.cpp



Module generice:



Procese

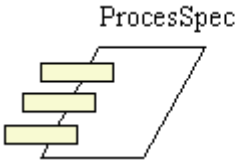
Procesele corespund componentelor care detin propiul lor flux de control. Daca procesele sunt compilate diferit decat modulele obisnuite, se poate aloca definirea unei clase, unui proces. Procesele sunt entitati dinamice in evolutie. Ele se constituie printr-o entitate program idependenta formata din instructiuni si una de distributie si executie cu contextul propriu ce implica un procesor si un mediu propriu (memorie, periferice, etc.).

Ca si celelalte elemente ale diagramei de componente si procesele sunt: procese specificatie si procese corp.

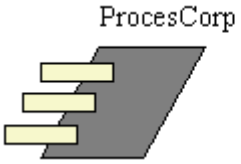
Fiecare proces trebui sa aiba un *nume*. In general, numele unui proces este un simplu nume de fisier. Procesele cu acelasi nume si acelasi tip reprezinta aceeasi componenta din model indiferent de diagrama de componente in care apare. De exemplu, doua procese specificatie cu acelasi nume reprezinta acelasi proces specificatie, in schimb ce un proces corp avand de asemenea acelasi nume reprezinta o componenta separata din model.

Reprezentare grafica

Procese specificatie:



Procese corp:

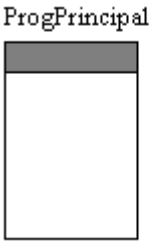


Programe principale

Programul principal reprezinta un fisier ce contine radacina programului. De exemplu, in C++ acesta reprezinta un fisier .cpp care contine definitia functiei ***main***. In mod normal se identifica doar un program principal intr-un program.

Fiecare program principal trebuie sa aiba un nume unic in model. De obicei, numele programului principal este un nume simplu de fisier. Programele cu acelasi nume reprezinta acelasi program principal, indiferent de diagrama de componente in care apare.

Grafic programul principal se reprezinta in modul urmator:



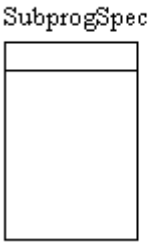
Subprograme

Subprogramele regrupeaza procedurile si functiile care nu apartin claselor in C++. Aceste componente pot contine declaratii de tipuri de baza necesare pentru compilarea subprogramelor. In schimb, ele nu contin niciodata clase.

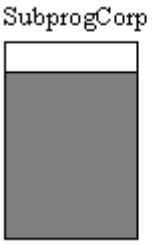
Fiecare subprogram trebuie etichetat cu un nume. In general, numele unui subprogram este un simplu nume de fisier. Subprogramele ce au acelasi nume si sunt de acelasi tip reprezinta aceeasi componenta in modelul respectiv, indiferent de diagrama de componente in care apare. De exemplu, daca avem doua subprograme specificatie cu acelasi nume atunci ele reprezinta acelasi subprogram specificatie; dar daca avem si un subprogram corp cu acelasi nume cu cele doua subprograme specificatie, subprogramul corp reprezinta o componenta a modelului diferita.

Exista doua reprezentari grafice: una pentru specificatia subprogramelor si alta pentru implementarea lor.

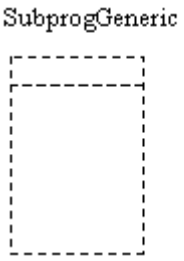
Subrograme specificatie:



Subprograme corp:



Subprograme generice:



Subsisteme

Pentru a usura implementarea aplicatiilor, diferitele componente pot fi grupate in pachete conform unui criteriu logic. Ele sunt adesea stereotipizate in subsisteme pentru a adauga notiunile de biblioteca de compilare si de management al configuratiei, intelesului de partitie asociat deja impachetarii. Subsistemele au pentru componente acelasi rol ca si categoriile pentru clase.

Subsistemele permit partitionarea modelului fizic al sistemului. Orice subsistem poate contine module si alte subsisteme. Prin conventie, orice modul continut intr-un subsistem este public, doar daca nu se defineste explicit restrictionarea accesului.

Un subsistem poate avea dependente cu alte subsisteme si module; un modul poate si el avea dependente cu alte module si subsisteme.

Orice subsistem trebuie sa aiba un nume unic in model. De obicei, numele subsistemului este numele unui fisier sistem. Subsistemele cu acelasi nume reprezinta acelasi subsistem, indiferent de diagrama de componente in care apare.

Subsistemul se reprezinta grafic sub forma unui pliant:

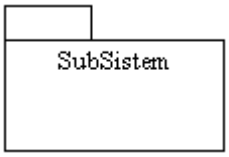
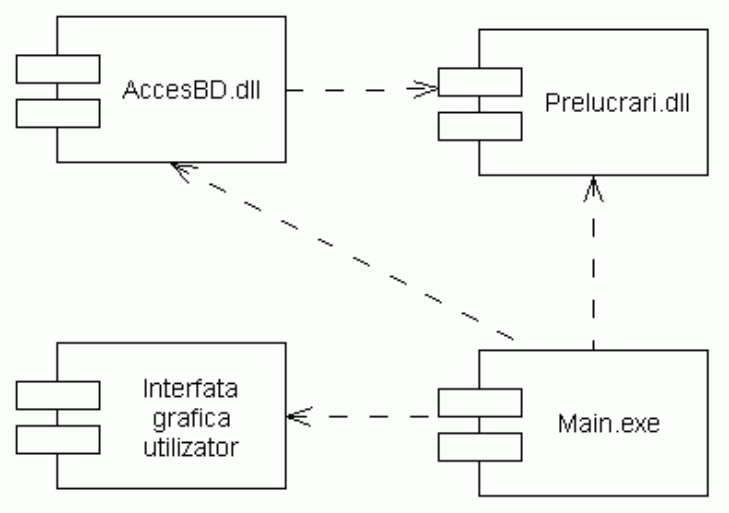


Diagrama de componente este asemanatoare cu diagrama pachetelor, permitând vizualizarea modului în care sistemul este divizat si a dependentelor dintre module. Diagrama componentelor pune însa accentul pe elementele software fizice (fisiere, biblioteci, executabile) si nu pe elementele logice, ca în cazul pachetelor.

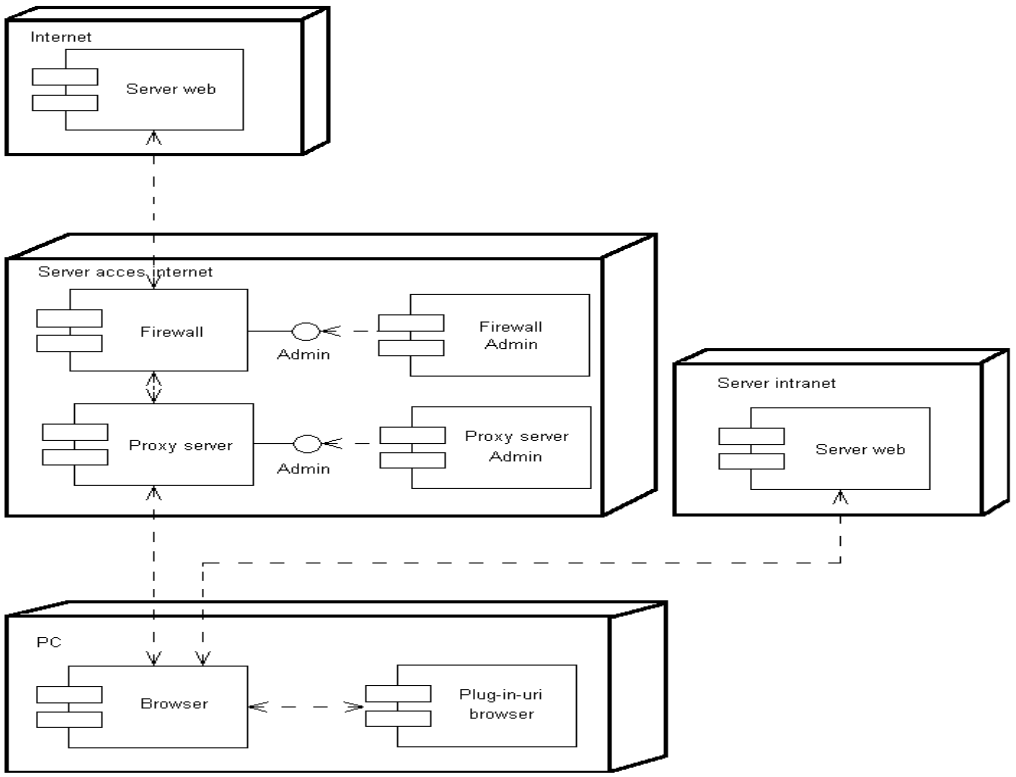
Exemplu



9.2. Diagrame de deployment (punere in functiune-lansare-constructie)

Diagramele de lansare -constructie (“deployment diagrams”) descriu configuratia elementelor de prelucrare la run-time si componentele software, procesele si obiectele care se executa pe ele. Aceste diagrame sunt grafuri de noduri, conectate de asociatii de comunicare. Nodurile pot contine instante ale componentelor, indicând faptul ca acea componenta ruleaza sau se executa în nodul respectiv. Aceste noduri sunt reprezentate prin paralelipipede; cercurile reprezinta interfete.

Exemplu diagrama:



Diagramele de constructie prezinta dispunerea fizica a diferitelor elemente hardware (noduri) care intra in componenta unui sistem si repartizarea programelor executabile pe aceste elemente. In diagramele de constructie se indica nodurile (procesoare si dispozitive) si conexiunile unui model. Fiecare model contine o singura diagrama de constructie in care se arata conexiunile intre noduri, si alocarea proceselor de catre noduri.

Specificatiile nodurilor si specificatiile conexiunilor permit prezentarea si modificarea proprietatilor componentelor respective din model. Informatia intr-o specificatie este prezentata textual, si o parte din aceste informatii pot aparea in cadrul simbolurilor grafice ce reprezinta aceste componente in diagrama de constructie.

Daca se modifica proprietatile unui nod sau ale unei conexiuni in cadrul specificatiei, diagrama de constructie se actualizeaza reflectand aceste modificari. Daca modificarile se realizeaza in cadrul diagramei, actualizarea se face in cadrul specificatiilor componentelor.

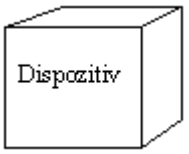
Noduri

Nodurile reprezinta orice componenta hardware prezenta in sistem, ele constituind echipamentul sistemului. Diferitele noduri care apar in diagrama de constructie sunt legate intre ele prin conexiuni care simbolizeaza un suport de comunicatie apriori bidirectional. Nodurile dupa natura lor pot fi: dispozitive sau procesoare.

Dispozitive

Un dispozitiv este o componenta hardware care, in sistemul din care face parte, nu are putere de calcul. Fiecare dispozitiv trebuie sa aiba un nume; acesta poate fi generic cum ar fi de exemplu "modem" sau "terminal"

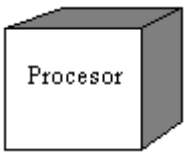
Un dispozitiv se reprezinta grafic printr-un cub:



Procesor

Un procesor este o componenta hardware capabila de executia programelor. Orice procesor trebuie etichetat cu un nume. Nodurile care corespund procesoarelor din punct de vedere al aplicatiei poarta numele proceselor pe care le gazduiesc. Fiecare proces cu nume in diagrama de constructie executa un program principal cu acelasi nume, descris in diagrama de componente.

Procesorul are aceeasi reprezentare grafica cu a dispozitivelor:



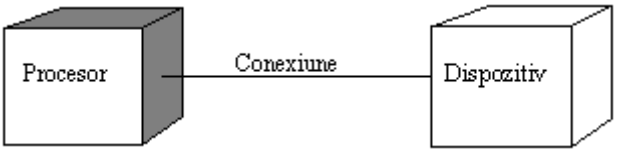
Distinctia intre un dispozitiv si un procesor depinde puternic de punctul de vedere asupra acestora. Un terminal X va fi vazut ca un dispozitiv de catre utilizatorul terminalului, si ca un procesor pentru un instrument software care se executa pe procesorul terminalului X.

Conexiuni

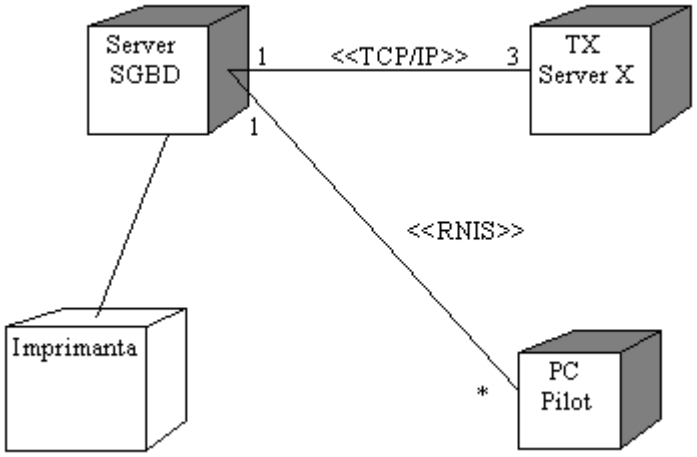
O conexiune reprezinta un tip de hardware care realizeaza cuplarea intre doua entitati. O entitate este de obiecei un nod (procesor sau un dispozitiv). Cuplarea hardware-ului se poate face direct, cum ar fi un cablu, sau indirect, ca de exemplu comunicarea prin satelit.

Conexiunile sunt de obicei bidirectionale. Optional ele pot fi etichetate cu un nume.

Conexiunile se reprezinta printr-o linie plasata intre componentele de cuplat:

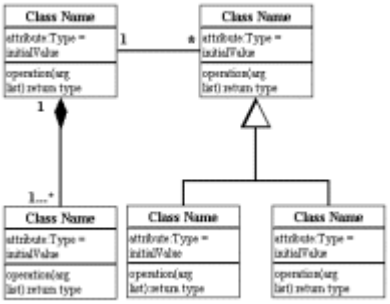


Exemplu diagrame de constructie:

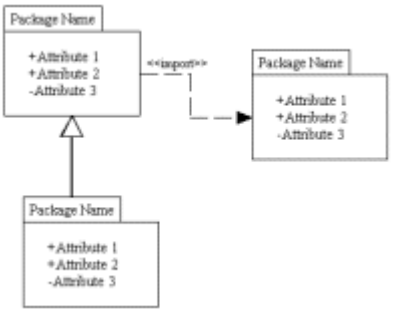


10. Concluzii

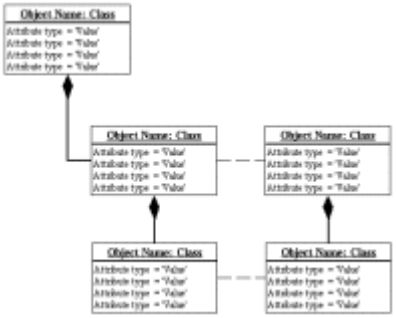
UML este un limbaj pentru specificarea, vizualizarea, construirea si documentarea elementelor sistemelor software. Este un standard de facto pentru modelarea software. UML permite modelarea cazurilor de utilizare si reprezentarea diagramelor de clase, de interactiune, de activitati, de stari, de pachete si de implementare. O sinteza a principalelor diagrame UML este prezentata in continuare.



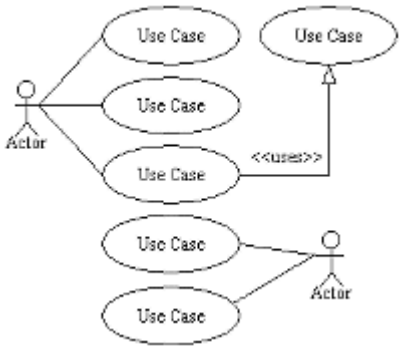
Class Diagrams
Class diagrams are the backbone of almost every object oriented method, including UML. They describe the static structure of a system.



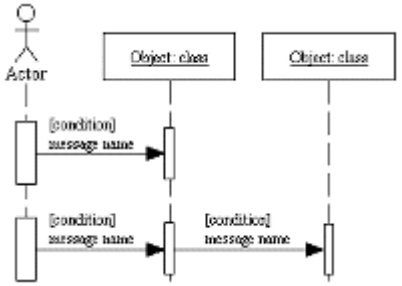
Package Diagrams Package diagrams are a subset of class diagrams, but developers sometimes treat them as a separate technique. Package diagrams organize elements of a system into related groups to minimize dependencies between packages.



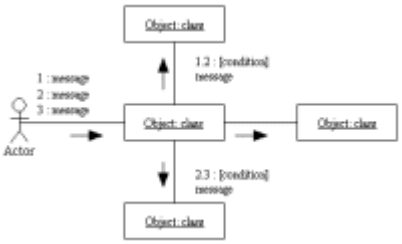
Object Diagrams
Object diagrams describe the static structure of a system at a particular time. They can be used to test class diagrams for accuracy.



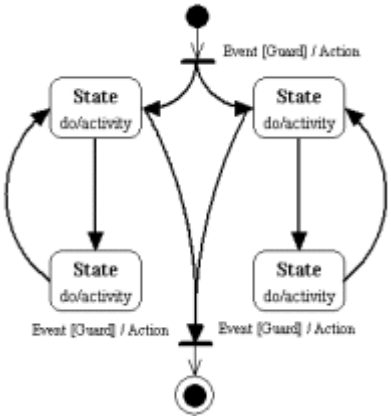
Use Case Diagrams
Use case diagrams model the functionality of system using actors and use cases.



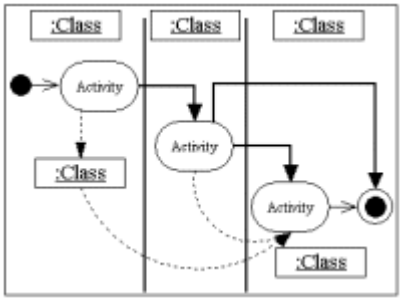
Sequence Diagrams
Sequence diagrams describe interactions among classes in terms of an exchange of messages over time.



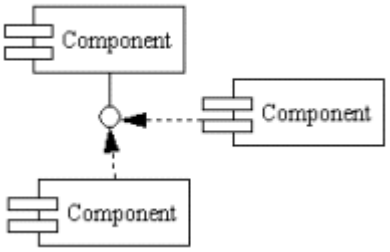
Collaboration Diagrams
Collaboration diagrams represent interactions between objects as a series of sequenced messages. Collaboration diagrams describe both the static structure and the dynamic behavior of a system.



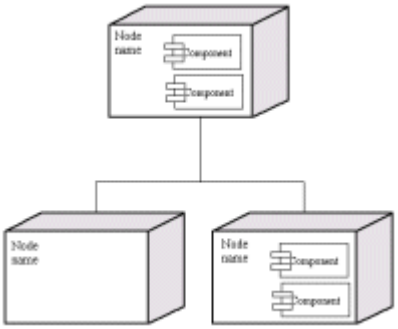
Statechart Diagrams
Statechart diagrams describe the dynamic behavior of a system in response to external stimuli. Statechart diagrams are especially useful in modeling reactive objects whose states are triggered by specific events.



Activity Diagrams
Activity diagrams illustrate the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Typically, activity diagrams are used to model workflow or business processes and internal operation.



Component Diagrams
Component diagrams describe the organization of physical software components, including source code, run-time (binary) code, and executables.



Deployment Diagrams
Deployment diagrams depict the physical resources in a system, including nodes, components, and connections.

11. Anexa: Exemplu de caz pentru utilizarea diagramelor UML

0. Introduction

The aim of this tutorial is to show how to use UML in "real" software development environment.

1. Elevator Problem

A product is to be installed to control elevators in a building with m floors. The problem concerns the logic required to move elevators between floors according to the following constraints:

- Each elevator has a set of ***m*** buttons, one for each floor. These illuminate when pressed and cause the elevator to visit the corresponding floor. The illumination is canceled when the elevator visits the corresponding floor.
- Each floor, except the first floor and top floor has two buttons, one to request and up-elevator and one to request a down-elevator. These buttons illuminate when pressed. The illumination is canceled when an elevator visits the floor and then moves in the desired direction.
- When an elevator has no requests, it remains at its current floor with its doors closed.

2. Unified Modeling Language

UML is a modeling language that only specifies semantics and notation but no process is currently defined. Thus, we decided to do the analysis as follows;

- Use Case Diagram
- Class Diagram
- Sequence Diagram
- Collabration Diagram
- State Diagram

3. Analysis

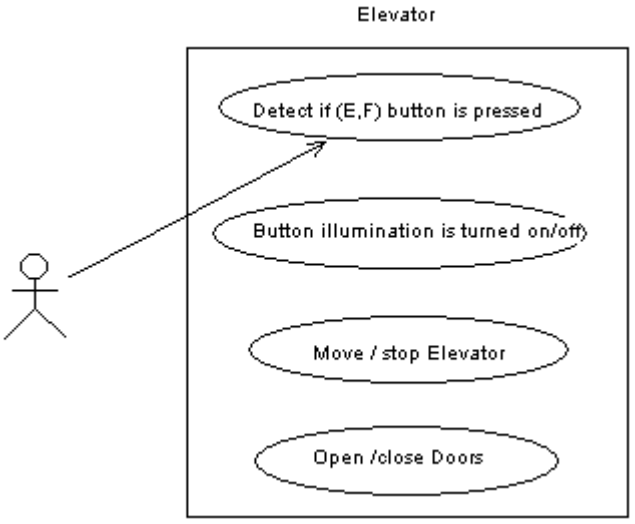
3.1. Use case diagram

Use case description:

- A generalized description of how a system will be used.

- Provides an overview of the intended functionality of the system.
- Understandable by laymen as well as professionals.

Use Case Diagram:



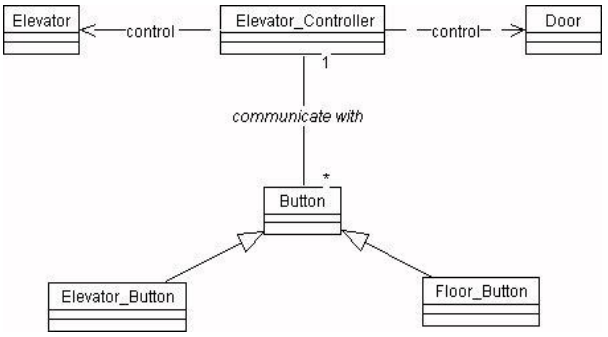
Elevator basic scenario that can be extracted from Use Case Diagram:

- Passenger pressed floor button
- Elevator system detects floor button pressed
- Elevator moves to the floor
- Elevator doors open
- Passenger gets in and presses elevator button
- Elevator doors closes
- Elevator moves to required floor
- Elevator doors open
- Passenger gets out
- Elevator doors closes

3.2. Class Diagram

Class diagrams show the static structure of the object, their internal structure, and their relationships.

Class diagram:



3.3. State diagram

A state diagram shows the sequences of states an object goes through during it's life cycle in response to stimuli, together with its responses and actions.

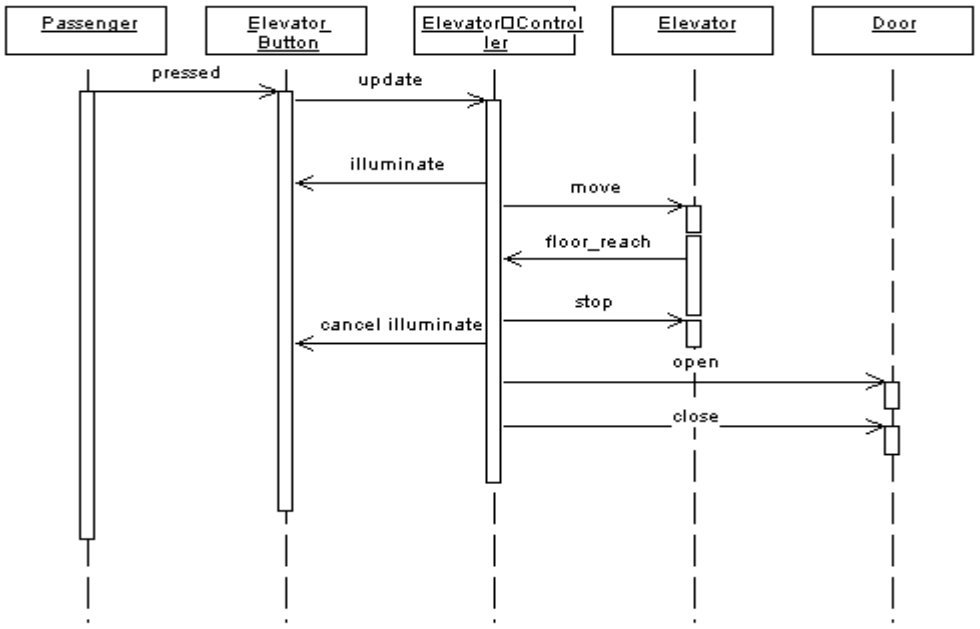
4. Design

The design phase should produce the detailed class diagrams, collaboration diagrams, sequence diagrams, state diagrams, and activity diagram. However, the elevator problem is too simple for an activity diagram. Thus, we are not using an activity diagram for the elevator problem.

4.1. Sequence Diagram

A sequence diagram and collaboration diagram conveys similar information but expressed in different ways. A Sequence diagram shows the explicit sequence of messages suitable for modeling a real-time system, whereas a collobration diagram shows the relationships between objects.

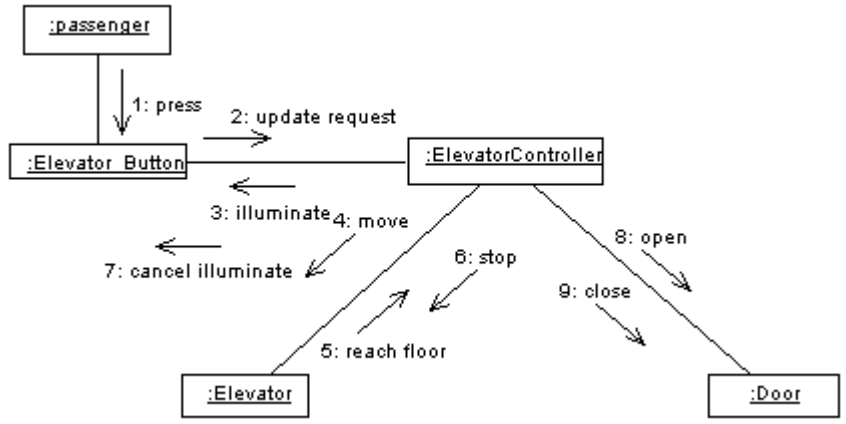
Sequence Diagrams:



4.2. Collaboration diagram

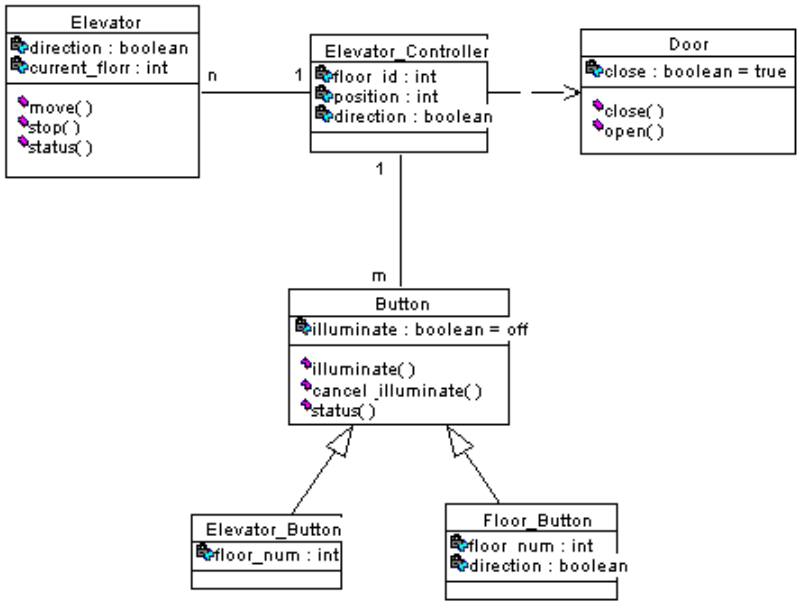
- Describes the set of interactions between classes or types
- Shows the relationships among objects

Collabration diagrams:



5. Detail Design

5.1. Detail Class Diagram



5.2. Detail Operation Description

Module Name	Elevator_Control::Elevator_control_loop
Module Type	Method
Input Argument	None
Output Argument	None

Error Message	None
File Access	None
File Change	None
Method Invoke	button::illuminate, button::cancel_illumination, door::open, door::close, elevator::move, elevator::stop
Narative	

5.3. Pseudo-Code

```
void elevator_control (void)
{
  while a button has been pressed
      if button not on
      {
          button::illuminate;
          update request list;
      }
      else if elevator is moving up
      {
           if there is no request to stop at floor f
              Elevator::move one floor up;
           else

}
```