1. Programming paradigms

The ***paradigm*** notion comes from Greek and it represents a example or a model. The ususal sense of this notion has been given by the historian Thomas Kuhn in his book "The Structure of Scientific Revolutions": a ***paradigm*** is a collection of theories, standards and methods, which represent a way to organize the knowledge.

Based on this notion, Robert Floyd in his paper "The Paradigms of Programming" defines the notion of ***programming paradigm*** as being a method to conceptualize the way of execution the calculations within a computer, and the structuring and organizing the tasks responsible with these calculations. Another notion used in place of programming paradigm is ***programming style***.

In the sense of programming paradigms, it is said that a programming language ***offers support*** for a programming paradigm, if it offer enough facilities that make it useful in this style. A programming language ***allows*** only this thing if the needed effort for writing a program in the respective style is greater, the language didn't offering enough facilities.

1.1 The procedural programming

This is one of the oldest and most used paradigms. Usually it supposes passing through the following steps:

a) the decomposition of the problem to be solved in smaller problems
b) finding for each small problem an optimal algorithm
c) implementing of each algorithm by using functions and procedures of an appropriate programming language

The oldest procedural programming language is FORTRAN, but most of the actual programming languages offer support for this kind of paradigm. The principal problems related with procedural programming concern the types of the function used (functions, procedures, small programs, routines, macros …), the type and the way of passing the parameters and the ways of calling.

Example 1.1. The definition and the using of a function which determines if an integer is prime:

```
int Prime(int n) {
  /*the code for the function */
}

void PrimeFactors(int n) {
   int i;
   for (i=2 ; i<n/2 ; i++) {
      if (n%i == 0 && Prime(i)) {
         printf("%d",i);
      }
   }
}
```

1.2 Data encapsulation

In time the accent in procedural programming moved from the function design to the data organization. The data are not regarded in isolation; they are regarded together with the functions which they process. The notion of ***module*** was defined as representing a set of related functions, together with data processed by functions.

So, a program can be divided in several modules in a clearer and more efficient way as the classic division in functions or procedures, the program data being hidden in the used modules.

Usually, a module contains an ***interface*** part where the data and the functions which are accessible from outside of the module, and also a part of the ***implementation***, which is inaccessible from outside of the module (they are locate inside of the module, where the functions of the module are defined).

The C language *allows* the modular programming, while Pascal or Modula –2 *offer real support* for such kind of programming technique. In the Turbo Pascal language the notion of module corresponds to that of ***unit***, the interface and implementation parts correspond to sections of ***interface*** and ***implementation*** respectively. In the case of C language the interface part is usually specified in a header file. This file must be included in all the others files of a program that use the module functions. The implementation part of the module is realized in a distinct file which must be included in the program project.

Example 1.2. The definition and the using of a module that allows the operations with integers.

```
//the file sequence.h : the interface
#define max_dim 100
void Init() ;
int Sum() ;
void Sort() ;
void AddElement(int) ;
void Print() ;

//the file sequence.c : the implementation
#include “sequence.h”
static int dim ;
static int v[max_dim] ;
void Init() { dim = 0 ; }
void AddElement(int k) { v[dim++] = k ; }
int Sum() {
  /* the code for the sum determination */
}
void Sort() {
  /* the code for sorting */
}
int Print() {
  /* the code for printing */
}
```

```
//the file pr.c : the using of the sequence module
#include “sequence.h”
void Processing() {
  int i, s, k, n = 0 ;
  Init() ;
  for(i=0 ; i<n ; i++) {
    scanf(“%d”, &k) ;
    AddElement(k) ;
  }
  s = Sum() ;
  printf(“\nSum=%d”, s) ;
  Sort() ;
  Print() ;
}
```

1.3 Data abstraction

In the previous example only one sequence of numbers is used. If it is desired to work with more sequences, the data encapsulation is not enough.

The notion of ***data abstraction*** supposes the possibility of defining the user data types, together with a set of specific operations for each type. The languages which offer support for data encapsulation allow also support for data abstraction, but generally they do not guarantee for it. For example, in the C language, heading files allow the declaration together of the data types and the functions they use.

Example 1.3. The definition in the C language of a data type representing the fractional numbers:

```
//the file fraction.h : interface
typedef struct {
  int p, q ;    /* the numerator and the denominator */
} fraction ;

fraction Sum(fraction, fraction) ;
fraction Multiplication(fraction, fraction) ;
fraction Division(fraction, fraction) ;

//the file fraction.c : implementation
#include “fraction.h”

fraction Sum(fraction f1, fraction f2) {
      fraction f ;
      f.p = f1.p * f2.q + f2.p * f1.q ;
      f.q = f1.q * f2.q ;
      return f ;
}

fraction Multiplication(fraction f1, fraction f2) {
      fraction f ;
      f.p = f1.p * f2.p ;
      f.q = f1.q * f2.q ;
      return f ;
```

```
}

fraction Division(fraction f1, fraction f2) {
      fraction f ;
      f.p = f1.p * f2.q ;
      f.q = f1.q * f2.p ;
      return f ;
}

//the file pr.c : utilization
#include “fraction.h”

void Processing() {
      fraction f1 = {1, 2}, f2 = {7, 4}, f3 ;
      f3 = Sum(Multiplication(f1, f2), Division(f1, f2)) ;
      printf(“\nf3 = %d/%d”, f3.p, f3.q) ;
}
```

Some languages such as Ada and C++ allow the users to define data type that work in the same way as the built-in types, because it is possible to use classes and operator overloading.

A ***class*** can be seen as an extension of the structure notion in the C language, which allows the definition in the class of the data and also of the functions that use this data. There are some specific categories of member functions. A ***constructor*** is a special function that allows initializing an object when it is created. A ***destructor*** is also a special function that allows deallocating additional memory of an object when it is destroyed. An ***operator overloading*** represents a function associated to a built-in operator whose significance is defined by the programmer, but whose call must corresponds to the language syntax.

Example 1.4. The redefinition in the C language of the user-defined type *fraction:*

```
//the file frac.h
//the definition of the fraction class
class fraction
{
  int p, q ;
public :
  //the class constructor
  fraction(int a = 0, int b = 1) { p = a ; q = b ; }
  //the declaration of the overloaded operators +, * and /
  friend operator + (fraction, fraction) ;
  friend operator * (fraction, fraction) ;
  friend operator / (fraction, fraction) ;
};

//the file frac.c
//the implementation of the fraction class
//the definition of the overloaded operators  +, * and /
#include “frac.h”

fraction operator+(fraction f1, fraction f2)
{
  fraction f ;
  f.p = f1.p * f2.q + f2.p * f1.q ;
```

```
  f.q = f1.q * f2.q ;
  return f ;
}

fraction operator*(fraction f1, fraction f2)
{
  fraction f ;
  f.p = f1.p * f2.p ;
  f.q = f1.q * f2.q ;
  return f ;
}

fraction operator/(fraction f1, fraction f2)
{
  fraction f ;
  f.p = f1.p / f2.q ;
  f.q = f1.q / f2.p ;
  return f ;
}

//the file pr.c
//the using if the class fraction
#include "frac.h"

void Processing()
{
  fraction f1(1, 2), f2(7, 4), f3 ;
  f3 = f1*f2 + f1/f2 ;
  // ...
}
```

An important problem concerning data abstraction is ***data parameterization***. Both the Ada and the C++ languages offer support for parameterization.

The parameterization appears when it is desired the definition of generic data types, where the type of the components is unspecified (it is generic, being seen as a parameter).

Example 1.5. For example the definition of a *vector* class where the type of the components is generic can be made in the C++ language in the following way:

```
template<class T>
class vector
{
  T *v;     //the components having the generic type T
  int dim;  //the vector dimension
public:
  //the constructor
  vector(int n)
  {
    if (n > 0)
    {
      v = new T[dim = n];
    }
  }
  //overloading of the [] operator
```

```
  T& operator[](int k) { return v[k]; }
  int dimension() { return dim; }
};
```

One can define now vectors that contain elements belonging to a specified type:

```
vector<int> v1(20);        //vector with 20 integer components
vector<double> v2(10);     //vector with 10 real components
v1[7] = 5;
v2[7] = 2.3;
```

1.4 Object oriented programming

As specified in the Section 1.3 the notion of class is specific to data abstraction paradigm, a class representing a way to implement an abstract data type. The properties of a class can be described by using both data (***attributes***) and functions (***methods***).

The instances of a class are represented by ***objects***, and from this point of view a class represents a way to describe the common properties of a set of objects. An object is uniquely identified by its name or by its reference. For example the definition of three distinct instances of the class *fraction* from the Example 1.4 can be made as follows:

```
fraction f1(1, 2), f2(7, 4), f3;
```

Because a class *C* encapsulates both data and functions, the methods of the class can be accessed for all instances of *C.* For example the statement:

```
f3.Print();
```

calls the method *Print* of the class *fraction*.

An object of a class is defined by its ***state***, which represents the values of its attributes at a certain moment. During the execution of a program the state of an object can be changed by changing the values of its attributes.

The ***behavior*** of an object is defined by the set of methods that can change its state. These methods are not private to the object; they are common to the class where the object belongs to. A method describes in fact how an object reacts when it receive a certain ***message***. A message represents a request for an object to invoke a specific method of the object. From this point of view sending a message to an object represents the calling of a specific method (function) of that object. A program that uses the data abstraction paradigm creates some objects that communicate by sending the messages.

The ***object-oriented programming paradigm*** uses the notions of classes and objects, but these notions are not specific to this paradigm. *A first essential element* of the object-oriented programming paradigm is the fact that it allows to *express the distinction between the general and the particular properties of the objects*. For example, all Dacia cars use 4 wheels (general property), but the Dacia 1310 cars have the engine capacity about 1300 cm^3 , while Dacia 1410 cars have the capacity of 1400 cm^3 (particular property).

It results some important properties of the object-oriented programming: this paradigm allows grouping the objects in ***classes***, and also it offers a mechanism for ***inheriting*** the properties of a class by another class. In this way classes can be related in a class hierarchy. For example,

in the case of Dacia cars, one can define a hierarchy formed by 3 classes, Dacia, Dacia 1310 and Dacia 1410, the last two classes inheriting the properties of the first class.

Example 1.6. The class hierarchy for the Dacia cars can be described as follows:

```
class Dacia
{
  // ...
  int nr_wheels ;
  Dacia() { nr_wheels = 4 ; }
  // ...
} ;

class Dacia1310 : public Dacia
{
  // ...
  int length ;
  int width ;
  int height ;
  string version ;
  int price ;
  Dacia1310() {
    length = /* … */; width = /* … */; height = /* … */;
    version = /* … */; price = /* … */;
  }
  // ...
} ;

class DaciaNova : public Dacia
{
  // ...
  int length ;
  int width ;
  int height ;
  string version ;
  int price ;
  DaciaNova() {
    length = /* … */; width = /* … */; height = /* … */;
    version = /* … */; price = /* … */;
  }
  // ...
} ;

class DaciaLogan : public Dacia
{
  // ...
  int length ;
  int width ;
  int height ;
  string version ;
  int price ;
  int nr_airbags ;
  DaciaLogan() {
    length = /* … */; width = /* … */; height = /* … */;
    version = /* … */; price = /* … */; nr_airbags = /* … */;
```

```
  }
  // ...
} ;
```

Example 1.7. Let us consider the following polygonal figures in the plane: triangles, quadrilaterals, pentagons, etc., each polygon being described by the coordinates of its vertices in trigonometric order. One can define a class hierarchy, which has the polygon class as the root class, the others classes inheriting the class polygon. Let us suppose that for the polygon class there are specified the coordinates for the first two vertexes of the *polygon*, P_0 (x_0, y_0) and P_1(x_1, y_1), the other classes having to memorize one after another the coordinates of the next vertex.

```
class Polygon
{
  // ...
public:
  Polygon(_x0, _y0, _x1, _y1);
  virtual ~Polygon();
  virtual double Perimeter() ;
protected:
  double x0, y0, x1, y1 ;
  virtual double TwoEdges() = 0 ;
  virtual double OneEdges() = 0 ;
  // ...
} ;

class Triangle : public Polygon
{
  // ...
public:
  Triangle(_x0, _y0, _x1, _y1, _x2, _y2);
  double Perimeter() ;
protected:
  double x2, y2 ;
  double TwoEdges () ;
  double OneEdge() ;
  // ...
} ;

class Qadrilateral : public Triangle
{
  // ...
public:
  Qadrilateral (_x0, _y0, _x1, _y1, _x2, _y2, _x3, _y3);
  double Perimeter() ;
protected:
  double x3, y3 ;
  double TwoEdges () ;
  double OneEdge () ;
  // ...
} ;
```

All polygons have a perimeter whose value can be determined in an incremental way by observing that the perimeter of a polygon with $n+1$ vertexes ($P_0, P_1, \ldots, P_n$) can be obtained from the perimeter of the polygon having n vertexes ($P_0, P_1, \ldots, P_{n-1}$) by adding the edges P_0P_n and $P_{n-1}P_n$. The *Perimeter* function determines the current perimeter value, and the *TwoEdges* function determines the lengths of the edges P_0P_n and $P_{n-1}P_n$.

These functions represent ***virtual functions*** because their definition is commune to all the classes but the implementations are specific to each class. In addition, the functions *TwoEdges* and *OneEdge* cannot be implemented in the class *Polygon* (in this class *TwoEdges* and *OneEdge* are ***pure virtual functions***).

The implementations of the above functions can be described as follows:

```
double Polygon::Perimeter()
{
  double l = sqrt((x0-x1)*(x0-x1) + (y0-y1)*(y0-y1)) ;
  return l ;
}

double Triangle:: TwoEdges ()
{
  double a, b ;
  a = sqrt((x0-x2)*(x0-x2) + (y0-y2)*(y0-y2)) ;
  b = sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2)) ;
  return a + b ;
}

double Triangle:: OneEdge ()
{
  double l;
  l= sqrt((x0-x1*(x0-x1 + (y0-y1*(y0-y1) ;
  return l;
}

double Triangle::Perimeter()
{
  double p ;
  p = Polygon::Perimeter() + TwoSides() ;
  return p ;
}

double Quadrilateral:: TwoEdges ()
{
  double a, b ;
  a = sqrt((x0-x3)*(x0-x3) + (y0-y3)*(y0-y3)) ;
  b = sqrt((x1-x3)*(x1-x3) + (y1-y3)*(y1-y3)) ;
  return a + b ;
}

double Quadrilateral::Perimeter()
{
  double p ;
  p = Triangle::Perimeter() - Triangle::OneSide() +
                TwoSides() ;
```

```
  return p ;
}
```

In the case of virtual functions, the effective selection of the function that will be called is realized automatically by the compiler. This property that allows the compiler to select a specific method of an object from several methods defined with the same name in a class hierarchy is called ***polymorphism***.

In conclusion, *the second essential element* of the object-oriented programming is the *polymorphism* and *the mechanism of virtual functions*, through which the calling of the member functions effectively depend on the object type.

For example, for the polygon case, the next declarations and instructions:

```
Polygon *f1 = new Triangle (0, 0, 1, 0, 0, 1);
Polygon *f2 = new Quadrilateral (0, 0, 1, 0, 0, 1, 1, 1);
double p1 = f1->Perimeter() ;
double p2 = f2->Perimeter() ;
```

allow the correct selection of the *Perimeter* function by the compiler for each object. The operator *new* is specific to the C++ language and it allows creating an object in the heap area of the memory image of a program (it returns a pointer to the new created object).

If it is desired the selection of a function from a specific class, the object oriented languages have a specific operator named ***scope resolution***. In the case of C++ language, the scope resolution operator is denoted by :: and is preceded by the class name. For example, the construction: *Triangle::Perimeter()* specific the calling of the *Perimeter* function from the *Triangle* class.

The language considered as a pure object oriented language is Smalltalk. Other languages which support this paradigm are Ada, C++, Simula, Java, etc.

Remark. A language who supports the object-oriented programming paradigm allows also data abstraction paradigm because of the class mechanism.

1.5 Object-Oriented Analysis and Design

Object-oriented languages represent useful tools for developing object-oriented programs, but developing software systems is a complex process involving several software processes. Usually the life-cycle of software systems contains the following phases:

- Requirement analysis
- Design
- Coding
- Testing
- Maintenance

Knowing a object-oriented language is useful in the (relatively simple) coding phase, while knowing methods for ***object-oriented analysis*** and ***object-oriented design*** are necessary in the earlier phases of the software systems developing especially in the requirement analysis and design phases.

One of the most used framework used in the requirement analysis and design phases for developing an object-oriented software application is the ***UML formalism*** (***The Unified Modeling Language***) proposed by Jacobson, Rumbaugh şi Booch. In the following there are presented some basic elements of the UML formalism.

During the requirement analysis there must be established the services that the customer requires from a system and the constraints under which it operates and is developed. When system requirements are identified a document specifying the detailed description of the system functionality and operational constraints must be produced. In this case the ***use case diagrams*** from the UML formalism can be used for describing the system functionalities. A use case diagram describes usually a functional requirement of the system from a user point of view. Use case diagrams uses the notions of ***actor*** and ***use case***.

An ***actor*** defines the system boundary. They are external entities (people or other systems) who interact with the system to achieve a desired goal. In UML formalism actors are drawn as stick figures. A ***use case*** is a procedure by which an actor may use the system. It is a collection of possible sequences of interactions between the system under discussion and its users (or actors), relating to a particular goal. Use cases made up of scenarios. Scenarios consist of a sequence of steps to achieve the goal; each step in a scenario is a sub goal of the use case. As such each sub goal represents either another use case (subordinate use case) or an autonomous action that is at the lowest level desired by our use case decomposition. In UML formalism an use case is drawn as a horizontal ellipse. ***Associations*** between actors and use cases are indicated in use case diagrams by solid lines. Additionally a ***system boundary box*** can contain several use cases in order to indicate the scope (boundary) of the system. As an example in Figure 1.1 it is presented a use case diagram that describes the functionality of an ATM. The actors are represented by the client and the bank that posses the ATM.

In order to specify all requirements of a system all use case diagrams that describe the entire functionality of the system must determined.
During the ***design*** phase the components of the software system must be determined. ***Components*** represent abstract entities and the system can be viewed as be made from these components. From the point of view of a programming language a component can be represented by a function, a class, or a collection of other components. In most cases components are represented by classes.

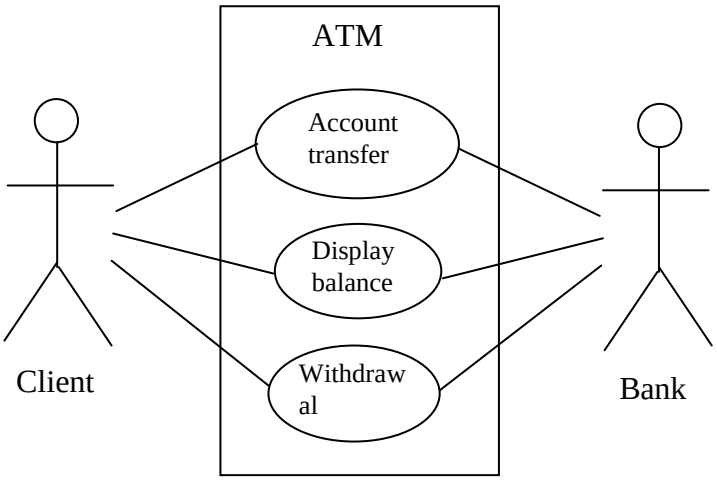

Figure 1.1

A component has several responsibilities, and it can interact with other components. A useful method (especially before the UML formalism) was the CRC method (`Component – Responsibility – Collaboration`). Using this method each component is associated to a card having associated the following information:

- The name of the component
- The description of its responsibilities
- The list of the other components interacting to the current component.

Because usually a CRC card represents a class its responsibilities are in fact public methods of the associated class. This method is very simple and it allows the refinement operation. However one disadvantage of the CRC method is the fact that it does not allow to specify the attributes of a class.

The UML formalism provides the ***class diagrams*** for describing classes and their relations. The class diagrams represent a static point of view of a system. The graphical UML notation for a class is a rectangle where it is specified the name of the class and the lists of its attributes and methods. The collaborations between classes can be specified by using the UML ***relations***. Relations represent a general notion in the UML formalism by relating different elements such as classes, actors, use cases, etc.

Some types of relations:

- *association*, which describes the semantic relations between objects;
- *generalization*, which describes the inheritance relation between classes;
- *realization*, which describes the relation between a specification and its implementation;
- *dependency*, which relates classes whose behavior or implementation affect each other.

The association is a relationship relating objects (not classes) and it represents the ability of one instance to send a message to another instance. A particular case of association is ***aggregation***, which is the typical whole/part relationship. From the graphical point of view aggregation is denoted with an open diamond placed at the end of the line reaching the "whole" class. A particular case of the aggregation relationship is the ***composition***. A composition relationship specifies that the lifetime of the "part" class is dependent on the lifetime of the "whole" class. From the graphical point of view aggregation is denoted with a black diamond placed at the end of the line reaching the composite class.

Figure 1.2 illustrates the aggregation and the composition relationships.

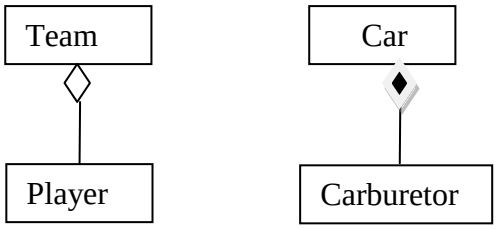

Figure 1.2 Aggregation and composition

Figure 1.3 illustrates the class hierarchy from the example 1.7.

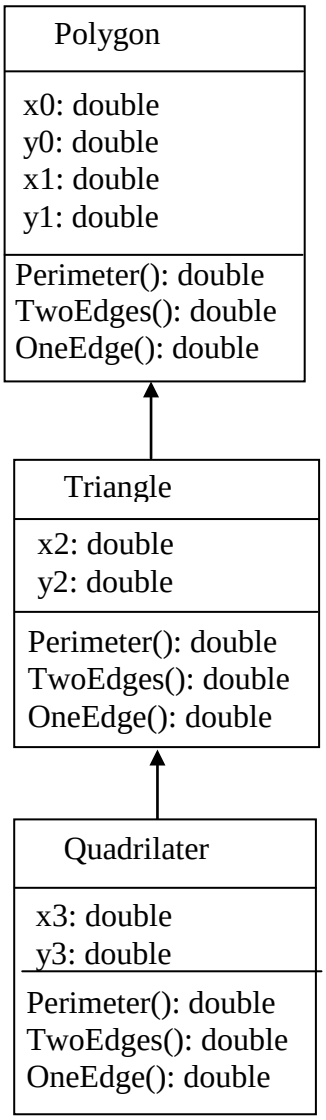

Figure 1.3 Class hierarchy of polygonal shapes

The UML formalism provides another types of diagrams that allow the description of object interactions and dynamic evolution of the objects. For example:

- ***collaboration diagrams***, that allow to represent objects and messages sent between them,
- ***sequence diagrams***, that contain the same information as Collaboration diagrams, but emphasize the sequence of the messages instead of the relationships between the objects.