Compilation of Inheritance for Object-Oriented Structures

ARON CRISTINA ELENA
Department of Computer Science and Economic Informatics
University Lucian Blaga Sibiu, Faculty of Sciences
Str. Dr. Ion Ratiu 5-7, 550012, Sibiu
ROMANIA

Abstract: - Software systems are becoming increasingly complex and large. Thus, there is a growing need to make the development of such systems more efficient and more transparent. The ultimate objective is to construct software systems from read-made standard building blocks. Attempts to progress towards this objective cover the following areas (among others): modularization, reusability of modules, extensibility of modules, abstraction. Object-oriented languages afford new possibilities in these areas. Thus, object orientation is viewed as an important paradigm in relation to management of the complexity of software systems.

Key-Words: - object, class, method, inheritance

1 Introduction

Object-oriented languages introduce a new modularization unit: object classes. Object classes may encapsulate both data and functions operating on this data. The consideration of both data and functions in one unit permits the implementation of natural, self-contained modularization units which are easy to integrate and extend.
The concept of inheritance is a powerful and convenient method for extending and constructing variants of existing modules (object classes).
The type system of object-oriented languages uses the inheritance concept: inheriting classes become subtypes of the base classes, and their objects can be used almost everywhere where objects of the base classes are permissible. Considered strictly from this point of view, they thus inherit the application area of their parents and are therefore very easy to integrate into existing systems.
Inheritance hierarchies introduce different levels of abstraction into programs. This means that at various points within a program or system, it is possible to work at different levels of abstraction, as required.
Abstract classes can be used in specifications, refined by gradual inheritance, and, finally, implemented.
Genericity permits the parameterization of class definitions. For examples, algorithms and associated data structures such as lists, stacks, queues, sets … can be implemented in this way, independently of the element data type.

2 Problem Formulation

2.1 Inheritance

Inheritance is defined as the incorporation of all features of a class A into a new class B. B may also define others features and, under certain conditions, overwrite methods inherited forma A. Some languages permit the renaming of the inherited features to avoid name conflicts or simply to allow more meaningful names in the new context.
If B inherits from A, then the class B is derived from class A and A is called a base class for B.
Inheritance is one of the most important of object-oriented languages. It makes extension and formation very easy. The resulting inheritance hierarchy also permits a structuring of class libraries and the introduction of different levels of abstraction.
Figure 1 shows a section of the inheritance hierarchy of a class library, with the object classes*: graphical objects, closed graphical objects, ellipse, polyline, polygon, rectangle and triangle*.
In this figure, object classes are shown by ellipses, which contain the name of the object together with a selection of the methods introduced by this class. Inheritance is represented by an arrow. For example, the class graphical object introduces the methods translate and scale: all the graphical objects can be *translated* and *scaled*. The class *closed graphical* and *polyline* inherit these methods form *graphical object*. In *polyline*, the inherited methods *translate* and *scale*, are overwritten and the method length

(the length of the *polyline*) is introduced. *Closed graphical* introduces a new method area which gives the area enclosed by the closed graphical object. The class *polygon* inherits from both *closed graphical* and *polyline*; *area* is overwritten. Finally, *rectangle* inherits from *polygon* and overwrites *area*.

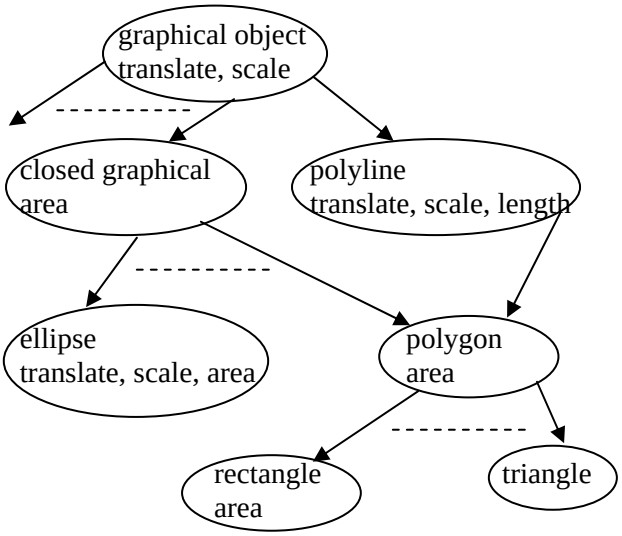

Figure 1

While the methods *translate* and *scale* are introduced in the class *graphical object* they are not yet defined there. The methods of translation and scaling can only be defined when graphical objects are firs given a concrete representation (in our example, in the class *ellipse* and *polyline*). The fact that these methods are nevertheless introduced in *graphical object* is intended to indicate that every graphical object must process these methods, although nothing can be said about this implementation. Classes that contain undefined methods are said to be abstract. They do not contain any object of there own, that is, objects that do not come from a proper subclass.

Analogously, the class *closed graphical* introduces the method *area*, without being able to define it. Area is firs definable in the class *ellipse* and *polygon*. However, the surface-area calculation of general polygons is complex and is based on a triangulation of the surface area enclosed by de polygon and a summation of the surface areas of the resulting triangles. On the other hand, the surface-area calculation for a rectangle is trivial. If rectangles are frequently used (and furthermore, in the application area under consideration, there surface areas often have to be determined) it is expedient to implement *area* in *rectangle* in a new and more efficient way, as we have done in our example. However, for polygons, it is best to take over all the methods of polyline without alteration.

We see that the concept of inheritance provides us with the possibility of reusing parts of an existing implementation in a simple way, extending them and, if need be, adapting them locally to particular requirements or circumstances by overwriting individual methods.

Moreover, we retain the facility to define abstract classes. This leads to flexibility in programming languages similar to that achieved in natural languages by means of abstract concepts: we retain different levels of abstraction. Anything that can be formulated at a high level for abstraction tends to have a wide area of application and thus is to a large extending reusable. Thus, we shall attempt to use as high a level of abstraction as possible. On the other hand, we are occasionally forced to move to a more concrete level, where more structure is available for the solution of solution of specific tasks. Let us suppose that a transformation of a graphical object can be described by a sequence of translations, scaling and (possibly) other operations defined for all graphical objects; then these transformation can be implemented by a single function, applicable to all graphical objects regardless of there type. If we had no abstract classes (but type checking by the compiler), then we would need a separate function for each specific class, although the implementations of these functions would always have the same appearance.

Thus, typed object-oriented languages take account of the inheritance hierarchy in there type system. If a class B inherits from a class A then the type assigned to B is a subtype of the type assigned to A. Every object of a subtype is automatically also an element of the super-type; an inheriting class is a subclass of the class from which it inherits. This has the following effect:

Subtype rule: When an object of a certain type is required at an input position (function input parameter, right-hand side of assignments) or as a function return value, objects of any subtype are allowed.

We shall call objects of B that are not also objects of a proper subset of B the **proper objects** of B or the **proper** B object. We refer to B as the **runtime type** of the **proper** B object. Thus, each object has a uniquely determined runtime type, which is the smallest type to witch that objects belongs. In addition, it is an element of every super-type of its runtime type.

By virtue of the subtype rule, methods and functions in object-oriented languages can be accepting objects with different runtime types and thus different structures, for example in a parameter position. This is a form of **polymorphism**.

The subtype rules of inheritance, together with the possibility that inheriting classes may overwrite an inherited method, have an interesting consequence which is important for compilers. By way of example, let us consider a function f witch permits objects of the class *closed graphical* as a parameter and let us suppose that it calls the area method of these parameter. Since *closed graphical* cannot define the area method, it is evident that the area method of the parameter and not that of the class *closed graphical* must be called in f. The following is generally true:

Method-selection rule: If class B inherits from class A and overwrites method **m** then for B objects b the definition of m given by B (or a subclass) *must* be used even if b used as an A object.

This rule presents a compiler with a problem: it has to generate code to activate a method it does not know at compile time. Consequently, for example, in generating code for f, the compiler cannot bind the method name area to a specific method. This binding can only be performed when the actual parameter is known; that is, generally, at program run time. We therefore speak of **dynamic binding** in contrast to **static binding** where the compiler carries out the binding itself. Thus, we can also formulate the method-selection rule as follows:

Dynamic-binding rule: A method of an object o, which can potentially be overwritten in a subclass, has to be bound dynamically if the compiler cannot determine the runtime of o.

The major part of this section is concerned with an efficient implementation of inheritance.

2.2 A compilation scheme for simple inheritance

In a language with simple inheritance, each class can inherit from at most one class. The inheritance hierarchies of such languages are trees or forests.

The mechanism outlined in this section for compiling simple inheritance can also be used relatively simply and directly in the imperative languages such as C. This also makes possible to use object-oriented structuring (including simple inheritance) in imperative languages.

The following program describes the class *polyline* and the derived class rectangle.

```
#include “graphical_object.h”
/*imported “graphical_objects” */

#include “list.h”                    /*imported lists*/
#include “point.h”       /*imported points*/

class polyline: public graphical_object {
 List<point> points;
public:
void translate (double x_offset, double y_offset);
virtual void scale (double factor);
virtual double length (void);
};
```

The class polyline

```
#include “polyline.h”

class rectangle: public polyline {
 double side1_length, side2_length;
public:
rectangle (double s1_len, double s2_len, double x_angle=0);
void scale (double factor);
void length (void);
};
```

The class rectangle

The definition of *rectangle* assumes that, for efficiency reasons, the lengths of the two sides of a rectangle are stored in the rectangle itself and do not have to be determined from the corner points whenever they are required. This definition permits an efficient redefinition of *length*; *scale* has to be redefined because the lengths of the sides are altered by scaling, and therefore the definition of scale cannot be taken directly from *polyline*; on the other hand, *translate* can be taken over directly from *polyline* because the additional state is unaltered by translation.

The overall attributes of class *rectangle* comprise the attributes of class *polyline* and the additional attributes *side1_length* and *side2_length* introduced in rectangle.

We still have to explain how the compiler can implement the dynamic binding efficiently. In this example, for efficiency reasons, the class *rectangle* incorporates the two lengths of the side of the rectangle in its objects. Consequently, the scale method of *polyline* cannot be used to scale a rectangle, since it knows nothing about the additional attributes which must also be changed by the scaling; the

method defined by *rectangle* must be used. Now, an object of the class *rectangle* may generally be passed as an argument of a function or method if an object of the super-class *polyline* is permissible there. In particular, this applies to the function zoom, which first translates a graphical object so that the point center comes to lie at the origin and then scales the result by the value *zoom_factor*.

```
void zoom (graphical_object &obj, double
zoom_factor, point &center) {
obj.translate (-center.x, -center.y);
/*move "center" to "(0,0)" */
obj.scale (zoom_factor);        /*scale*/
}
```

The function zoom: center and scale

If applied to a rectangle, the body of *zoom* must call the scaling function of rectangle, and not *polyline*, or even that of *graphical_object*; however, *zoom* may already be compiled and stored in a library before the class rectangle is even defined. Thus, when compiling *zoom*, the compiler may not even know the method to be activated in the body of *zoom*. Consequently, the compiler cannot bind *scale* to a concrete method. *scale* has to be dynamically bound, that is, bound at run time, to a method within *zoom*.
Compilers may use the following scheme to handle this dynamic binding efficiently. For each class, the compiler creates a method table, which includes all the methods defined in the class that may have to be dynamically bound. For evident reasons, these methods are called **virtual function tables in C++**. They contain entries for all methods of a class or its super-class that are defined to be **virtual**. Each object includes as its first component a pointer to the method table corresponding to its proper class. The compiler binds methods names to indices in the method table. To call a method it activates the function stored under the corresponding index in the method table. The method table of an inheriting class is generated as follows. Begin with a copy of the method table of the base class. In this copy, redefined methods are overwritten by the definition. Then the methods newly introduced are appended to the table. This ensures that the names of methods previously defined in the base class are assigned the same index in the new class.
If B is a class and A is a super-class of B then an A view of an object b of B consists of an initial section of b and an initial section of the method table referenced by b. The initial section of b belonging to A view comprises precisely the pointer to the method table and attributes inherited from A. The section of the method belonging to the A view covers the indices for the methods introduced in A, or a super-class. All views of b are represented in the same way by a pointer to b. Thus, the transition between different views is trivial.
Figure 2 shows the method tables for *graphical object*, *polyline* and *rectangle*. The method *table* for *polyline* is derived from that for *graphical object*; first the methods *translate_PL* and *scale_PL* which are redefined in *polyline* replace the corresponding methods of *graphical objects* and second, the newly defined (virtual) method *length_PL* is appended. In turn, the method table for rectangle is derived from that of *polyline*; the methods redefined in *rectangle* replace the corresponding methods of *polyline*. The method *translate_PL*, which is not defined, is passed on. The compiler binds *translate* to index 0, *scale* to index 1 and *length* to index 2.

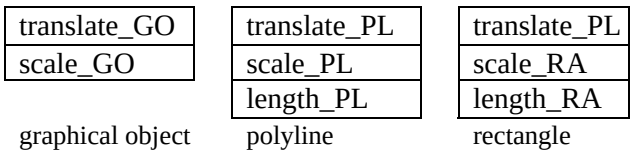

Figure 2
Methods tables for different subclasses of graphical objects

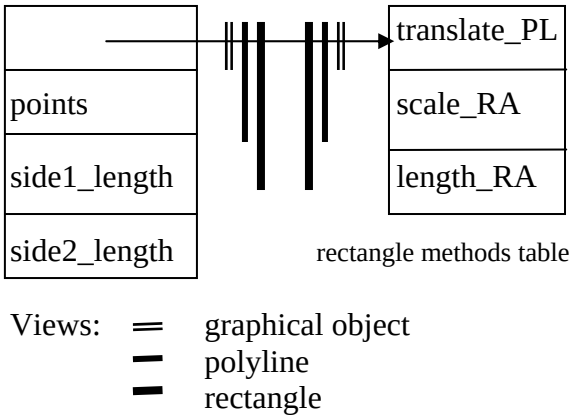

Figure 3
Representation of rectangle objects

Figure 3 shows the representation of rectangle objects. In addition to its own state, each such object contains a pointer to the method table of the class rectangle.

3 Conclusion

Thus, the implementation of simple inheritance with dynamic binding is associated with a storage overhead of one pointer per object. In addition, the method table associated with each class has to be stored. The dynamic binding leads to an increase in the run time for a method call, because of the dereferencing a pointer to find the method table and indexing to locate the method to be activated.

References:

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compliers Principles, Techniques and Tools*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[2] R. Wilhem, D. Maurer, *Compiler Design*, Addison-Wesley Publishing Company, 1995

[3] J. Engel, *Programming for the Java Virtual Machine*, Addison-Wesley Publishing Company, 1999

[4] J.M. Berge, *Object-Oriented Modeling*, Springer, 1996

[5] P.A. Fritzson, *Principles of Object-Oriented Modeling and Simulation*, Wiley-IEEE, 2004

[6]http://citeseer.ist.psu.edu/cardelli88semantics.html

[7]http://portal.acm.org/citation.cfm?id=28702

[8]http://www.oopweb.com/Compilers/Files/Compilers.html

[9]www.springerlink.com

[10]http://en.wikipedia.org/wiki/Inheritance(object-oriented_programming)