BULETINUL Universităţii Petrol – Gaze din Ploieşti	Vol. LIX No. 1/2007	53 - 74	Seria Ştiinţe Economice

Cycle: UML FOR MANAGERS (IV) The Class Diagram

Liviu Dumitraşcu, Gabriel Irinel Marcu

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

Abstract

The class diagram represents the structure of an object-oriented application by means of the classes and of the relations established between them. The present paper thoroughly describes the concept of "class diagram" by means of UML notations and of numerous examples.

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*

Example of Class Diagram

A Simplified System of Administrating an International Scientific Conference

Class diagrams are probably the most important diagrams of UML. They can be used in the development stage of an application.

A class diagram represents the entities with their attributes and operations, as well as the generalizations between classes as they derive from the example presented in figure 1.

The purpose of class diagram is to describe the entities encountered into an software application.

The class diagram in figure 1 models a simplified system of administering an international scientific conference, adapted after Laurent DEBRAUNER and Naouel KARAM [4].

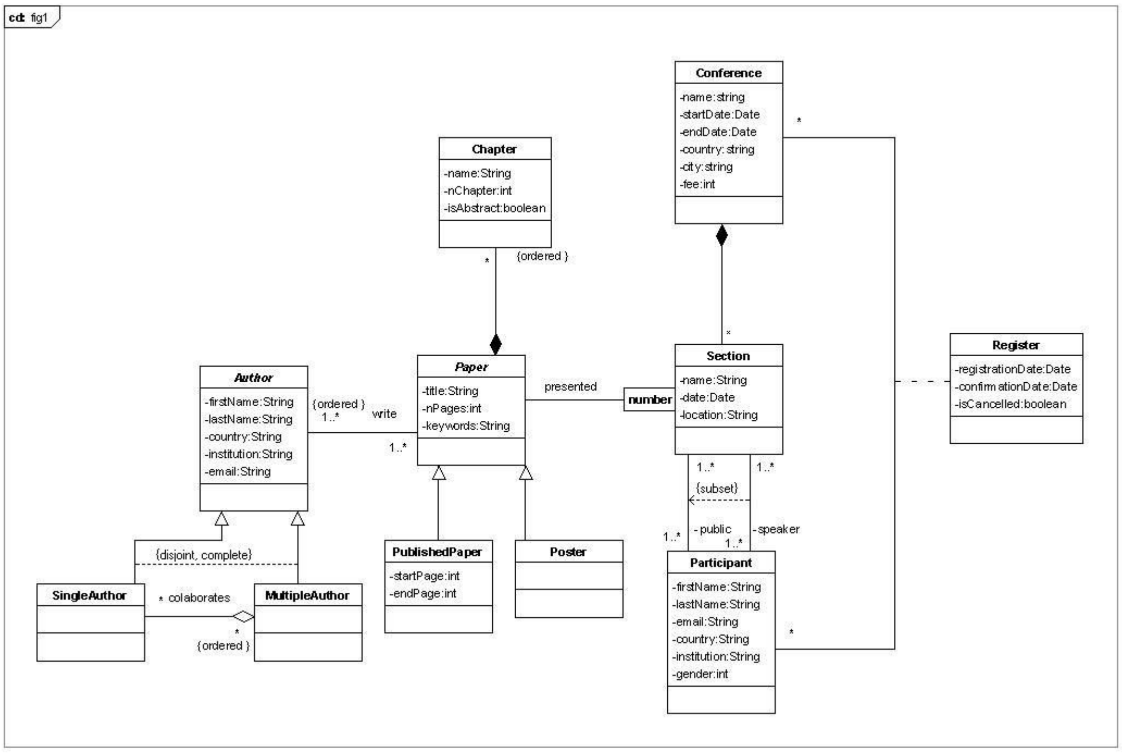
The domain concept identification and the creation of the class diagram are based on the following description of the domain as sentences.

1. The conference is made of more sections.

This (first) sentence emphasizes two important concepts for the domain: `Conference` and `Section` which are presented by means of classes (figure 2).

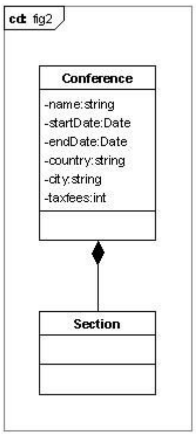
The relation which connects the two classes is a composite aggregate having in mind the fact that a section cannot function without being part of a conference (figure 2).

Figure 1



The class `Conference` is characterized by: name, beginning and ending date, the organizing country and city, things which represent attributes of the model (figure 2).

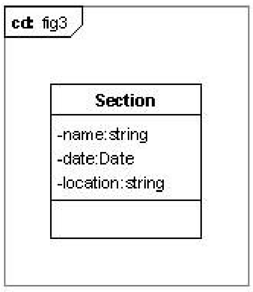
Figure 2



2. *Each section has a name, a date and a location.*

The name, the date and the location are notions which represent significant values for each section. They are represented as attributes of the class `Section` (figure 3).

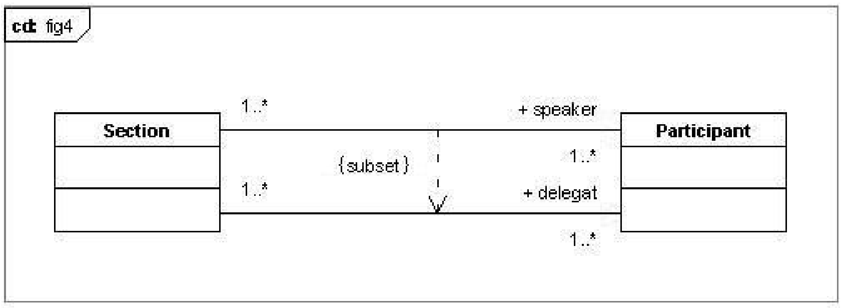
Figure 3



3. The participants registered for the conference within a certain section can play the role of speakers (presenting the paper) or of delegate or both.

The solution we have applied (figure 4) has been that of using the notion of role in an association (*`delegate`*, *`speaker`*) as well as the OCL restriction (Object Constant Language), *`{subset}`* between the two associations.

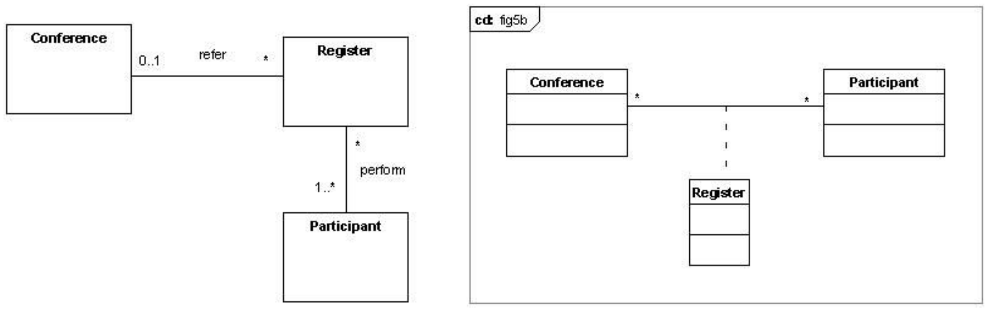
Figure 4



The relation which connects the two concepts of the domain –*`Section`*, *`Participant`*- represented by classes (see the attributes of the two classes) is of an association type where they have been placed at the two poles: the role and multiplicity (the number of objects susceptible to participating at an association).

4. All participants must register for the conference. A registering may be cancelled or confirmed.

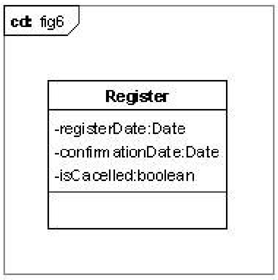
Figure 5 *a.* *b.*



This sentence describes a connection between the participant and the conference by means of an association. Two variants are possible. In the first variant (figure 5a), the registration is represented by means of a class (`Register`) connected to the classes `Participant` and `Conference`. The second variant (figure 5b) the registration is represented by means of an association class `Register` between the classes `Participant` and `Conference` (figure 5).

The confirmation or canceling of a registration describe the operations which can be applied to the class `Register` (figure 6).

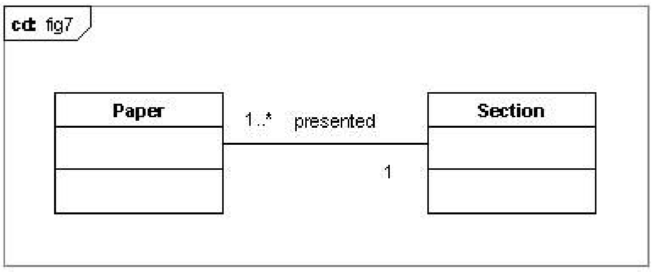
Figure 6



5. A scientific paper is presented within a section of the conference

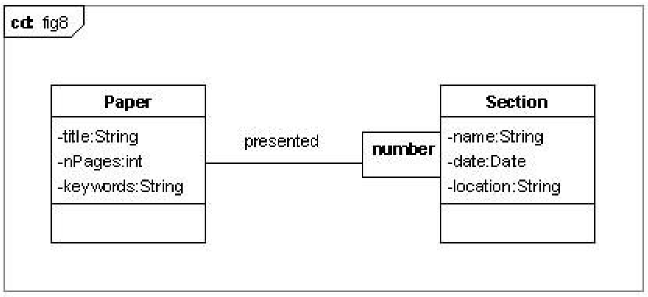
This sentence is modeled by means of an association between the classes `Paper` and `Section` (figure 6/7)

Figure 7



Due to the fact that the papers are numbered within one section, we also added the term Number to the class Section and the cardinality of the class `Paper` has become 1 (figure 7). The class `Paper` is represented by means of the attributes: title number of pages, key-words(figure 8).

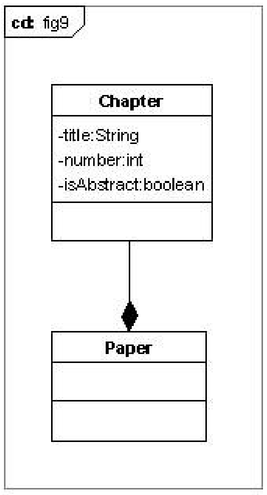
Figure 8



6. An article is made of chapters which are numbered and it contains a certain subject, an abstract and key-words.

For the modeling of this sentence we have used a relation of compounds between the classes `Chapter` and `Paper` since a chapter belongs only to a single `Paper` and its existence depends on the existence of the `Paper` (figure 9).

Figure 9

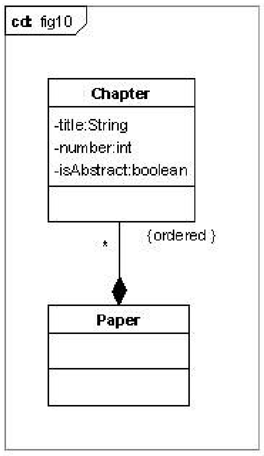


The title of the paper and the number of pages have been modeled as attributes of the class `Paper`. Since the chapters are numbered, we added to the class `Paper` the UML restriction `{ordered}` (figure 10).

7. An individual or collective author can take part in the international scientific conference with one or more articles (papers)

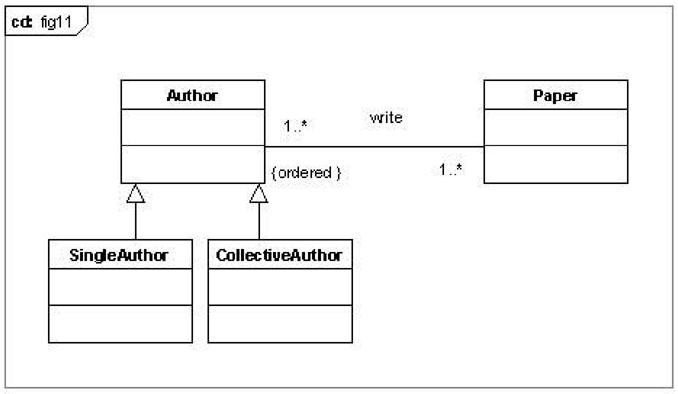
This last sentence describes the connection between the classes `Author` and `Paper`. An author can write more articles and an article can be written by more authors (figure 9).

Figure 10



The order of the authors has been modeled by means of the UML restriction `{ordered}` (figure 9). The classes `SingleAuthor` and `CollectiveAuthor` are specialisations of the class `Author` and between the two classes there is a relation of aggregation (figure 11).

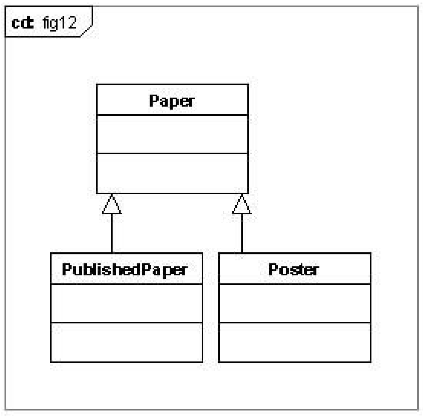
Figure 11



8. A paper can be a poster or a published article.

In order to model this sentence we have used a relation of generalisation in which the classes `PublishedPaper` and `Poster` are specialisations of the class `Paper` which, in this case, becomes abstract (figure 12).

Figure 12



The Class Diagram

Principles and Definitions

The class diagram is beyond doubt the most used UML diagram and, at the same time, the one which contains the biggest number of UML notations. The class diagram represents the structure of an object-oriented application by means of the classes and of the relations established between them.

In a class diagram, classes are represented by means of a rectangle with three sections: the first one containing the name of the class, the second one containing the attributes and the third one the operations. A class can be schematized as a reference represented by means of a simple black box.

The relations that can be established between the classes presented within a diagram are: association, dependency, inheritance or generalization.

Class diagrams are used in order to define the static relations of a system. They equally allow for the defining of the physical structure of a system.

UML class diagrams can be used at all abstracting levels. Nobody prevents us from representing an software application by means of a single class (high abstracting level) or to represent all the components of this application as classes (low abstracting level).

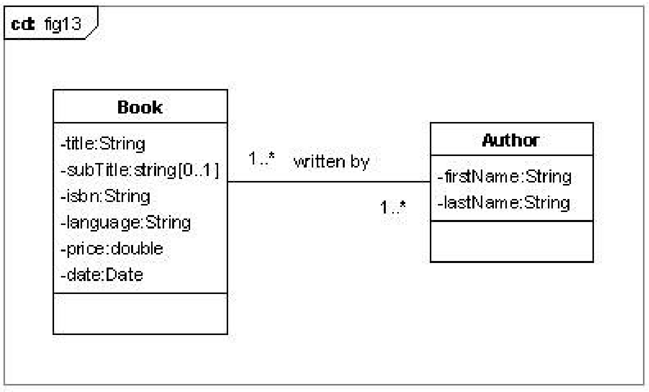
The Role of the Class Diagram in the Analysis Stage

In the analysis stage the class diagram is used in order to describe the entities manipulated by the user as well as the restrictions or administrating rules.

In the analysis stage the class diagram is used in order to precisely formalise the relations between classes. The association class allows for the representation of a complex association between classes which can carry particular operations and attributes. This concept is important because it brings semantic precision to the analysis models.

In the analysis model of an application represented by means of a diagram it is good not to hide the associations under the shape of attributes. In order to solve the dilemma: attribute or concept?, it is recommendable to apply the following criterion: if an entity is not demanded more than its value, we are talking about one attribute, but if more than one question is asked, we are talking about a concept which is endowed at its turn with more attributes. Figure 13 presents an evocative example. For a book, the publishing date and the language are attributes, while the author is a concept because are demanded: the name, the surname and the title of the books he has written.

Figure 13



The Role of the Class Diagram in the Conception Stage

The conception has to complete the analysis model in order to specify which models have to be used in order to generate the code (Java).

Class diagrams directly represent the structure of the code which has to be accomplished. It is the reason why the class diagram occupies ranks quite high in the conception stage.

The conception of the attributes and of the operations is a first conception task which resides in the precise definition of the type visibility, of the cardinalities, of the property parameters and of the restrictions.

The visibility of the attributes corresponds to the property public (+), protected (#), private (-).

The types of attributes and the operations correspond to the basic types of the target language (Java) or of the defined types for the application by means of the stereotypes (data type, enumeration, etc).

The standard UML data types are: *`Integer`* (for whole type data); *`String`* (for lines of characters); *`Boolean`* (for logical type data); *`Real`* (for real type data).

UML 2 doesn't stipulate anything for the declaration of the exceptions to an operation: an improvement of this inconvenient is proposed by using a restriction of a close stereotype.[3]

Interfaces are frequently used in the conception stage.

Interfaces are pseudo-classes whose role is that of defining a collection of operations realized by means of a particular class.

From Object to Class

An Example of an UML Representation of a (Concrete) Class

An object is an identifiable entity of the real world. It can have a physical existence (a book, a car etc.) or a logical existence (a law text, code blocks when constructing a Java program etc.).

The used objects in UML are abstracting images of the objects in the real world.

Very many natural or informatic objects are similar in what concerns the level of their description or their behaviour: in order to restrict the modelling complexity you can regroup the objects sharing the same characteristics in classes. Thus, a class describes the states (attributes) and the behaviour at a high abstracting level of the similar objects. This way you can define the class as a regrouping of the data and of the processings applicable to the objects they represent as shown in figure 2.

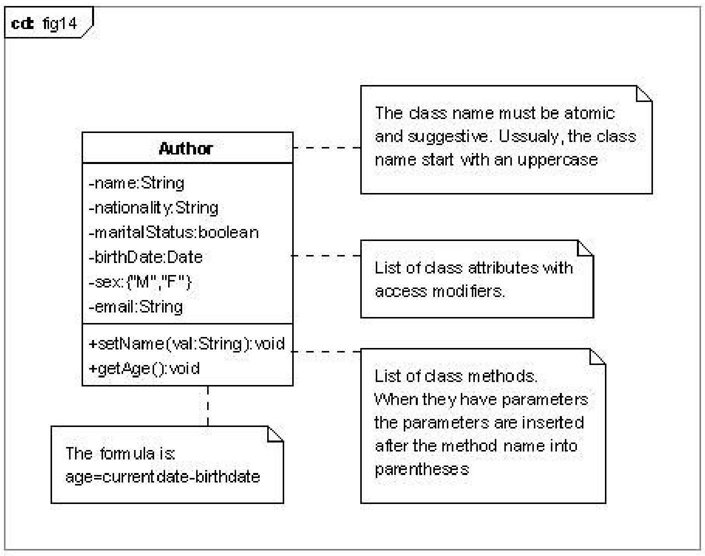
In the object approach, an object is an entity with well defined frontiers endowed with identity and encapsulating a state and a behaviour. An object is an instance of a class.

Figure 14 models the following situation: Vasile is a single 30 year old Romanian man having the email address vasile@exel.ro and Bianca is a Thai 40 year old married woman with the email address: Bianca@thalia.com.

In the example presented in figure 14 there have been regrouped the similarities between the two objects (Vasile and Bianca) in a class named *`Author`*. The information exposed by this example are: name, nationality, marital status, sex, email. Thus, the class *`Author`* contains the following attributes: *`name`*, *`nationality`*, *`marital status`*, *`sex`*, *`email`*.

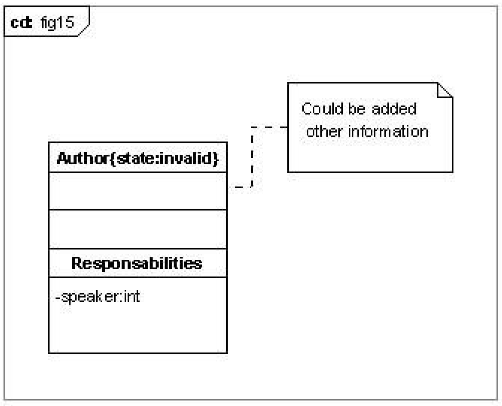
Age is a piece of information whose value can be changed. An author's age can be calculated by making the difference between the current and the birth date. In this situation we can say that we have to do with derived attributes. This last analysis allows us to complete the description of the class with a new attribute, *`birthDate`* and an operation *`getAge`* calculus which allows for a person's age calculation.

Figure 14


cd: fig14
Author
-name:String
-nationality:String
-maritalStatus:boolean
-birthDate:Date
-sex:{"M","F"}
-email:String
+setName(val:String):void
+getAge():void
The class name must be atomic and suggestive. Ussualy, the class name start with an uppercase
List of class attributes with access modifiers.
List of class methods. When they have parameters the parameters are inserted after the method name into parentheses
The formula is: age=currentdate-birthdate

In UML, the class `Author` is graphically represented by means of a rectangle divided into more sections. The first one indicates the name of the class, the second one the attributes of the class while the third one the methods. Taking into account the stage of the modelling and the elements that are to be emphasised, we can also add other sections, obviously these going beyond the standard UML frame (figure 15).

Figure 15

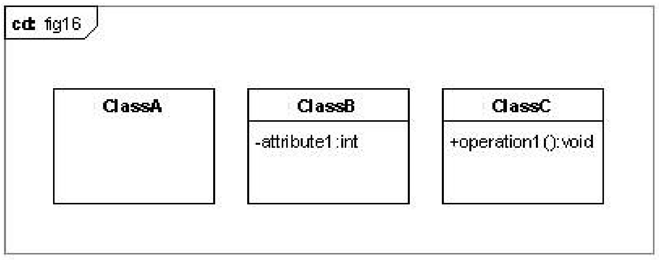


It is possible to mask any of the sections of a class in order to make possible the graphical representation (of the class) in a diagram. When one reads a class diagram one cannot formulate any hypothesis regarding the missing section; this does not mean that it is a void (figure 16).

An object belonging to a class must have unique values for all the attributes defined within the class and must be able to perform (no exception accepted!) all the models of the class.

*Remark.*In the object programming languages the notion of instance and that of object are often mistaken, the former being more general than the latter. An object is the instance of a class but an instance can be the instance of a wider range of elements of the UML language. For example, a connection is the instance of an association.

Figure 16



Abstract Class and Method

In the every day life, we are more than once placed in the position of ignoring a multitude of details which allow us to have a global view over reality. This notion of abstracting can also be found in the object-oriented approach, more precisely in the abstract classes and methods.

A concrete class can be instantiated. It constitutes a complete object model (all the attributes and methods are fully described).

An abstract class, as opposed to a concrete class, cannot be directly instantiated because it does not provide a full description. The role of an abstract class is that of possessing concrete sub-classes. This abstracting is important for the factorization of the attributes and common methods realized by the sub-classes.

More often than not, the sub-class factorization of the common methods can be translated as a sole factorization of the description. A method introduced in a class with a single description and without any code is named abstract method.

In UML, a class or an abstract method are represented by means of the stereotype `abstract`. Graphically, this is represented both explicitly (*`{abstract}`*) or implicitly with a cursive formatting (in italics) of the name of the class or method (see the example in figure 1-classes *`Author`* and *`Paper`* are abstract).

Remarks:

- Abstract methods are redefined within the classes `Author` and `Paper` in order to become concrete (the polymorphism principle).
- The polymorphism represents the fact that a class (abstract) describes a group of different objects because they are instantiated by different sub-classes. The calling of one method with the same name can be translated by means of different behaviours (except for the case when the method is common and it inherits the hyper-class in sub-classes).

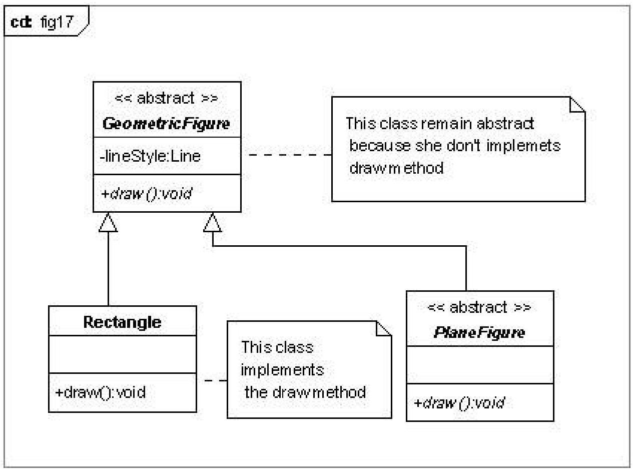
An abstract class is conceived for: the classification of conception; the imposition of the same attributes and definitions to the same classes and for different methods; writing certain more generic algorithms due to polymorphism.

A method is called abstract when its name is known but not the way in which it is to be accomplished.

A class is called abstract when it defines at least an abstract method or when a fostering class contains an abstract method which is not yet realized.

Figure 17 illustrates an example of modelling an abstract class (adapted from [3]) regarding the graphic representation of an UML diagram by means of a geometrical figure.

Figure 17

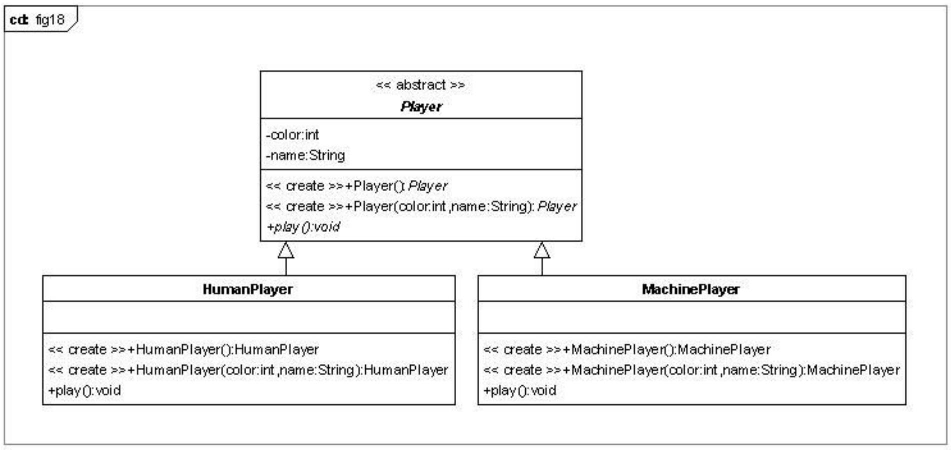


It is obvious that if we do not know the exact shape of the figure to be drawn we cannot actually realise it. Thus we say that the *draw* method is abstract within the *GeometricFigure* class whereas the *GeometricFigure* class is abstract in itself. If we define a rectangle as a particularization (specialization) of the geometrical figure then the rectangle can be drawn and the *draw* method provides a body for the *draw* operation which will be called concrete. An abstract class contains at least an abstract method.

Figure 17 illustrates an example of an abstract class which contains an abstract method and the consequences of such a property on the derived classes. Particularly, a class pertaining to an abstract class must implement all the abstract methods on the contrary remaining abstract as well.

Figure 18 illustrates an example of abstract class modelling (*Player*) regarding a game involving human players competing against an artificial intelligence.

Figure 18


cd: fig18
<< abstract >>
Player
-color:int
-name:String
<< create >>+Player():Player
<< create >>+Player(color:int,name:String):Player
+play ():void
HumanPlayer
<< create >>+HumanPlayer():HumanPlayer
<< create >>+HumanPlayer(color:int,name:String):HumanPlayer
+play():void
MachinePlayer
<< create >>+MachinePlayer():MachinePlayer
<< create >>+MachinePlayer(color:int,name:String):MachinePlayer
+play():void

In order to model such a game we have defined a class called *HumanPlayer* in order to make possible for the player in front of the computer to play and a class *MachinePlayer* in order to make possible for the computer to play. Only these two classes are very close because they will be interchangeable: the game will allow for the following sessions- *HumanPlayer-HumanPlayer*, *MachinePlayer-MachinePlayer* and *HumanPlayer-MachinePlayer*.

There is no inheritance between the *MachinePlayer* class and the *HumanPlayer* class. In order to connect them and to ensure clarity and the simplification of the conception, we must introduce the fostering class *Player*. The game will be accomplished either by the human player or by the computer but never by the *Player*. This class, *Player*, will not be usable in a program other than by means of polymorphism (the abstract method *play()* from the abstract class *Player*).

The Name of the Class

Modelling a class presupposes several phases eventually involving more characters. In order to make a difference between these two modelling stages, you will have to add all the necessary information to the name of the class.

The most frequently used information is: the name of the class, the slots containing it, optionally the class stereotype, the author of the modelling, the date and the status of the class (validated or invalidated).

Implicitly, a class is considered to be concrete. If not, add the key-word abstract in order to specify the fact that it is abstract.

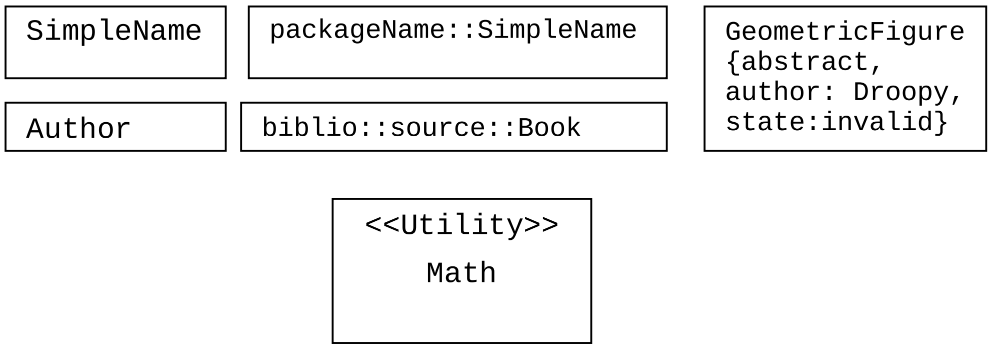
The basic name declaration syntax of a class is presented in figure 19.

Figure 19

```
[<<stereotype>>]
[<packageName1>::...........<packageNameN>::]
  <className>[{[abstract], [author], [state],...}]
```

Figure 20 presents some examples of class names.

Figure 20



Remarks:

- The name of the class has to be significant- to suggest the concept defined by the class.
- The name of the class always has: to start with a capital letter; to be centred in the upper compartment of the class; to be written in bold; to be written in italics if the class is abstract.
- If the name of the class is compounded, each word has to start in capital and the spaces between the words have to be eliminated.
- In UML 2, inverted commas (such as in <<*Utility*>>) are used in order to state the stereotypes and the key-words of the language.

Encapsulation

The principle of encapsulation allows for the defining of the right of access to the properties and methods of a certain object. They are encapsulated and they are called private attributes and methods of the respective object.

UML, as well as the majority of the modern object-oriented languages, introduces three possibilities of encapsulation: private, protected, package.

The private attribute/method- the property is not visible in the exterior of a class.

The protected attribute/method- the property is visible from within the respective class and from all the derived (sub-classes).

Encapsulating the package- the property is visible only from the instances of the classes pertaining to the same package.

Remark. The notion of private property is rarely used because it imposes the existence of a clear difference between the instances of a class and of a sub-class. This difference is connected to the very subtle aspects of the object-oriented program.

The encapsulation of the packages comes from Java language. It is reserved to the diagrams destined to the programmer.

Very many authors [3] recommend using protected encapsulation.

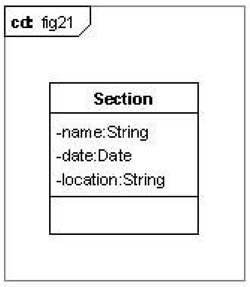
Encapsulation is represented by means of a symbol (access modifier) that precedes the name of the attribute. Table 1 thoroughly presents the four representational symbols of encapsulation.

Table 1.

Symbol	*Keyword*	*Significance*
+	public	New encapsulated element visible from anywhere
#	protected	Encapsulated element visible in the subclasses of the class
-	private	Encapsulated element visible only from within the class
~	package	Encapsulated element visible only in the classes of the same package.

Figure 19 displays the class `Section` which illustrates the characteristics of encapsulation (figure 21).

Figure 21



Attributes

The attributes define the information that a class or an object has to contain. They represent the data encapsulated in the object of this class.

Example: the colour, the number of doors and the registration number are attributes of the class `Car`.

The attributes can be simple primitive data (complete, letters etc.) or relations with other complex objects (dependency, association, navigability, aggregation, compounding, generalization etc.)

Each attribute is of a special type. All the objects of a class contain the same attributes but sharing different values (of those attributes). The attributes are placed within a separate compartment. Each attribute is defined by means of a name, a type of data and visibility (access modifier). The name of the attribute must be unique within a class. The triggering syntax of an attribute is unfolded in figure 22.

Figure 22

```
[access modifier][1]<attributeName>:
<type>['['multiplicity']']
[=initialValue]
<access modifier]:={+|-|#|~|}]
<multiplicity>:=[min..max]
```

The significance of the syntactic elements of this notation is presented in table 2.

Table2.

Syntactical element	*Significance*
Access modifier	It indicates the visibility of a certain attribute. For this we shall use the following symbols: +, -, # or ~ in order to signal if an attribute is, respectively, *public*, *private*, *protected* or *package*.
/	It indicates if an attribute is derived. A derived attribute is an attribute which can be calculated starting from other attributes of the class.

Table 2(cont.)

Attribute Name	Name or short phase which identifies the attribute. Attribute name starts with a small letter. When the name is compounded, each word starts with a capital letter and the blanks are thus eliminated.
type	The type of the attribute which in UML can be modelled via: data types; primitive data types as a particular instance of the data type; enumeration types as a particular case of the data types; classes. Data types are those types whose values do not have an individual identity. This is the main difference between these and classes whose objects always possess an identity. The identity of an object which is not always available involves the fact that the values of a certain data type cannot exist other than within the object they belong to. Moreover, each value exists only once. As well as the classes, the data type can contain attributes and operations. For the notation, we shall use they class symbol. In addition to this, the data type are characterised by the stereotype <<*datatype*>>. Generally, data types are used in order to describe structures. A *primitive datatype* is a predefined data type which has no structure. It is modelled with the class notation and it is characterized by the stereotype <<*primitive*>>. The primitive data types in UML are the following: Boolean (it can accept the values *true* and *false*); *String* (line of characters); *Integer* (real numbers); *Real* (real numbers); *Unlimited Natural* (natural numbers). Other types of primitive data are defined in the UML software products. An enumeration datatype defines a clear number of values. It is characterised by the stereotype <<*enumeration*>>. The values of the enumeration type can be freely chosen.
multiplicity	It specifies the number of values stocked in the attribute. Multiplicity can be absent (the synonym of a multiplicity equalling 1); it can be represented by a number or by an interval whose extreme values are specified between square brackets ([ ]) and separated by two dots (..) The character '*' is used as a supreme infinite form; used alone, this character symbolizes: 'bigger or equal to zero'.
Initial value	It determines the initial value that a newly created object accepts for this attribute. This value can be changed afterwards.

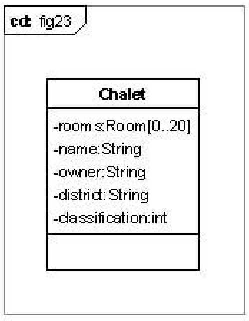
Figure 23 presents an example of class (`Chalet`) which emphasises the access modifiers, the multiplicity and the attributes of the class.

Remark. In order to be able to model the class `Chalet` one must know the classes `String` and `Room`. At the same time, one must be able to model multiplicity (between one and twenty rooms).

To an attribute one can also associate properties and restrictions. They are mentioned between accolades and separated by means of commas. In order to define restrictions we recommend either a natural language or the OCL language.

Example: *`{readOnly}-`* the attribute cannot be changed.

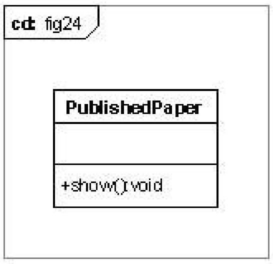
Figure 23



The natural properties of an attribute defined by means of UML are: *`readOnly`*; *`uniin`*; *`subsets <nameAttribute>`*.

Figure 24 unfolds an example of a class (PublsihedArticle) emphasising its operations.

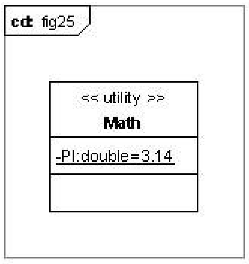
Figure 24



Static Attributes

Implicitly, the values of the attributes defined within a class differ from one object to another. Not in few cases, it is necessary to define an attribute storing a unique value which is partitioned by all the instances of the respective class. This attribute is static and it is called class attribute. For example, the constant *`PI`*, defined within the *`Math`* class of the Java language(figure 25).

Figure 25



Regardless of the object of this class, the value of *`PI`* remains unaltered. Moreover, this value is accessible even if there is no object pertaining to they *`Math`* class. In this case we may say that *`PI`* is an attribute of the *`Math`* class and not one of its instances. All the instances of the *`Math`* class have access to this attribute.

Graphically, a class attribute is underlined as shown in figure 25. In Java, C++, for example, a class attribute is accompanied by the key-word *`static`*.

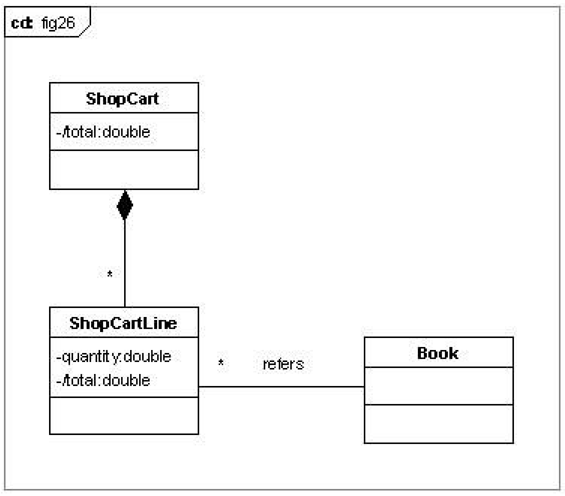
Derived Attributes

Derived attributes, symbolized by means of a slash (/) placed before their name, can be calculated any time starting from other attributes and the calculation formulae.

A derived attribute cannot be marked.

Figure 26 presents an example of a derived attribute (`/total`) which intervenes in the classes `ShopCart` and `ShopCartLine` within a diagram of the conceptual classes for the administration of the virtual shopping basket. An e-commerce application allows for the web navigators to search for books (works) taking into account the theme, the authors, the key-words etc. and thus to construct their own virtual basket which they shall later on order and pay for directly via the web.

Figure 26



Remarks:

- The class `ShopCartLine` represents lines of the virtual basket each corresponding to a book title but which share the attribute quantity.
- A line of the basket cannot pertain to more than one basket, this being determined by the overall number of lines. The destruction of the basket triggers the automatic destruction of all its lines (see the compounding relation).
- The attribute `/total` of the class `ShopCart` is different from the attribute `/total` of the class `ShopCartLine`. The two attributes differ in the sense that they pertain to different classes (polymorphism).

Methods and Operations

The behaviour of an object is modelled by means of a group of methods. Each method corresponds to a well specified implementation of a specific behaviour.

Generally, during the conception phase of the classes, one should always begin by setting up the headers of the methods without specifying the way they will be implemented. Before the instantiation of a given class we must formulate a potential implementation of its methods. Naturally, for each header more implementations are possible.

UML makes a distinction between the specification of a method corresponding to the definition of its header and the implementation of a method. The specification is called `<< operation>>` while the implementation is called `<< method>>`. As a consequence you can associate more methods to one operation.

Example: In order to implement the operation *`fact (n)`* representing n factorial (*`n!`*) we can use two methods : a recursive method- *`n* fact(n-1)`* and an iterative method- *`for(i=1; i<=n; i++) fact*=i`*

In a class, an operation (the same name and the same types of parameters) must be unique. When the name of an operation appears more than once with different parameters we can say that the operation (and/or method) is overloaded. Reversely, it is impossible for two operations not to be differentiated by means of the returned value.

Operations are placed within a separate compartment. Each operation is defined by means of a name, of parameters, return value and visibility (access modifier). The declaration syntax for an operation is presented in figure 27.

Figure 27

```
<access modifier><methodName>
<([parameterList])>:<returntype>
{properties}
<access modifier]:={+|-|#|~|}]
```

The specification of the syntactic elements of this notation is presented in table 3.

Table 3

Syntactic element	*Significance*
Access modifier	It indicates the visibility of an operation. We shall use the symbols +, -, # or ~ in order to indicate a visibility which may respectively be *`public`*, *`private`*, *`protected`* or *`package`*.
The name of the method	It allows for the brief description of the operation. An operation name is generally made of a verb which represents an action and eventually a complement. The UML specification recommends for the first letter of an operation to be small and for the complement to be eventually a succession of words starting in capital.
	The syntax of the list of parameters is the following: *`<direction> <name of Parameter>: <type> [multiplicity]= Implicit value`* *`<direction>`* shows that the parameter is used for entering (in); exiting (out); entering – exiting (inout) *`<Parameter_name>`* represents a name or a short expression which identifies the parameter. Usually, the name of the parameter starts in small letters the other words attached starting in capital.

Table 3.(cont)

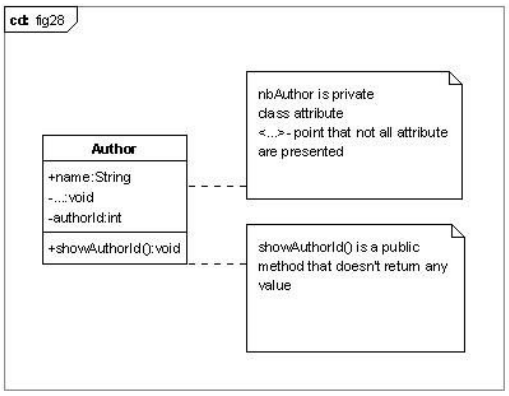
List of parameters (cont.)	`<type>` represents the type of parameter- primitive, class, interface etc. `<multiplicity>` specifies the number of instances provided by the parameter. `<Implicit value>` mentions the optional implicit value of the parameter `{properties}` specifies the properties of the parameters which must be presented between accolades. They are defined within the context of a specific model with some exceptions: `ordered`, `readOnly` and `unique`.
Return type	It specifies the type of information returned by the operation, if that exists. If the operation does not return any type of information (`int`, `long`, `float`, `String`, other type of returned datum), the returned type must be `void`.
{properties}	It specifies the restrictions and the properties associated to the operations.

Static Operations

As in the case of the static attributes (class attributes), we very often feel the need of defining an operation which does not depend on the values of each object but on the behaviour of the class itself. In this case too, the operation becomes the property of the class and not of its instances. It does not have access to the attributes of the objects of the class. Such an operation is considered to be static; it is directly induced by means of the class and not by means of its instances. Static operations or class operations are frequently used in the case of the utility operations which require none of the attributes of the class.

Graphically, a class method is emphasised as shown in the example from figure 28.

Figure 28


cd fig28
Author
+name:String
-...:void
-authorId:int
+showAuthorId():void
nbAuthor is private class attribute <...>- point that not all attribute are presented
showAuthorId() is a public method that doesn't return any value

Remark: The execution of an application creates more objects of the class *Author*. In order to simultaneously display the number of objects of this available class in the system you can add a class attribute which is to keep count of the number of authors and a method which display this number.

Interfaces

Principles and Definitions

The role of an interface is that of regrouping a set of operations which ensure a coherent service offered by the class.

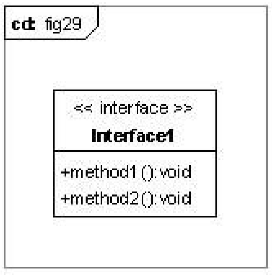
The concept of interface is used in order to sort the operations on categories without stating the way in which they are implemented.

The definition of an interface, contrary to the definition of a class does not specify an inner structure or the algorithms which allow for the accomplishment of its methods. It can state the conditions and the effects when it is used.

An interface does not have direct instances; it is impossible to directly instantiate an interface. In order to use it, an interface must be accomplished by means of a class. An interface can be specialised and/or generalised through other interfaces.

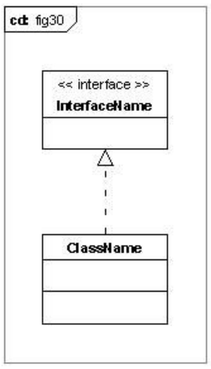
An interface is defined through a class which contains the same compartments. The main differences are the non-using of the key-word `abstract` (because all the interfaces and all its methods are abstract by definition) and adding the stereotype interface before the name of the interface (figure 29).

Figure 29



Accomplishing an interface by means of a class is graphically represented as a discontinuous line which ends in a triangular point (figure30).

Figure 30



A simple way of representing an interface is based on the notation displayed in figure 31 which uses a circle (lollipop) under which we place the name of the interface.

Figure 31

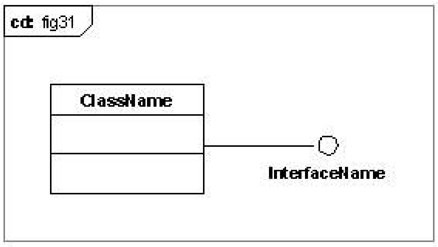
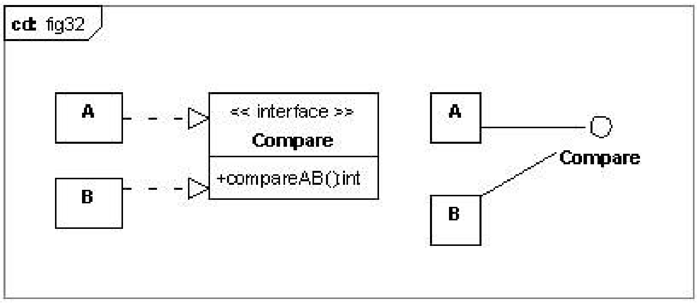


Figure 32 displays classes *A* and *B* which generate the interface *compare().* The objects belonging to the two classes must be comparable. The operation compounding the interface and allowing for the comparison of the instances belonging to classes *A* and *B* must be the same. Naturally, the comparison criteria are different. Figure 32, models this common interface, accomplished for each of the derived classes.

Figure 32

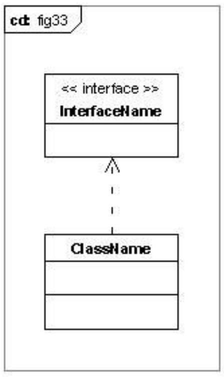


Remarks.

- The inheritance operation existing between the interface and the derived classes is called relation of realization (<<*realize*>>).
- A class can also depend on an interface in order to accomplish its operations. When a class depends on an interface (demanded interface) in order to accomplish its operations, it is called <<the client class of the interface>>.

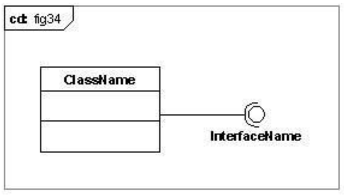
Graphically, it is represented by means of a dependency relation established between the class and the interface (figure 33).

Figure 33



An equivalent lollipop notation is also possible (figure 34).

Figure 34



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

Ciclul: UML PENTRU MANAGERI (IV)
Diagramele de clasă

Rezumat

În acest articol se prezintă conceptul UML de diagrame de clasă şi al notaţiilor UML asociate, prin intermediul a numeroase exemple. Aceste diagrame constituie nucleul oricărei aplicaţii software orientate obiect prin evidenţierea entităţilor participante şi a relaţiilor existente între ele.