BULETINUL Universităţii Petrol – Gaze din Ploieşti	Vol. LIX No. 1/2007	75 - 104	Seria Ştiinţe Economice

Cycle: UML FOR MANAGERS (V)
The Class Diagram (continuation)

Gabriel Irinel Marcu, Liviu Dumitraşcu

Universitatea Petrol-Gaze din Ploieşti, Bd. Bucureşti 39, Ploieşti
email: gimarcu@upg-ploiesti.ro, ldumitrascu@upg-ploiesti.ro

Abstract

This paper continues description of UML class diagram wich represents the structure of an object-oriented application. The present paper describes the class relations of association, aggregation, compounding, dependency and generalisation/specialisation. Also, presents the concepts of navigability, multiplicity, association and qualified association.

Key-words: *object, class, class diagram, object diagram, encapsulation, polymorphism, abstracting, attribute, methods, operations, association, aggregation, compounding, dependency, navigability, multiplicity, association class, qualified association, generalisation/specialisation, class models, derived attributes, static attributes, static operations, abstract method.*

Class Relations

Classes are basic elements of the class diagram. An application obviously demands for the modelling of more classes. Isolated classes will never allow for the modelling of a system and of the different interactions between its components.

UML proposes more solutions for the connection between classes.

Class relations express the semantic or structural relations. The most frequently used relations are: *association*, *aggregation*, *compounding*, *dependency* and *inheritance*.

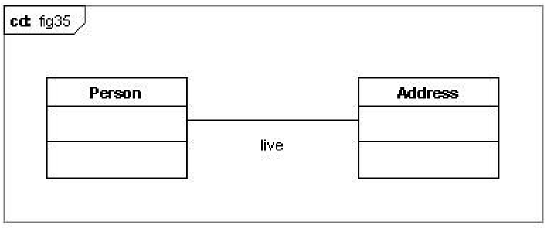
Remark. Even if the relations are described within the class diagrams, they express in the majority of the cases, the connections between objects. Due to this fact, the binary relations between classes can generate the intervention of more objects within each class. The notion of multiplicity allows for the control of the number of objects controlling each instance of the relation.

Association

UML language defines the concept of association between two classes. This concept is very interesting and it does not belong to the elementary concepts of the object modelling and it also allows for the definition of the relations between more objects.

Within UML an association is made between two classes. It has a name and two extremities that allow for the plugging to each of the associated classes. When an association is defined between two classes it means that the instantiated objects of the two classes can be connected between them.

Figure 35



The name of the association, more than once corresponds to a verb in the infinitive with a supplementary observation (a full triangle) of the sense of the reading if ambiguity might occur. Sometimes, a stereotype better characterizing association replaces the name.

Figure 35 displays the association called lives which *associates* the classes `Person` and `Address`. This association signifies the fact that the instance objects of the class `Person` and the instance objects of the class `Address` can be connected between them. In other words, this signifies the fact that *persons* live at *addresses*.

Each extremity of the association indicates the role of the class within the relation and it specifies the number of objects involved in the association (figure 36).

Figure 36

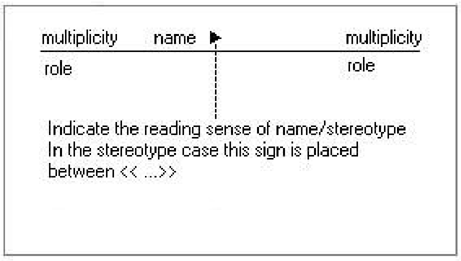
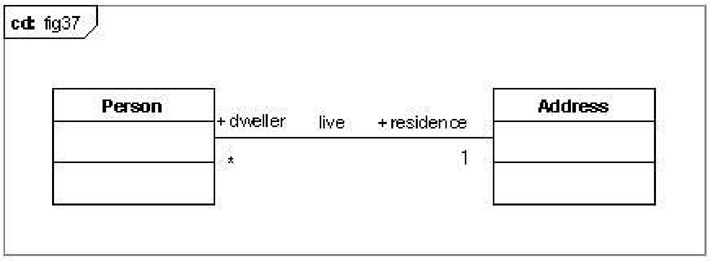

multiplicity
name ▶
multiplicity
role
role
Indicate the reading sense of name/stereotype
In the stereotype case this sign is placed
between << ...>>

Figure 37 represents the same association (figure 34) where we have stated the names of the roles for each class and the minimum and the maximum number of objects before being connected. In the context of this association, the class `Person` represents he inhabitant while the class `Address` represents residency. In other words, we can say that this association signifies the fact that the persons live at addresses and that they are the inhabitants of those premises.

Figure 37



Remark. The diagram in figure 37 has to be read as follows:

- Premises 1: for one inhabitant there is a minimum of one premises and a maximum of one residency.
- Inhabitant *: for one premises there is minimum 0 inhabitants and a maximum of an infinite number of inhabitants.

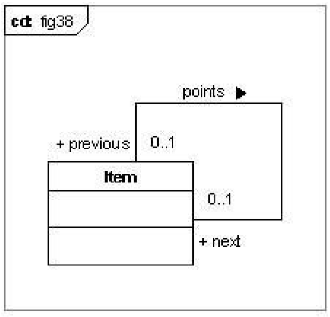
Within UML it is possible for one association to be able or not to express navigability.

The most frequently used reflexive association is the binary one (connecting two classes). Not in few cases, the two extremities of the association lead to the same class. In this case, the association is reflexive.

The reflexive association between classes has as a main function the structuring of objects belonging to the same class.

Figure 38 illustrates an example of reflexive association for the elements of a list.

Figure 38



Remarks:

- The association `points` connects the class Element to itself by means of a reflexive association.
- An element has maximum a precedent and a follower. The corresponding cardinality of the two extremities of the association is the following: `0..1`.

The Relation of Aggregation

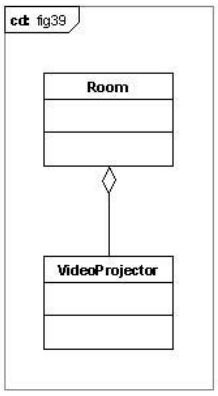
Aggregation constitutes a type of relation between two classes which is stronger than an association. As opposed to association, the aggregation involves a notion of ownership between objects if, obviously, these objects can be connected (for example, the destruction of one can lead to the destruction of the other).

An aggregation is a particular form of association. It represents the relation of structural or behavioural inclusion of an element within a group. Contrary to association, aggregation is a transitive relation.

An aggregation is represented by means of a void rhomboid placed close to the owner class and eventually with an arrow towards the possessed class, the connection between the two being materialised by means of a continuous line.

Figure 39 illustrates an aggregation relation between a class called `Room` and a class called `Video-projector`. In each course room there is one video-projector placed on the ceiling.

Figure 39

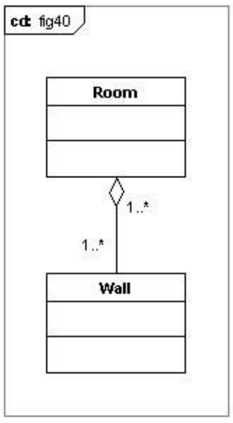


Remark. An instance of the class `VideoProjector` can be protected by more lecture rooms.

Like in the case of a relation of association, it is possible for one to mention certain navigability and a certain multiplicity on the aggregation line.

Figure 40 models the following sentence by means of an aggregation relation in UML: <<*A room has walls*>>.

Figure 40



Remarks:

- A wall can belong to two contiguous rooms.
- A room contains at least a (circular!) wall.

The Relation of Compounding

Compounding, also called <<*composite aggregation*>> is a particular aggregation. This means that all compounding may be replaced by means of an aggregation which, in its turn, may be replaced through an association. The only consequence is the information loss.

The compounding relation describes a structural content between instances. This implies the idea that, particularly, the composite element is responsible for the creation and the destruction of its elements. As a consequence, the destruction of the composite object automatically involves the destruction of its components. An instance of the part always belongs to maximum one instance of the composite element.

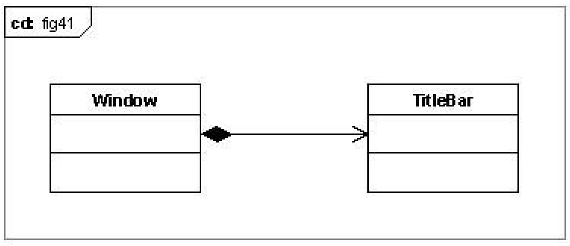
Compounding represents the strongest type of the relation between classes; it is used in order to indicate complete possession of a class towards another. At a given moment, the possessed class can only be involved into a single compounding relation. The destiny of the instances of the classes associated by means of compounding is always connected; if the owner class is destroyed, the same thing will happen to the possessed class as well.

A relation of compounding can be interpreted as a relation of the type <<*…is part of…*>>. A compounding must be read from the class possessed to the possessing class. For example, let us imagine that a *`Window`* must possess (contain) a title bar. This relation can be represented by means of a class *`TitleBar`* which <<*…is part of…*>> the class *`Window`*.

A relation of compounding is represented by means of a plane rhomboid placed close to the possessing class and eventually with an arrow in the direction of the possessed class. The multiplicity next to the possessing class can take only two values: 0 or 1. Particularly, multiplicity is 1.

Figure 41 presents an example of compounding between a class called *`Window`* and a class called *`TitleBar`*.

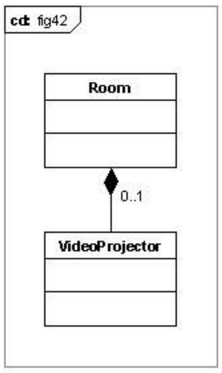
Figure 41



As for an association relation, it is possible to maintain navigability and multiplicity on the compounding line.

Figure 42 illustrates a compounding relation between the class *`Room`* and the class *`VideoProjector`* with the observation that there is one projector in each room, attached to the ceiling.

Figure 42



Remark. As the compounding relation is structural, UML language authorises the imbricate presentation as well (figure 43).

Figure 43

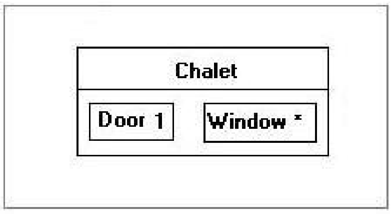


Figure 44 illustrates the compounding relation between the class *ShopCart* and the class *ShopCartLine*. A line of the basket can belong only to one single basket, the latter being determined by the totality of its lines.

The destruction of the basket triggers the automatic destruction of all its lines.

Figure 44

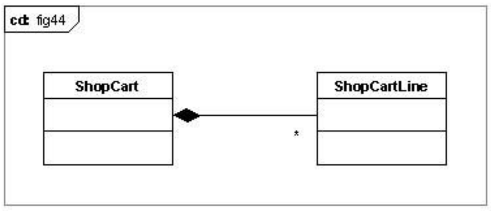


Figure 45 illustrates the compounding relation between the class *Client* and the classes *BankCard* and *Address*.

Figure 45

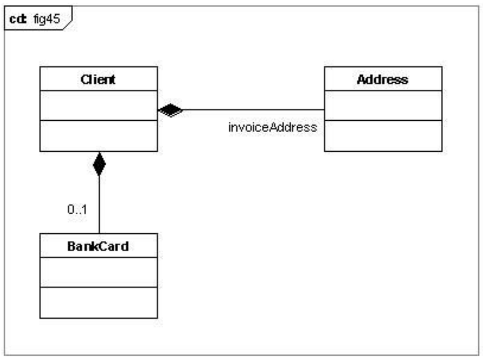


Figure 46 illustrates the compounding relation between the class *MaterialsCatalogue* and *Material* for the application <<*Material Management*>>.

Figure 46

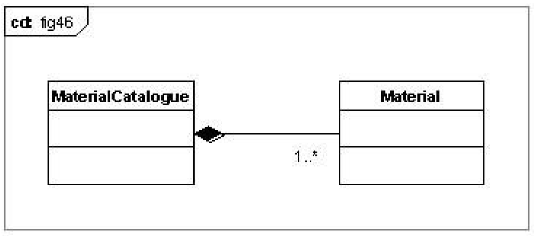


Figure 47 models the structure of an email address.

Figure 47

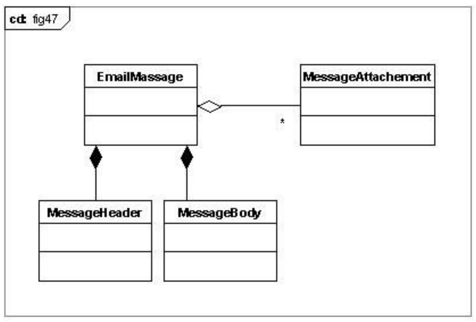
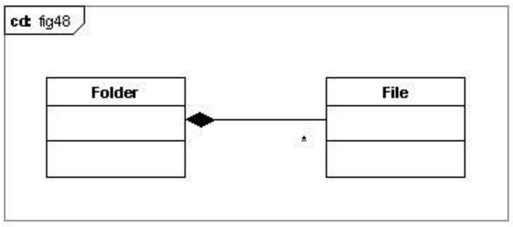


Figure 48 illustrates the compounding relation between the classes `Folder` and `File`.

Figure 48



The Relation of Dependency

A relation of dependency is a unidirectional relation expressing a semantic dependency between the elements of a model. It is represented by means of an oriented discontinuous line.

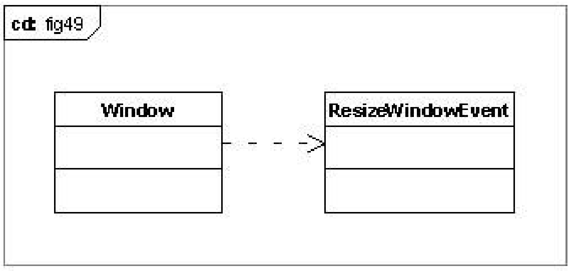
Dependency means the weakest form of a class relation.

A dependency between two classes signifies the fact that one of the two uses the other under one way or another.

A dependency can be interpreted as a relation of the following type: <<*…uses a…*>>

Figure 49 illustrates a dependency between the class *`Window`* and another class called *`ResizeWindowEvent`*.

Figure 49



Remark. Figure 49 allows us to see that the `Window` does not share a durable interaction with `ResizeWindowEvent`.

Navigability

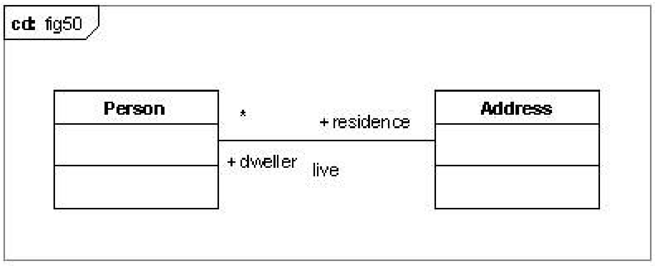
For associations, one can additionally model navigability. In order to mention the fact that one can navigate from one class to another, we shall use an arrow in the direction of the class towards which we can navigate. (see figure 50).

If one can navigate in two directions, we shall not mention any arrow. As the UML specifications state it, eliminating the arrows does not allow for the differentiation of the non-navigable associations from those with double navigability.

Remarks:

- If an extremity is navigable, then the object can navigate towards the object it is connected to and thus one can obtain the values of its properties or perform the operations it is responsible for.
- In figure 50, the inhabitants can navigate towards their residency (and not the other way round) which allows for the obtaining of, for example, the number of the street.

Figure 50

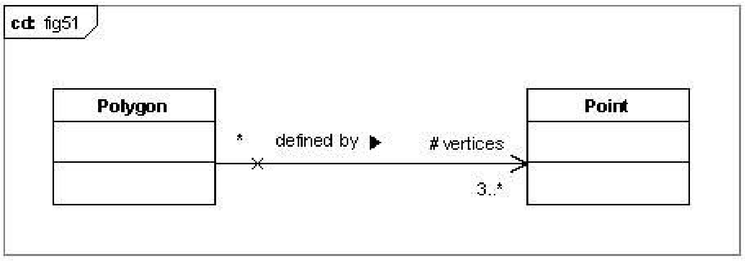


Explicitly, one can forbid any navigation to a class towards another class, placing a small ‘x’ on the association line, next to the class towards which navigability is not possible.

Figure 51 displays an association towards a class with the name `Polygon` and a class with the name `Point`.

Because it is impossible to navigate from an instance of the class `Polygon` towards an instance of the class `Point`, we shall place one navigability arrow towards `Point` as well as a small ‘x’ on the association line.

Figure 51



Remark. The polygon is defined by means of a group of points playing the role of crowns. The crowns of the polygon are not accessible but through the `Polygon` class and its descendents.

Figure 52 displays an example of modelling navigation by means of attributes which reiterate the oriented association example illustrated in figure 52 [3].

Figure 52

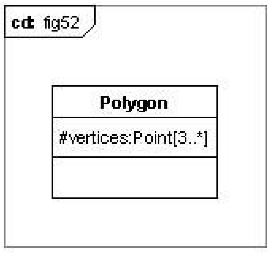
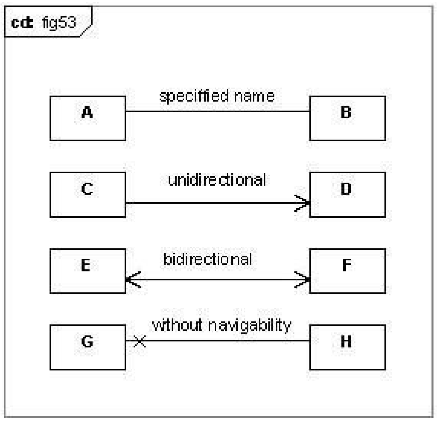


Figure 53 synthetically presents the navigability types allowed within UML.

Figure 53



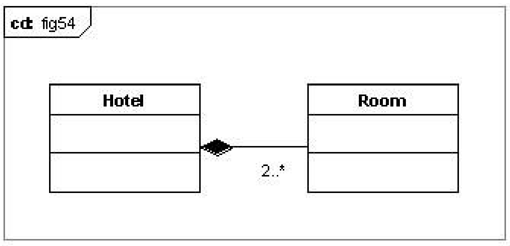
Multiplicity

Multiplicity indicates the number of objects susceptible of taking part into a given association.

The main multiplicities are: <<*more*>>(*); <<*exact n*>>(n); <<*minimum n*>> (n..*); <<*between n and m*>>(n, m). It is also possible to have convex intervals: $n_1..n_2$; $n_3..n_4$ or $n_1..n_2$, n_5, n_6…* etc. If no value is specified, then multiplicity has the implicit value 1. Then, when multiplicity is used for an association, one must not place the values of the multiplicity between accolades.

Figure 54 presents an association explicitly mentioning a multiplicity: a hotel resides at least in two rooms.

Figure 54



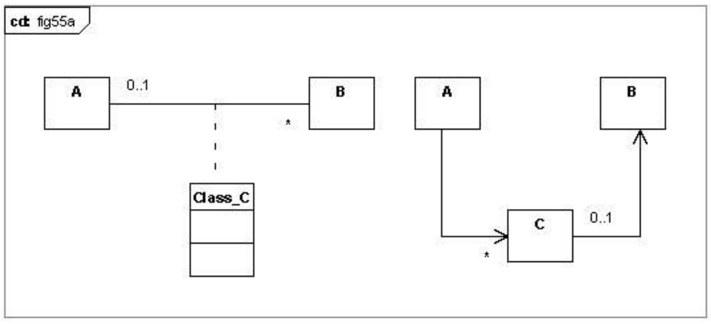
Association Class

The association class is a programmed link reaching the class level. It possesses both the characteristics of an association and those of a class and it can thus contain attributes valorised for each connection.

An association class possesses a name and attributes like any normal class. An association class is presented in the same way as the traditional class but a discontinuous line connects it to the association it represents.

An association class possesses the properties of a class and of an association. The advantage of such a modelling is that the initial association between the two classes is a lot more visible within the model. Each association class can be transformed into two associations and an autonomous class (figure 55 [1]).

Figure 55



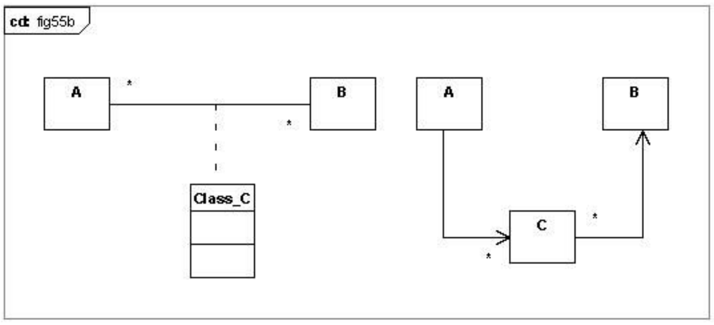
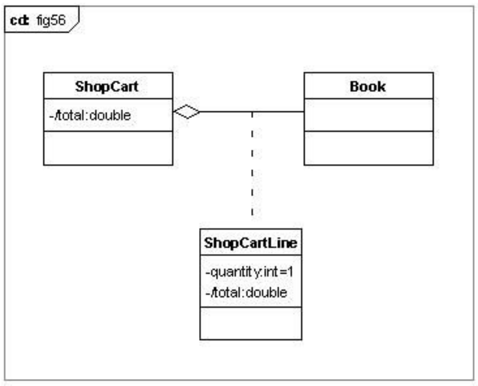


Figure 56 presents an example of association class regarding the basket administration (the II-nd solution, Application for the administration of a virtual shop selling books).

Figure 56

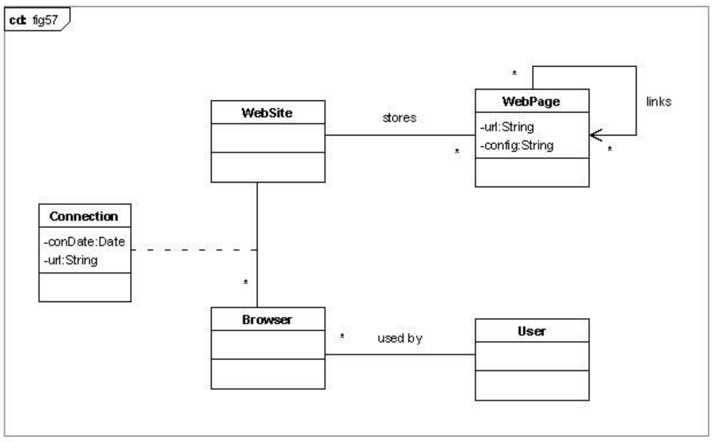


Remarks:

- `ShopCartLine` is associated to the relation between the `ShopCart` and the `Book` so that an object `ShopCart` connected to an object `Book` can accept as attachment an instance `ShopCartLine` containing the attributes `quantity` and stating how many books of the same type the client intends to buy. The attribute `quantity` is implicitly positioned on 1.
- The aggregation represented in the figure clearly expresses the fact that an object `ShopCart` contains more objects `Book` (the aggregation expresses a relation of the type <<*…contains…*>> , <<*…it is made of…*>> and for this reason it is not necessary to nominate it.
- The attribute `/total` of the class `ShopCartLine` expresses the total cost of the items which are to be ordered by the client for a book and the attribute `/total` of the class `ShopCart` mentions the overall cost of the basket calculated by putting together all the lines in the basket.
- Each connection between the basket and a book contains a value of the attribute, quantity representing one line of the basket.

Figure 57 models a connection between a navigator (browser) and a web site by means of an association class (connection).

Figure 57



Qualified Association

Sometimes, the association between two classes grants little information regarding the actual involvement of the classes within the relation which makes the modelling imprecise. This imprecision particularly refers to the multiplicity of an extremity of the association and/or the duration of the association compared to the associated classes.

Qualifying an association in various cases allows for the transformation of an undetermined or infinite multiplicity into a defined multiplicity.

Qualifying an instance represents a value or a group of values allowing for the finding of such an instance. A qualification is often an index, such as for example a key to retrieve an article within a relational database. This qualification is then introduced at a different extremity as one or more attributes (figure 58) where the instances of class 2 are qualified at the level of class 1. The qualifier represented as a small rectangle is placed between the association and the source element.

Figure 58

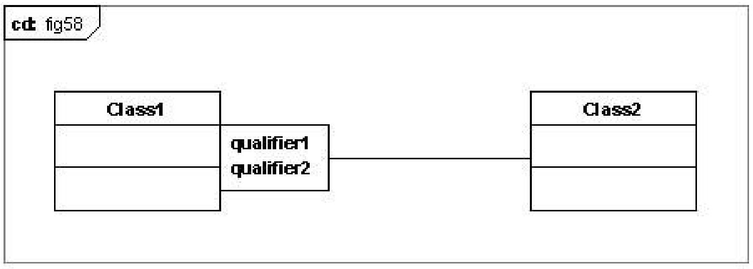
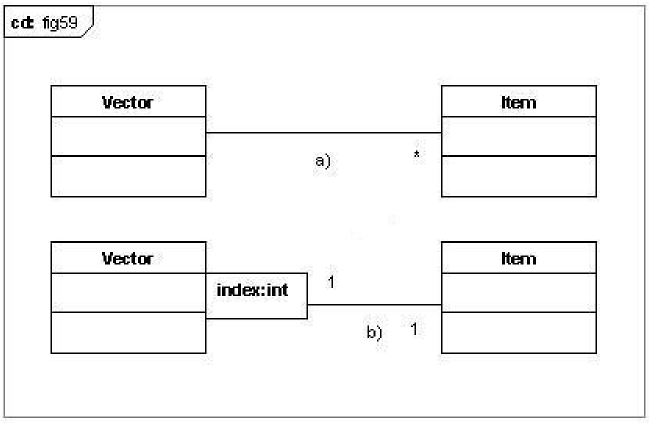


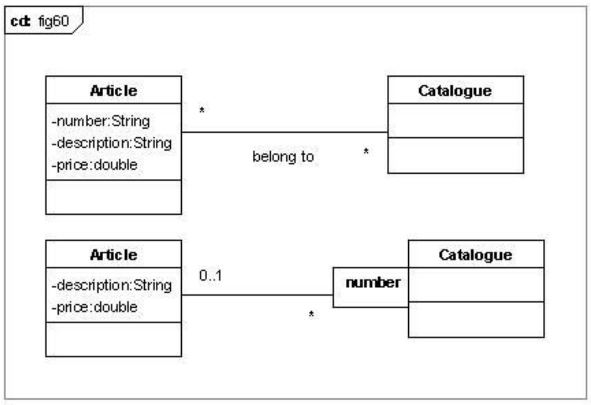
Figure 59 (a,b) describes the modelling of a vector with or without qualifier. In figure 59 b an object of the class `Item` is referred to by means of an `Integer` type index.

Figure 59



Typical multiplicities for qualified associations are the following: `0..1` (a small object can be selected but it is not a necessity); `1` (each qualifier selects an object and only one); `*` (the qualifier divides the quantity of objects in portioned quantities). Figure 60 presents the modelling of qualified associations between two classes: `Article` and `Catalogue`.

Figure 60



The Generalisation/Specialisation Relation

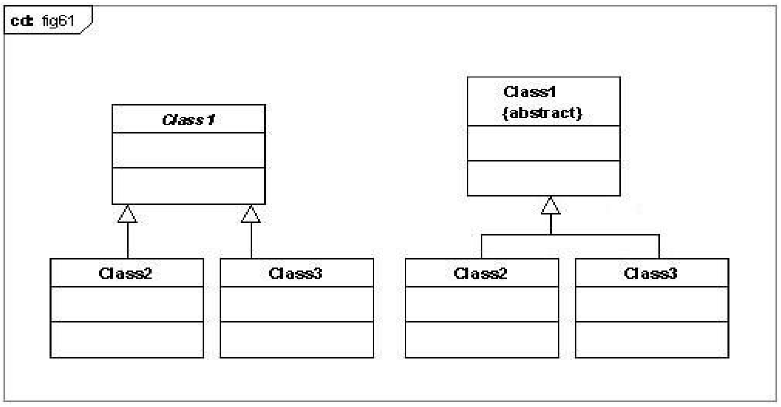
The generalisation describes a relation between a general class (basic class) and a specialised class. The specialised class is completely coherent with the basic class but it involves additional information (attributes, operations, associations).

An object of the specialised class can be used everywhere where an object of the basic class is authorised. In such a case we can speak about a generalising structure.

The general class is also called *supercla*ss and the specialised classed is also called *subclass*.

The generalisation is represented as a finished association with a white triangle with the arrow towards the generalised class (figure 61).

Figure 61



Remarks:

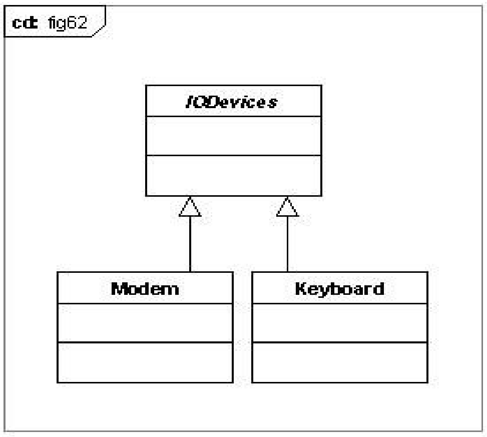
- A relation of generalisation can be interpreted as a relation of the following type: <<*…is an…*>>. As opposed to associations, generalisation relations do not contain any name or any multiplicity.
- Within the modelling process various generalization structures contain abstract classes and interfaces.
- UML authorises multiple heritage, which means that a class can constitute the object of several generalizations, each of them representing a particular aspect of the respective class.

From the implementation and programming language point of view, the generalisation is associated to the inheritance. The subclass inherits all the attributes and methods of the superclass. The subclass can redefine the inherited models and can sometimes affect even the direction of generalization.

Various modern languages –Java, C# forbid multiple heritage; for such situations, these languages appeal to the concept of interface.

Figure 62 presents an example of relation of generalisation between the classes incomplete, *`IODevices`*, *`Modem`* and *`Keyboard`*.

Figure 62



Remark. The generalisation tree is incomplete: there are also other peripheral elements of 1/0: screens, mouse etc.

Figure 63 uses the generalisation/ specialisation relation in order to model the sentence: *A bank account belongs to an individual or to an institution.*

Figure 63

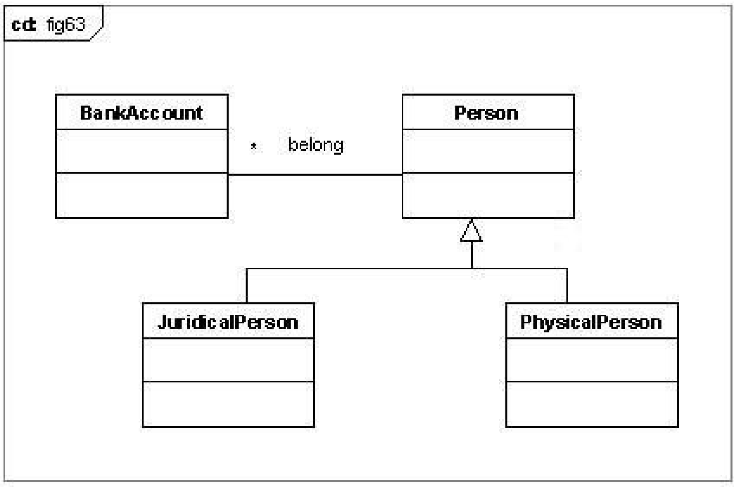
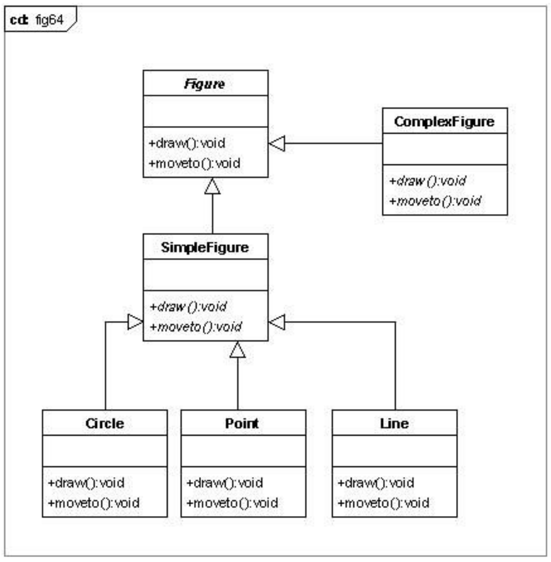


Figure 64 presents the diagram of classes which models simple geometrical figures (circle, point, line) and the composed ones. [3]

Remarks:

- All figures involve the operations `draw()` and `moveto()`.
- Associated models are not known so the class `Figure` is abstract.

Figure 64



A figure can be specialised as a simple figure or as a composed figure which, in its turn, can be composed (aggregation relation) from many figures.

Object Diagram

Representing the Objects

In the majority of cases, the static description of a system is accomplished by means of the class diagram. This allows for the simplification of modelling and for the synthesising of the common characteristics for more objects. Sometimes it is useful, even necessary, to use an object diagram as well.

An object diagram is a topographical representation of the objects of a system at a given moment. They represent instances of the classes (not classes!), that is why they are also called instance diagrams.

Figure 65

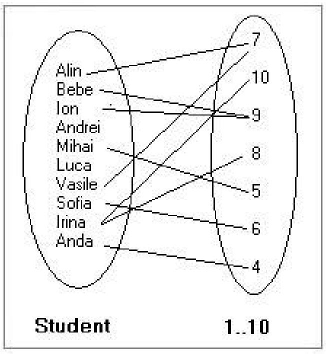


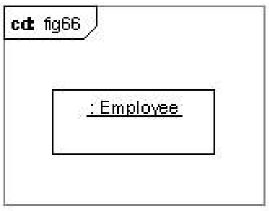
Figure 65 describes a `Grades` relation between students, on the one hand, and grades on the other hand. Certain students do not have grades; others have more. Reversely, certain grades are not distributed to the students and there are students having the same grade.

Starting from the graphic representation of the `Grades` relation in figure 65, we can deduce the object diagram-`Grades`. Graphically, one object is presented as a class with two compartments, that of the name and that of the attributes.

The compartment of class operations is not clear because the interpretation of operations is the same for all objects and all objects of a class possess the same operation.

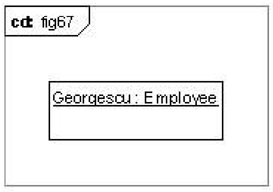
As opposed to the name of a class, the name of an object is underlined. In the case of an anonymous object, the name of the class is also indicated (figure 66).

Figure 66



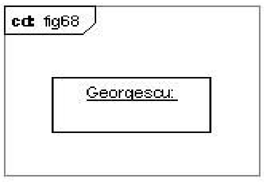
When the object is to be called by means of the name, its identity can be added before the name of the class (figure 67).

Figure 67



When the name of the object is enough for the identification of the object and the name of the class is obvious, we can use from the context the notation for sibling instances as shown in figure 68.

Figure 68



The attributes receive values. UML authorises the following alternatives for the attributes of the object.

<attribute>: <type>= <value>

<attribute>= <value>- this notation is recommended because it often allows for the deducing of the type.

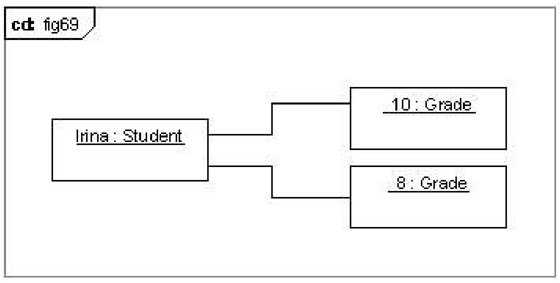
<attribute> -this notation is recommended when the value of the attribute is of no particular interest.

Remarks:

- The class diagram mainly contains the classes and the relations between them.
- When an object diagram is created started from a class diagram, classes are instantiated and become objects, while the relations become connections when instantiated.
- Graphically, a connection is presented as a relation but the name of the relation is underlined.

Figure 69 displays by means of an object diagram the *Grades* relation in figure 65.

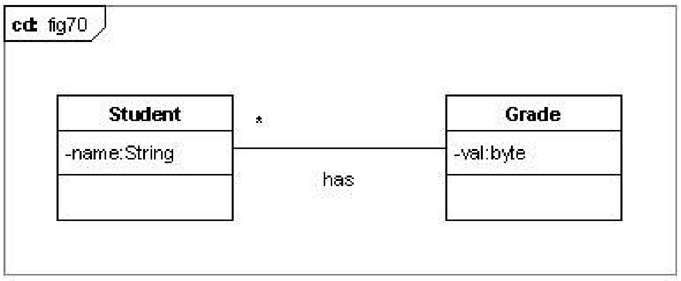
Figure 69



One should also note that the name of the object is underlined while the name of the class is not.

This graphic element is enough in order to make the difference between the object diagram (figure 64) and the class diagram (figure 70).

Figure 70

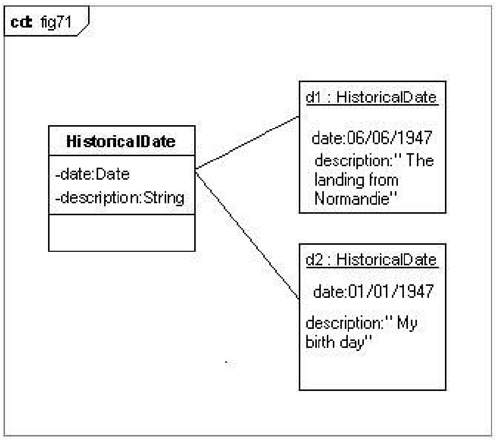


Remarks:

- Since the students' grades are independent, the `Grades` association is not an aggregation or a compounding.
- The names of the students and the values of the grades are represented by means of character string and `Enumeration` type attributes: `1..10`.
- One can pose the problem of a unique representation for a given grade- two grades may have the same value.
- The relation grades can be more simply represented by means of a full group attribute called grades: Set [full] within the `Student` class. The grades are thus values and not objects.

Figure 71 represents a class `HistoricalData`, two instances, four attributes.

Figure 71



Restrictions

OCL Language

UML offers the possibility to express restrictions by means of the modelling constructions we have already studied: cardinalities, type of attribute etc.

Still, these types of restrictions have proven to be insufficient. For other restrictions, UML proposes to express them in a natural language.

There is also the OCL language (Object Constraint Language), an object restriction language expressing them as logical conditions. OCL language is part of the UML notation.

OCL mainly allows for the expressing of four types of restrictions: the invariants, the preconditions, the postconditions and the guarding conditions.

Within the class diagrams, the most often used restrictions allow for the specification of the initial values of an object or of an extremity of an association, of the heritage rules: (*`{complete}`*, *`{incomplete}`*, *`{overlaps}`*, *`{distinct}`* etc), of a method to reduce the portability of an association (*`{subset}`*, *`{score}`* etc), expressing the way the objects evolve (*`{frozen}`*, *`{addOnly}`* etc), the organization (*`{ordered}`*, *`{frozen}`* etc).

For more information consult the OCL specifications for UML 2 available on the OMG site (http://www.omg.or/).

Class Models

The same way as an interface allows for the defining of the specifications for the objects which are to interact with a class, UML also provides abstractings for the class type a class will interact with. For example, one can define a class *`List`* which can contain any type of object (in Java thus can be translated as *`Object`*). Still, if one wants for a class, in our case *`List`*, to be capable of structuring any type of object, we definitely impose that all objects stored in a given list to be of the same type. UML allows for the specification of this abstracting type by means of class models.

One can mention that a class should be based on a class model (the class is templated) by describing a discontinuous rectangle in the upper right corner of this class which contains a list of formal parameters. Each parameter contains a name and a type as well as an optional implicit value.

Figure 72

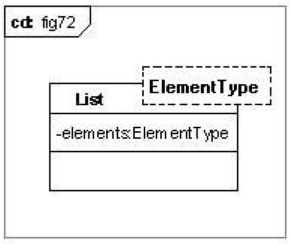
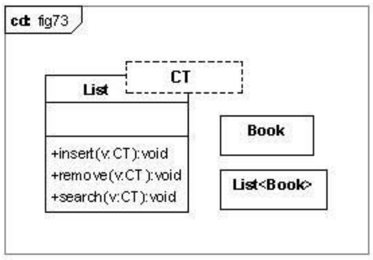


Figure 72 presents an example of a class `List` which can contain any type of object. This example uses `ElementType` as a generic name of the class model. Practically, this generic type is called *T* (figure 72).

Various programming languages (C++) use templated classes. The aim is that of regrouping the associated behaviours to the structure of the class, regardless of the objects it contains.

Figure 73 models the class `List` as a standard class, adding to this definition one or more parameters $CT_1..CT_n$ representing the target objects classes. Once defining the class `Book`, the operations of inserting, of suppressing and of searching for a book are automatically applied.

Figure 73



Opinions regarding the Construction of a Class Diagram

Approaches

The construction of a class diagram as well as of other UML diagrams also depends on the aim and the finality of the modelling.

If one looks for the finality of the modelling, there are at least three points of view guiding the modelling [1], [3]:

- The point of view regarding specification- within the informatic application area the accent falls on the interfaces of the classes;
- The conceptual point of view- the accent falls on the concepts of the domain and on the connection between them;
- The point of view regarding the implementation- the accent falls on detailing the concept and on implementing each class.

Remark. Taking into account the set objectives, you will get models which are sometimes different. From this point of view, one must state the context of the modelling before actually starting to build a diagram

The process of modelling by means of the class diagrams proposes the following stages [3]:

- Identifying the classes of the domain;
- Identifying the associations between classes;
- Identifying the attributes of the classes;
- Organising and optimising the diagrams by using the heritage principle;
- Testing the access paths to the classes;
- The stating and the affirmation of the model.

General Rules

The stages we have presented have a guiding character. Yet, within the process of realising a class diagram one will have to respect the following simple and efficient rules [3]:

- get rid of redundant classes;
- delay the adding of new classes up to the moment of choosing the conception and the realization;
- do not add charts or lists of objects to the classes;
- when a class disposes of more responsibilities, find a simplifying solution (see the abstracting principle);
- for the modelling of objects, differentiate the physical objects from their description;
- an attribute (structural property) can be represented by means of an association, a description with a text within a class or a combination between the two;
- avoid attribute access operations (get and set) which are in the majority of cases useless.

Implementing the UML Concepts within an Object-Oriented Language (Java)

From UML to Java

UML is a visual modelling language and Java, C# are textual programming languages.

UML is richer than the programming languages in the sense that it offers stronger and more abstract ways of expressing.

Java is a rapid, elegant and strong programming language starting with the launching of the 1.02 version, Java attracting programmers by means of its friendly syntax, the object oriented characteristics, the memory administration and portability.

Java 5.0 (versions 1.5 and newer ones) has brought major changes within the proper language, making it simpler for programmers and adding to it already popular characteristics.

Java cycle is different from the model. The information contained by an UML model has as objective generating the application code.

To accomplish an UML language it is not easier than writing a Java code.

The information contained within the models has to be synchronised with the Java code.

There is in this respect a privileged way of translating UML concepts into Java statements.

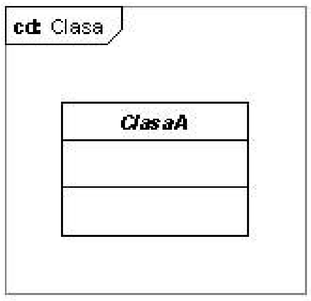
This chapter offers you a synthesis of the major correspondences between the UML modelling concepts and the rules for generating the Java code.

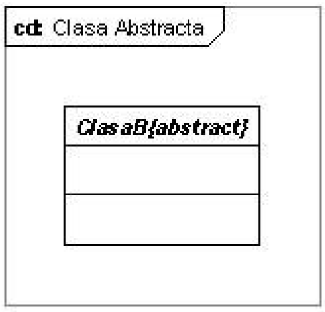
UML Correspondence Rules towards Java

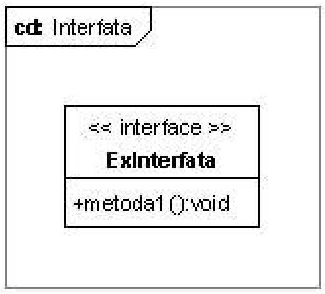
Table 4 presents a synthesis of the major correspondences between the UML modelling concepts and the way they are implemented within Java.

Table 4.

Concept	*UML*	*Java*

Class		`public class A {`
		`…`
		`}`

Class(abstract)		`public abstract class B {`
		`...`
		`}`

Interface	 cd: Interfata
	<< interface >>
	ExInterfata
	+metoda1 ():void
		`interface ExInterfata`
		`{`
		`void metoda1( );`
		`}`

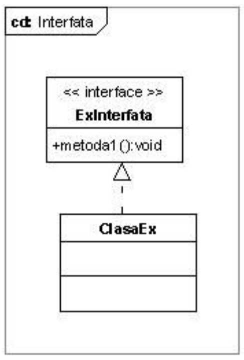
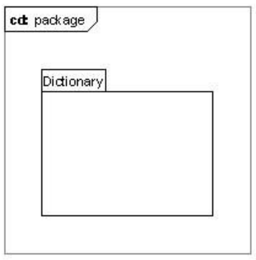
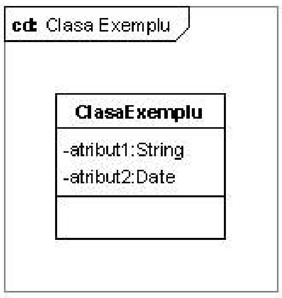
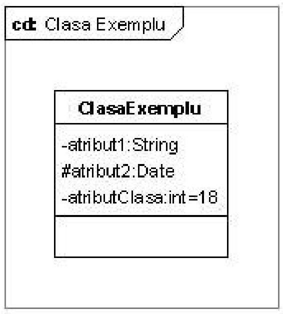
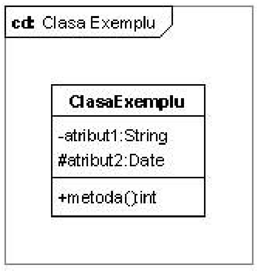
Interface	 cd: Interfata
	<< interface >>
	ExInterfata
	+metoda1 ():void
	ClasaEx
		`interface ExInterfata`
		`{`
		`void metoda1( );`
		`}`

Table 4.(cont.)

package		`package dictionary;`

Attribute	 cd Clasa Exemplu
	ClasaExemplu
	-atribut1:String
	-atribut2:Date
		`import java.util. Date;`
		`public class ClasaExemplu {`
		`private String atribut1; private Date atribut2;`
		`…`
		`}`

Attribute
 	
(of class)
	 cd Clasa Exemplu
	ClasaExemplu
	-atribut1:String
	#atribut2:Date
	-atributClasa:int=18
		`abstract public class clasaExemplu {`
		`private String atribut 1; protected Date atribut 2; private static int atributClasa=18;`
		`}`

Operation	 cd Clasa Exemplu
	ClasaExemplu
	-atribut1:String
	#atribut2:Date
	+metoda():int
		`public class ClasaExemplu {`
		`private String atribut1; protected Date atribut2; public int metoda( ) {`
		`…`
		`}`
		`}`

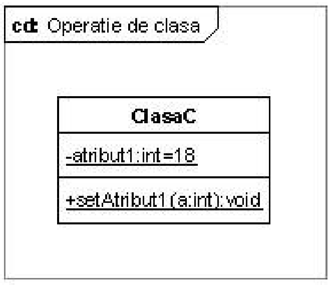
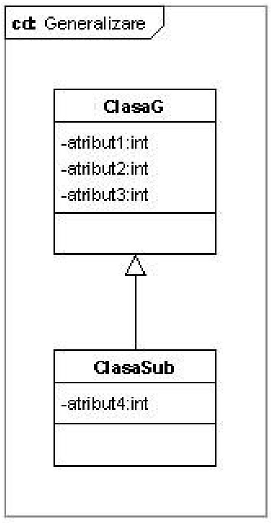
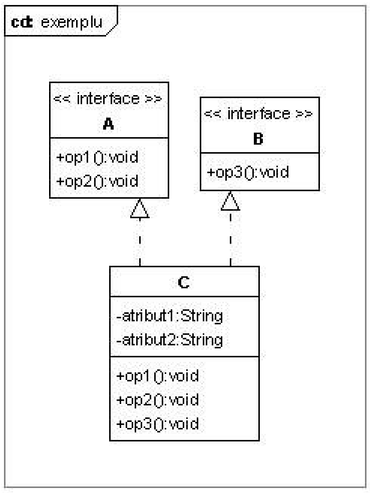
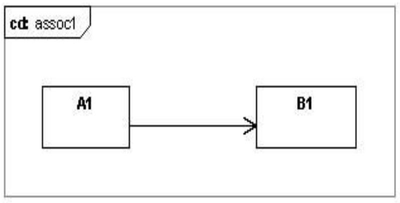
Operation
 	
(of class)
	 cd Operatie de clasa
	ClasaC
	-atribut1:int=18
	+setAtribut1 (a:int):void
		`public class clasaC {`
		`private static int atribut1= 18;`
		`public static void setatribut1 (int a) {`
		`...`
		`}`

Table 4.(cont.)

Generalisation	 cd: Generalizare
	ClasaG
	-atribut1:int
	-atribut2:int
	-atribut3:int
	ClasaSub
	-atribut4:int
		`public class ClasaSub extends ClasaG {`
		`private int atribut4;`
		`...`
		`}`

Realisation	
	cd: exemplu
	<< interface >>
	A
	+op1():void
	+op2():void
	<< interface >>
	B
	+op3():void
	C
	-atribut1:String
	-atribut2:String
	+op1():void
	+op2():void
	+op3():void
		`public class C implements B, A {`
		`private String atribut 1;`
		`private String atribut 2;`
		`public void op3 ( ) {`
		`…`
		`}`
		`public void op1 ( ) {`
		`…`
		`}`
		`public void op2 ( ) {`
		`…`
		`}`

Navigable association		`public class A1{`
		`private B1 b1;`
		`}`

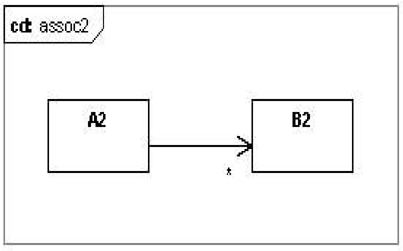
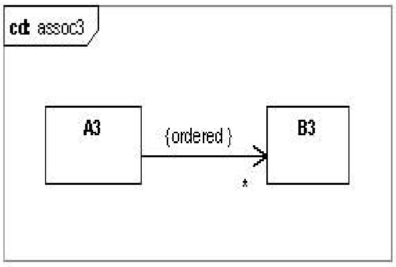
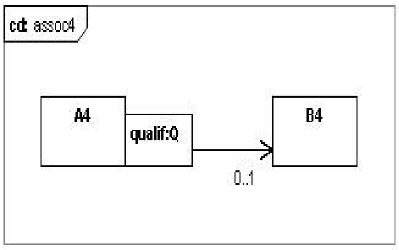
Navigable association		`public class A2{`
		`private B2[] b2;`
		`}`

Table 4.(cont.)

Navigable association		`public class A3{`
		`private List<B3> b3= new ArrayList<B3>();`
		`}`

Navigable association		`public class A4{`
		`private Map<Q,B4> b4= new MapSet<Q,B4>();`
		`}`

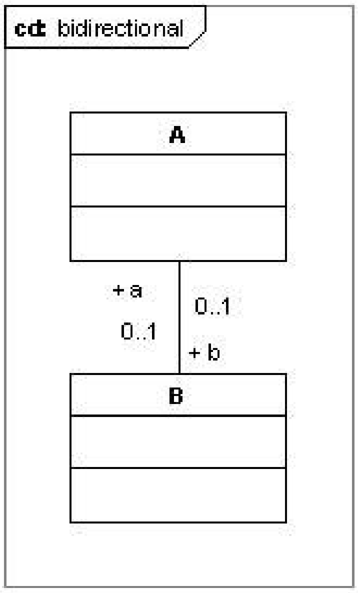
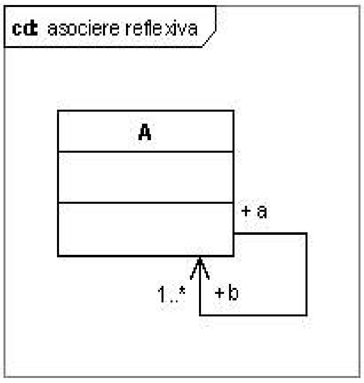
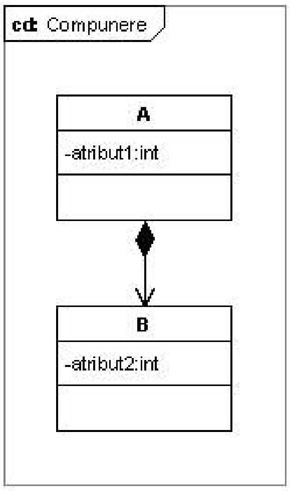
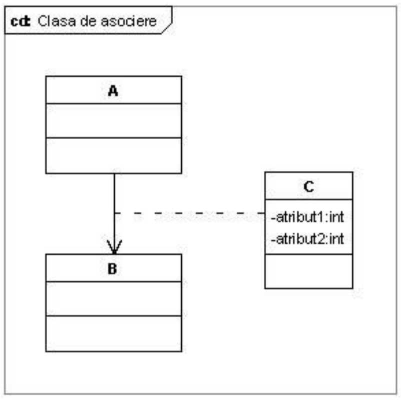
Bidirectional association		`public class A {`
		`private B b;`
		`…`
		`}`
		
		`public class B {`
		`private A a;`
		`…`
		`}`

Table 4.(cont.)

Reflexive association		`public class A {`
		`private A b [ ];`
		`…`
		`}`

Composition		`public class A {`
		`private int atribut1;`
		` private B b;`
		`private static class B {`
		`private int atribut2;`
		`}`
		`…`
		`}`

Association class	 cd: Clasa de asociere
	A
	B
	C
	-atribut1:int
	-atribut2:int
		`public class C {`
		`private int atribut1;`
		`private int atribut2;`
		`private A a;`
		`private B b;`
		`…`
		`}`

Remarks:

- The UML tools transform any UML class into Java class stored in separate file.
- An abstract class is always written using italic style of font.
- The UML interface is translated in Java language using keyword `interface`.
- In Java language the attributes become class member variable. The attribute type could be simple (int, double, float, long, char etc.) or could be a class (String, Date etc.).
- The UML class atribute become static member variable in Java language.
- The UML operations are class methods in Java language and class operations become private static methods of class.
- The generalization in Java language is the inheritance feature of object oriented programming and will be introduced by keyword `extends`.
- The realisation of one UML interface is done by Java language keywords `implements`.
- A navigable association of multiplicity 1 is translated in Java language using a member variable of referred class type. A multiplicity of kind <<*>> is translated in Java language using a collection of referred class.
- The bidirectional association is translated into Java language by one member variable of referred class type in each class. The variables identifier is the name of the role placed at the end of the association.
- The UML reflexive association is translated into Java by a member variable of same class type.

Figure 75 presents the generated Java code starting from the classes, book, magazine, CD showed in figure 74.

Figure 74

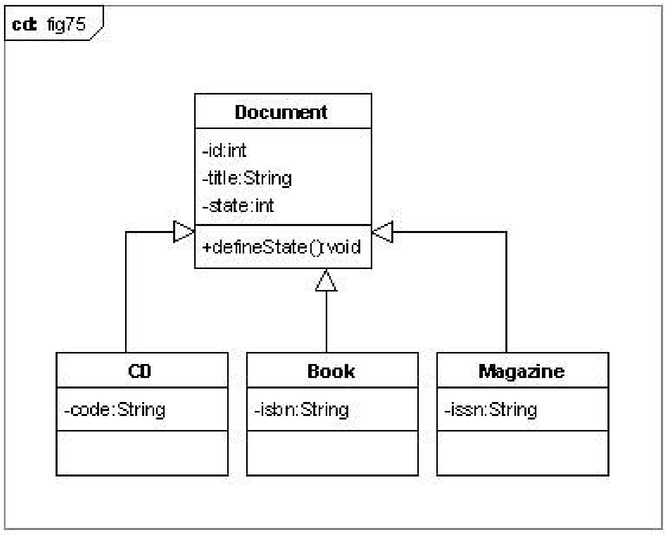


Figure 75

```
public class Document {
    private int id;
    private String title;
    private int state;
    public void defineState() {
        // your code here
    }
 }
public class CD extends Document {
    private String code;
 }
public class Book extends Document {
    private String isbn;
 }
public class Magazine extends Document {
    private String issn;
 }
```

Figure 77 displays the generated Java code starting from the classes `University`, `LectureRoom`, `Projector` presented in figure 76.

Figure 76

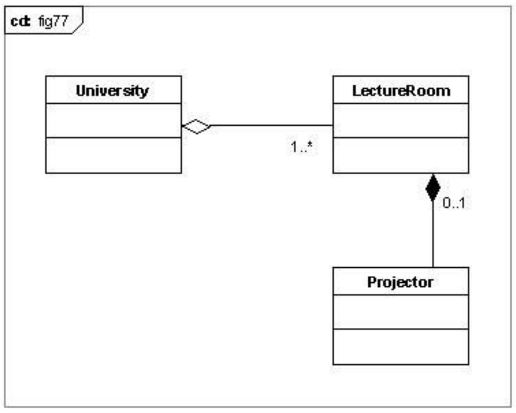


Figure 77

```
public class University {
    public java.util.Collection lectureRoom = new java.util.TreeSet();
 }
public class LectureRoom {
    public University university;
    public Projector projector;
 }

public class Projector {
    public LectureRoom lectureRoom;
 }
```

Figure 78 displays the generated Java code starting from the classes `Vector` and `Item` (figure 59).

Figure 78

```
public class Vector {
  public Item[] item;
 }
public class Item {
 }
```

Figure 79 displays the generated Java code starting from the classes *`Person`*, *`BankAccount`* presented in figure 63.

Figure 79

```
public class BankAccount {
 }
public class Person {
  private BankAccount[] bankaccount;
 }
public class JuridicalPerson extends Person {
 }
public class PhysicalPerson extends Person {
 }
```

Figure 81 displays the generated Java code starting from the classes *`RealPile`*, *`RealCell`* (figure 80).

Figure 80

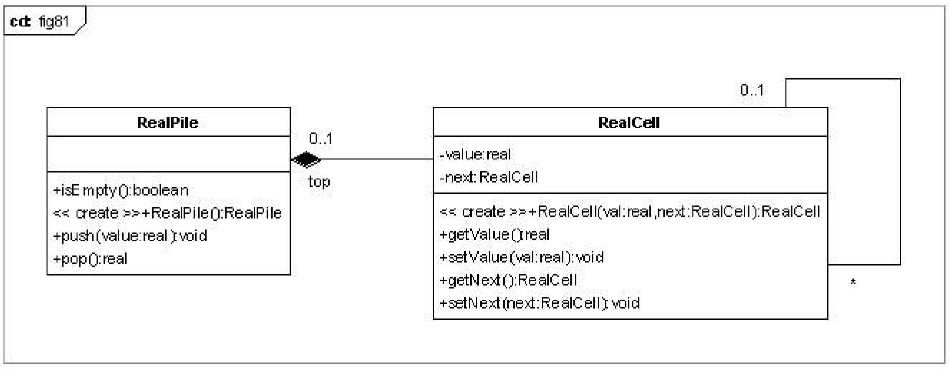


Figure 81

```
public class RealPile {
    public RealCell top;
    public boolean isEmpty() {
        // your code here
        return false;
    }
    public  RealPile() {
        // your code here
    }
    public void push(double value) {
        // your code here
    }
    public double pop()
        // your code here
        return null;
    }
 }
```

```
public class RealCell {
    private double value;
    private RealCell next;
    public  RealCell(double val, RealCell next) {
        // your code here
    }
    public double getValue()
        // your code here
        return null;
    }
    public void setValue(double val) {
        // your code here
    }
    public RealCell getNext() {
        // your code here
        return null;
    }
    public void setNext(RealCell next) {
        // your code here
    }
}
```

References

1. B a l z e r t , H. - *UML 2 compact*, Eyrolles, Paris, 2005
2. B l a n c , X., M o u n i e r , I., B e s s e , C. - *UML 2 pour les developpeurs*, Eyrolles, Paris, 2006
3. C h a r r o u x , B., O s m a n i , S., T h i e r r y - M i e g , Y. - *UML 2*, Pearson Education France, Collection Syntex, 2005
4. D e b r a u n e r , L., K a r a m , N. - *UML 2, Entrainez-vous a la modelisation*, ENI, Paris, 2006
5. P i l o n e , D, P i t m a n , N. - *UML 2. En concentre. Manuel de reference*, O'Reilly, Paris, 2006

R o q u e s , P. - *UML 2 par la pratique*, 5-e édition, Eyrolles, 2006

Ciclul: UML PENTRU MANAGERI (V) Diagramele de clasă (continuare)

Rezumat

Articolul continuă descrierea diagramelor de clasă, care reprezintă nucleul oricărei aplicaţii software orientate obiect. Articolul descrie relaţiile între clase, relaţii de asociere, agregare, compunere, dependenţă, generalizare/specializare, precum şi conceptele de multiplicitate, navigabilitate şi asociere calificată.