End-User Class Definition in CAD Systems

Guillaume Texier, Fabrice Depaulis, Laurent Guittet

End-User Class Definition in CAD Systems

Guillaume Texier, Fabrice Depaulis, Laurent Guittet
Laboratoire d'Informatique Scientifique et Industrielle, ENSMA,
1 rue Clément Ader, 86961 Futuroscope Chasseneuil
http://www.lisi.ensma.fr/ihm
{texier,depaulis,guittet}@ensma.fr

Abstract

The object-oriented paradigm is very used in CAD systems. It permits users to create objects and to interrogate their attributes to use them in other processes. While some CAD or drawing systems support end-user programming in order to abstract building functions, none of them permit creating classes where several functions (constructor and selectors) share the same data. A data model that permits to abstract a class from one of its instances built by the end-user is described in this paper. The proposed technique permits the user not only to describe interactively the class constructor, but also to build the class selectors without any programming knowledge. The created class can be used directly thanks to a specific interpretation mechanism, or the corresponding code can be generated and compiled to have persistent classes. This technique has been used in a CAD system that permitting end-user specialization.

1. Introduction

In order to increase the usability of interactive systems, and particularly of Computer Aided Design (CAD) applications, customization seems unavoidable. In the meantime, this adaptation cannot be realized by end-users, who generally have no sufficient expertise in programming. In this paper, we describe how end-users with no programming knowledge may define true domain specific classes.

A class can be viewed as a description of a set of objects that share the same structure and the same behavior. It is composed of a set of attributes (properties) and a set of subroutines or functions. Each class owns two components [6]:

- The static component is composed of attributes, e.g. named fields with values. Object states are represented by these fields.
- The dynamic component is characterized by the methods that represent the common behavior of objects that belong to the same class. Methods are used to handle the object fields. They represent the actions that can be done on or by objects, and permit the transitions between the states attributes describe.

Defining interactively new classes consists in describing both attributes and methods. Methods can be structured into three categories:

- Constructors are used to create class objects. They consist in assigning values to class attributes. The values come from input parameters.
- Selectors are used to recover class attributes of objects. They generally have no parameter.
- Modifying methods are used to change the class attribute values. They usually have input parameters.

In order to create new classes, the main difficulty comes from the fact that several functions that share the same data have to be defined interactively. The goal of this paper is to propose a specific data structure that permits to overcome this problem.

Two kinds of techniques permit end-users to create programs interactively. On the one hand, *programming by demonstration*[3], which from the Human Computer Interaction research field, consists in recording user interaction in order to abstract programs. Another definition might be creating a program from an example of its execution. On the other hand, a technique to record the building process of geometric objects in order to reevaluate it with different data has been proposed by geometric modeling community. This technique is called *parametric geometry* [10]. Even if these two techniques permit recording programs, they cannot be directly used to describe classes. Actually, they cannot be used to group different methods together in one class and then to describe the attributes of that class.

The solution we suggest in this paper is based on a parametric model augmented by abstraction principles that come from programming by demonstration techniques. In the next section, the advantages and drawbacks of *programming by demonstration* and *parametric geometry* are studied in the perspective of interactive class definition. The third section is dedicated to the description of our approach. It focuses on the way constructors and attributes may be described. Section 4 details the generalization phase, while section 5 details an example.

2. End-user programming methods

End-user programming methods generally consist in abstracting a program from an example of its execution. In our context, it means that a class may be abstracted from one of its instances. So the user has to define not only how the object is built (abstracting the class constructor) but also the way the attributes are computed (abstracting the selectors). Two slightly different approaches have been developed to capture a program without explicit programming. The first one is known as *programming by demonstration*, the second one is called *parametric geometry*.

2.1 Programming by demonstration

Programming by demonstration (PBD) has been introduced to permit end-users to create programs without explicit programming. This technique is used in several domains: games, desktop applications, drawing applications, and so on. For example the Topaz system [7] creates macros that can generalize some drawing actions. Other systems are able to generate programs in neutral language that can be used by other systems. For example the EBP system [8] generates FORTRAN programs for exchanging CAD geometry.

Programming by demonstration means creating a program using an example of its execution. An important notion brought by the programming by demonstration technique is the *abstraction method*. In classical programming languages, the program manipulates variables by their names. Conversely, PbD allows the "program" to directly manipulate variables through their values. The link between names and values is made in a symbol table, called the context [8]. Thus, every program variable is referenced without ambiguity regardless of its value. Creating a program, that deals only with variable names, from an example of its execution needs to associate each variable name with each value. This is the task of the *dynamic context*. For each created value in the example, a new variable name (whose data type is defined by the value) is added to the dynamic context. These variable names may be used in the generated program.

The ability to specify program parameters must also be provided by programming by demonstration systems. There are two main approaches to define the parameters of programs constructed by demonstration. The *implicit manner* described by Bauer [1] or Lieberman [5] consists in creating several examples of the same program. Then, the system automatically infers the parameters after an analysis of the different examples. With the *explicit manner*, the PBD user has to define which values are program parameters. The other values are considered by the PBD system as constants. This paper follows the second approach.

The two main advantages of PBD are the program abstraction and the parameter definition. For our purpose, PBD might be a good approach; for example, Mondrian [4] allows end-users to create new objects and to include them as native objects in the interactive system. However, it does not allow the user to define parts or attributes onto the designed objects. The main drawback of programming by demonstration regarding class definition is that in re-execution mode, the internal data of the program cannot be accessed by other programs. This is a real difficulty for designing class attributes.

2.2 Parametric geometry

The parametric geometry comes from the CAD field. In this area, geometrical objects are often made by constrained constructs; for example a line can be explicitly parallel to another line [10, 11]. The goal of parametric geometry is to permit the dynamic modification of geometrical entities; for example if a circle is created using the intersection of two lines as center (Figure 1), the modification of one of these lines leads to move the circle.

Figure 1 shows an example of a parametric construct: a circle whose radius is calculated an another circle radius and whose center is the intersection of two lines. In parametric geometry, each entity records the sequence of functions that are used for its creation. This sequence of functions can be seen as a building tree like in Figure 1, and might be seen as a good starting point for program creation. Unfortunately, most of CAD systems that use parametric geometry do not really have the necessary tools to abstract programs. In parametric geometry, there is no notion of parameters or constants: all the nodes of the tree can be modified and then the objects that depend on the modified nodes are re-evaluated.

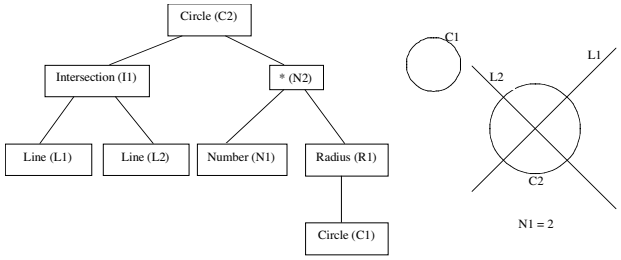

Figure 1: Building tree in a parametric construct

The main advantage of this approach is that it permits a program (a building process) to access internal data (building process nodes) of another program. So this technique can be easily used to define classes with constructors and selectors. But, this method has two drawbacks. The first one concerns the abstraction of the program. Extracting independent programs from the recorded sequence of functions is rarely given to the user. The second one is that few systems offer the possibility to define program parameters. Parametric geometry does not offer the possibility to explicitly define the signature of classes.

3. Data model for class definition

We propose a model based on a functional parametric model (which allows recording the building process), enhanced with attributes process definition.

3.1 The parametric model

The main characteristic of the parametric models regarding the classical geometric models is the possibility to save the building process of each geometrical entity.

Figure 2 shows an EXPRESS [9] schema that represents the parametric model. Rectangles represent classes, thin lines represent attributes, and thick lines represent inheritance relations (not used on Figure 2). The *Object* class owns an attribute called *Parameters* which is a list of *Object.*

Each class of our model have an optional link to a geometric, numeric or textual value which depends on the class (the current instance attribute). As an example, the Circle class has a link with the geometric circle class whereas the Number class is linked to the standard float class. This link is optional because during the construction, some geometrical entity could be destroyed. For example, two solids used in a boolean operation can be deleted from the data base during this operation. But, their representation in the parametric model must be persistent while objects refer to them in their building process.

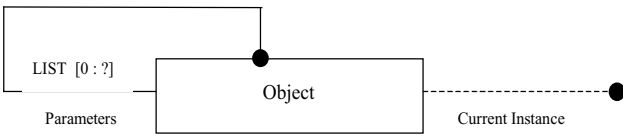

Figure 2 : Parametric model class

The attribute named parameter represents the list of parametric model instances used to create an instance. Then, every parametric model instance knows its parameters. The set of these linked instances can be represented as a tree (see Figure 1), and when one node of this tree is modified, the whole tree is recomputed.

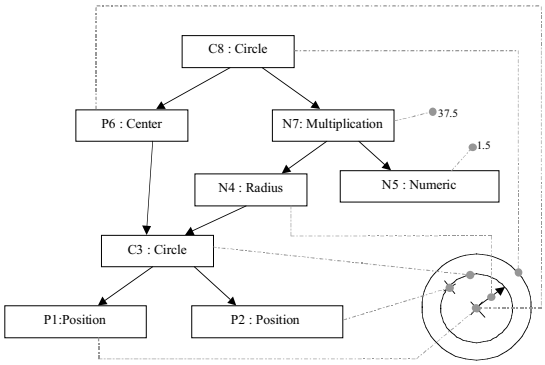

Figure 3: Building tree example: the wheel

The building tree of an entity made of two concentric circles is shown on Figure 3. The C3 circle is built using two literal positions (*literal data* are data directly provided by the end-user). The C8 circle has the same center as C3 and its radius is one time and a half bigger. The parametric model instances are represented by rectangles on the figure. The link between the objects and their parameters are represented by arrows. The mixed lines show each object current instance.

Contrariwise the usual terminology, the building process is not exactly a tree but a Direct Acyclic Graph (DAG). The C3 object (see Figure 3) is used several times in the building process.

This kind of building DAG is used by our class definition model to describe all the construction or computation processes.

3.2 Class definition

Defining interactively new classes requires solving two problems:

1. Defining constructor parameters in order to permit the instantiation of different objects with different values,
2. Defining object attributes and selectors that may be used by other processes.

3.2.1 Constructor definition. Our proposition consists in abstracting the class constructor from a parametric object that represents an instance of this class as it is made in programming by demonstration.

As seen in the previous section, each parametric model object keeps its own building process in a DAG which leaves are values directly provided by the user. In order to define a constructor, this building process is used, and the effective parameters of the instance are identified among the nodes and the leaves of the DAG. A boolean attribute (*Class_Constructor_Param*) (see Figure 4) has been introduced. It indicates if a parametric model object represents a class constructor parameter. The default value is false. Then, by default, all the leaves of the building DAG are considered as constants and the nodes are considered as local variables. It is up to the user to decide which are the instance constructor parameters among the objects the DAG of the example contains.

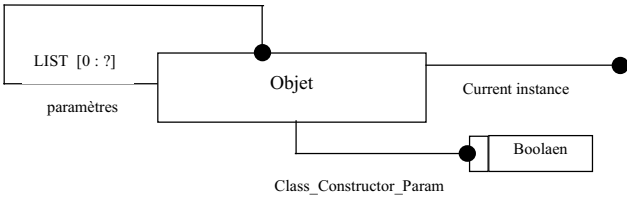

Figure 4 : Class parameter notion in the parametric model

Let us detail the example shown on Figure 3, called *wheel*: the parameters of this entity are the center of the circles and a point from the small circle.

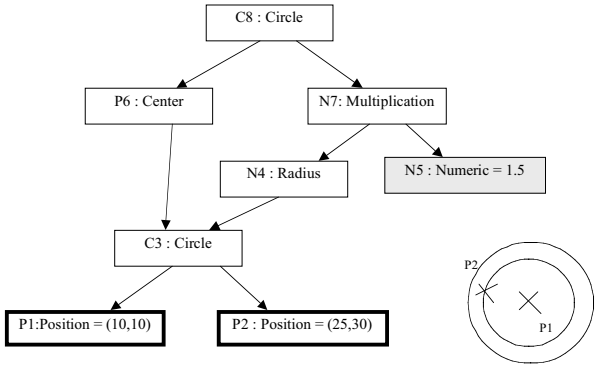

Figure 5 : Wheel constants and parameters

On Figure 5, the building process parameters of the entity are represented by thick rectangles. The grayed one indicates a constant and the others are local variables of the building process. So, in this example, the relation between the radius of the two circle will have always the same value whatever the value of the two parameters.

When the class is generated from the DAG, the objects that represent effective parameters are transformed in formal parameters (i.e. the generator does not take into account their own building tree, the expression used to provide their values is ignored when the class constructor is generated). For example, if the user indicates that the center of a circle is a parameter, then, at generation time, a position object replaces this center object in the class constructor.

3.2.2 Attributes definition. The possibility to define attributes and their computation functions is essential to define classes although, at our knowledge, it is not provided by any programming by demonstration or parametric system. Passing from interactive program description to interactive class definition requires offering the user the possibility to describe not only the building process of the objects, but also processes able to access the attributes of objects. Object attributes, as the center or the radius of a circle for example, have an essential role in parametric models. Related to PBD works, they act for the user intent, which is explicitly given by CAD users during the drawing phase.

The method we propose for the definition of an attribute and its computation function consists in associating a parametric model object to a string and a parametric building tree. The name of the attribute is represented by the string. The object holds the type of the attribute and its computation function as a building tree.

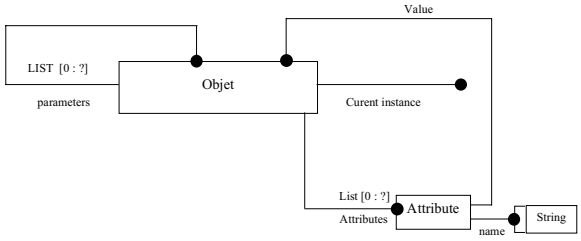

Figure 6: Attribute notion in our parametric model

Figure 6 shows the modification adduced to the parametric model in order to define attributes. The list of class attributes is defined by the attribute named *Attribute*. An attribute is defined by its *name* which is a character string and by a parametric model object representing its type. This object has its own building DAG as all the

objects of the parametric model. Then, when the value of a parameter of the instance is modified, the instance value is re evaluate and also the values of its attributes.

As an example, an attribute that represents the distance between the two wires of the circles is added to the entity *Wheel* described on Figure 5. This attribute is named *Tire Size*. The link between the C8 object (the wheel) and its attribute is represented by the triple arrow on Figure 7. It is important to notice that the leaves of the attribute building DAG are either class building DAG nodes or constants. Indeed, the attributes of an object represent the internal structure of this object. The values of the attributes have to be computed at object instanciation time. So, these values computation must only rely to constants or values of object building parameters.

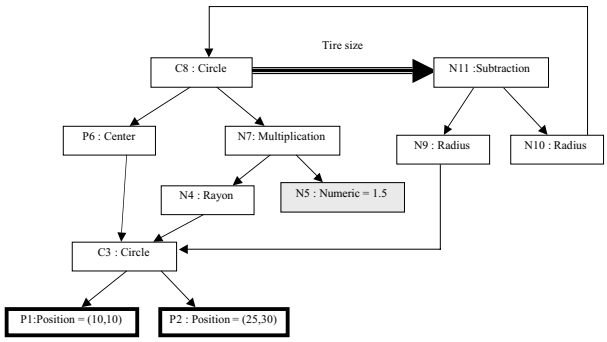

Figure 7 : Tire Size attribute definition

3.2.3 Constructor generation. In order to abstract the class constructor from the building DAG of the instance example, the internal variables, the constants and the parameters contained in the building DAG must be distinguished. The formal parameter definition is realized using the list of objects that represent constructor parameters. Every object from the building DAG which is neither a class parameter nor a computation result is considered as a constant in the building process.

The transformation of the effective parameters of the example instance in formal parameters of the class entails differences between the class constructor DAG and the instance building DAG. Indeed, the computation processes of the values of the effective parameters are not taken into account in class constructor DAG. Thus, the following rule can be stated:

An object named O belongs to a class constructor DAG only if a path between the root object and O (not included) exists and contains no parameters.

Figure 8 shows the way a class constructor DAG is abstracted from the building DAG of an example instance. Parameters are represented by thick rectangles, constants by grayed rectangles, and the remainder are internal variables. The objects used at creation time to affect values to parameters are not taken into account (the objects named O0 and O1 do not appear in the class constructor DAG and the link between O4 and O2 has been destroyed).

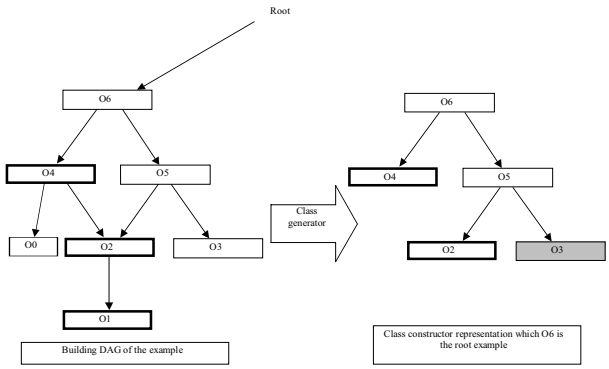

Figure 8 : Constructor DAG abstraction

3.2.4 Attributes generation. Contrariwise to the class constructors, the attribute computation functions do not have parameters. They only use either constants or objects contained in the class constructor DAG. In order to abstract a computation function from the object that represents an attribute value in the example, the class generator browses the attribute definition DAG from the root until it finds either constants or objects of the class constructor DAG. Then, the DAG that represents the computation function of the attribute is created using parametric objects.

4. Application

Once the class has been defined by the user, it can be used following two ways. The first one consists in directly using the generated DAG to create new instances. The second one regards the generation of the code that corresponds to the class.

4.1 Direct instanciation

The re interpretation of the class interactively created permit the user to test the new classes. Then, He/She can immediately correct the design mistakes because He/She can always access the example and re generate the class. This re interpretation stage can be compared to the debug stage in classical programming.

The class instanciation consists in copying the parametric structure of the example. The whole building DAG, the constructor DAG and the computation DAG of the attributes are duplicated without their links to the geometric model. Then, a non-evaluated tree that represents the class is created. Every time the end-user creates a new instance of the class, the non evaluated tree is duplicated and the objects that represents formal parameters are substituted by objects (the effective parameters) provided by the user. At last, the tree is evaluated using effective parameter values, which results in creating the new instance (see Figure 9).

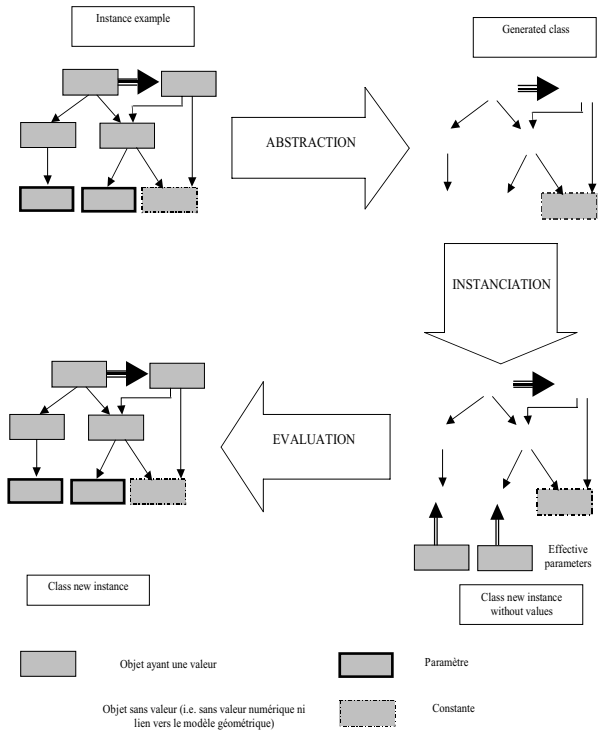

Figure 9 : Dynamic instanciation

Figure 9 illustrates the dynamic creation of a new class from an example of its instances. This interactive definition method is similar to the abstraction method described in programming by demonstration, which consists in creating a program from an example of its execution. In oriented object paradigm, this kind of instanciation is called prototyping [2].

4.2 Code generation

Once the class is being tested using the direct instanciation mode, it must be recorded in a persistent form. This is done by applying standard compilation techniques while crossing the DAG structure.

The code generator can be used either to define new classes for application specialization or to exchange data between heterogeneous parametric systems.

5. Example of application : TexAO

The TexAO system is a “light” CAD system that permits end-users to define new classes and then, to integrate them in the system in order to specialize it. This application uses the model described upper to generate new classes. The following example demonstrates how easy it is for end-user to define new classes and to specialize his/her application.

5.1 The motion rod

In order to present how our system works, a typical example of mechanical entity, a motion rod, is used. The drawing is shown in Figure 10, exactly as the end-user realizes it. We can describe it using the class paradigm. So, it might be composed of:

- A constructor with three parameters that represent two characteristic points (P1 and P2) of the motion rod and its thickness (H),
- A numerical attribute which stands for the motion rod length,
- A geometrical attribute which represents the center of the motion rod.

This two attributes are calculated from the remainder of the drawing.

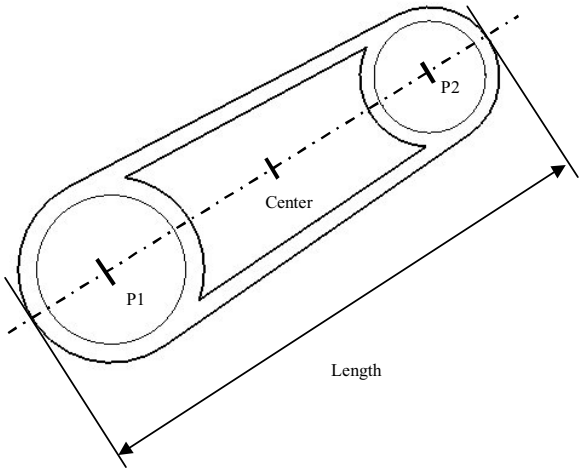

Figure 10 : Motion rod definition

5.2 Building the example

TexAO essentially reacts as any parametric system but it allows associating parameters and a signature to any constructive process, in order to convert it into a class.

Parameters may be defined by two methods:

- *a prior,i* by creating a new object which will be a parameter of any construction that may use it in the future,
- *a posteriori,* by selecting existing objects that become parameters of the construction that references them.

While our users are expert users, they generally prefer the first solution: the class extraction process is completely deterministic, and end-users perfectly know what they are doing.

In order to create a new parameter, the end-user activates a specific command (make parameter)(see Figure 11) and selects or creates a object that stands for the effective value of the parameter in the example. The parameter name is provided using a modal dialog box. Then, a new button that represents the parameter is created in the interface (Figure 11). It is used by the user to select the parameter and to use it in any construction.

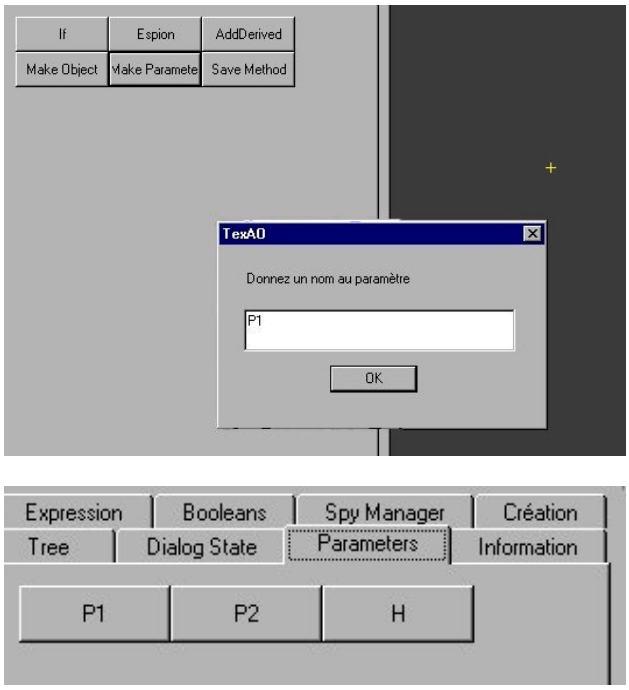

Figure 11 : Parameters definition

The attribute definition is realized in the same way as the parameter definition. The end-user selects a specific command "*add attribute*", then he/she selects or creates the value of the attribute and then the system asks him/her for providing the attribute name with a specific dialog box (Figure 12).

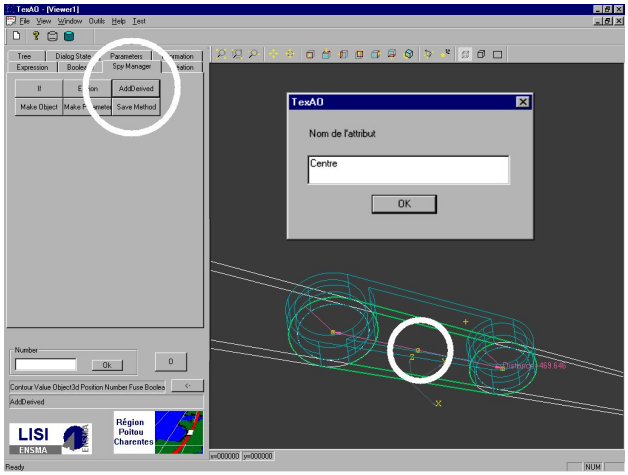

Figure 12 : Attribute definition

At last, the user selects the "*make object*" command to generate the class. The generation begins as soon as the user has provided the root object that represents the example and the name of the class. Then, new commands are automatically added to the interface in order to permit the user to handle the new class (for the constructor and for the attribute computation functions).

The TexAO system has now a new class called Motion Rod in its specific component. This class acts exactly as native system classes. The end user has specialized its application without explicit programming but only by demonstrating an example for the new class.

6. Conclusion

The interactive definition of new object classes is an important goal to achieve in order to permit end-users specialization of interactive applications. A class is composed of two main parts, the constructor used to create new instances and the attributes selectors used to access internal values of these instances.

Two great methods allow end-users to define programs:

- Programming by demonstration provides techniques able to abstract a program from an example of its execution, and allow the specification of the program signature (i.e. distinction between parameters, constants and internal values)
- Parametric geometry is able to preserve in a specific structure the building process of any geometrical or

topological entity in order to re evaluate it when one of its values is modified.

Unfortunately, none of these techniques provides the necessary structure to define new classes, and more particularly class attributes. Programs created by programming by demonstration are independent and cannot generally share internal values. Parametric models offer sometimes the possibility to associate a signature to the building process, but they cannot associate several computation processes to a single object.

Our solution to define interactively new classes is based on a standard parametric model augmented with specific characteristics that permit to get together in a same structure several building processes. More precisely:

1. The possibility to distinguish parameters, constants and internal variables (which comes from programming by demonstration) has been added. Thus, it is now possible to define real signature for class constructors and for attribute functions.
2. A mechanism permits the association of attributes to the class. In fact, each attribute is defined by a parametric function that can access any constructor value to generate its proper value. This function is the attribute selector in the class.

Our solution presents two important advantages. First, it can be applied to any parametric model able to store some functional building process. Second, end-users do not need any programming or algorithm expertise to create complete classes, they simply use a conventional parametric CAD system in a particular way.

This class definition model has been implemented in a small CAD system that owns all the standard functionalities in order to show that our method can be really applied. It might be very interesting to evaluate this approach with real users. The need for specific (firm dependent) CAD components is very strong, and our approach might solve this problem.

In the future, we think about adding the possibility to define methods to modify the attribute. And, we will try to use our approach to interactively define forms features. Another interesting point is to study how this method can be applied to other kinds of systems such as games or desktop applications.

7. Bibliography

[1]Bauer, M.A. (1979) *Programming by Examples*. *Articial Intelligence*. 1979, pp. 1-21.

[2]Cohen, B. et Murphy, G.L. (1984)*Models of Concept*.

[3]Cypher, A. (1993)*Watch What I Do: Programming by Demonstration*. The MIT Press, Cambridge, Massachusetts

[4]Lieberman, H. (1993)*Mondrian: a Teachable Editor*. Cypher, A. (Ed.). In *Watch What I Do: Programming by Demonstration*, The MIT Press, Cambridge, Massachusetts, pp. 341-360.

[5]Lieberman, H. (1993)*Tinker: A Programming by Demonstration System for Beginning Programmers*. Cypher, A. (Ed.). In *Watch What I Do: Programming by Demonstration*, The MIT Press, Cambridge, Massachusetts, pp. 49-66.

[6]Masini, G., Napoli, A., Colnet, D., Léonard, D. et Tombre, K. (1989) *Les Langages à Objets*. InterEditions, Paris.

[7]Myers, A.B. (1998)*Scripting Graphical Applications by Demonstration*. In *Proceedings of Human Factors in Computing Systems (CHI'98)* (18-23 April, Los Angeles, Californie), ACM/SIGCHI, pp. 534-541.

[8]Pierra, G., Potier, J.-C. et Girard, P. (1996)*The EBP system : Example Based Programming for Parametric Design*. Teixeira, J. et Rix, J. (Ed.). In *Modelling and Graphics in Science and Technology*, Springer-Verlag, 124-140.

[9]Schenck, D. et Wilson, P. (1994) *Information Modelling The EXPRESS Way*. Oxford University Press,

[10]Shah, J.J. et Mäntylä, M. (1995) *Parametric and Feature-based CAD/CAM: Concepts, Techniques and Applications*. John Wiley & Sons, New York.

[11]Zalik, B. (1996)*An Interactive Constraint-Based Graphics System with Partially Constrained Form-Features*. In *Proceedings of Computer-Aided Design of User iterface (CADUI'96)* (5-7 Juin, Namur, Belgium), Presse Universitaire de Namur, pp. 129-139.