On the Reuse of Object Oriented Applications as Components

Sergio Eduardo Rodrigues de Carvalho
Departamento de Informática, Pontificia Universidade Católica RJ
Rua Marques de São Vicente 225, Gávea, Rio de Janeiro, 22453
Telephone: (55)(21)274-4449 Fax: (55)(21) 511-5645
e-mail: sergio@inf.puc-rio.br

ABSTRACT

Object orientation seems to provide a welcoming framework for software reuse. This software is typically in the form of individual classes, and the reuse is typically obtained through composition and inheritance. In this paper we address the reuse of stand-alone applications, without modifications, as components in other applications. We do this in the TOOL programming system, taking advantage of its handling of asynchronous messages at the class level.

KEYWORDS AND PHRASES

object orientation, asynchronous message, software reuse, GUI application, class, interface-implementation decoupling

On the Reuse of Object Oriented Applications as Components

INTRODUCTION

The ability to reuse a known piece of software in the construction of a new program has long been recognized as one of the main goals in the software engineering discipline [1, 2]. Object orientation (OO) seems to provide a welcoming framework for software reuse [3]. Encapsulation, genericity, abstraction and inheritance, OO features associated with *classes*, or the models for the real life entities programs must simulate, are mentioned as the keys to reuse in the OO world [4, 5, 6, 7].

A class is typically a well-delimited syntactic structure having the following properties:

- it contains all attributes concerning the entities modeled, including the data structure for each entity, and the operations that can be applied to them;
- it may be constructed, compiled, and tested separately, and stored in libraries for further use.

It has been pointed out that, although the reuse of a single class seems to present no great difficulties, the reuse of classes collected in hierarchies needs to be treated with care [8]. In this paper we also discuss the reuse of multiple classes, not necessarily related in a class hierarchy, but which form an application program. We consider the reuse of this program as a component in another, without having to modify its contents. In other words, we will discuss the construction of software pieces that can be used, without changes, both in the stand-alone and in the component modes.

We do this in the TOOL Programming System (TPS) [9], shortly described below, which has been created to simplify the development of applications involving graphical user interfaces (GUI's), and where the asynchronism implied by such interfaces is handled at a very high level.

We begin by presenting well-known mechanisms for OO reuse, namely composition and inheritance. Next we introduce the problems of reusing applications, able to execute by themselves, as components in other applications.

We proceed by introducing some concepts in the TPS that are relevant to the discussion at hand: the notion of application program, the structure of classes, the semantics of messages, the ownership relation among objects, and the importance of a special class method called Main. We also present the TPS way of constructing a stand-alone application, where the interface part (responsible for the screens the application user sees), and the implementation part (responsible for the real work that must be done to satisfy the user) are completely decoupled [10]. This decoupling is a key ingredient in the reuse to be discussed next.

We finally address the problem of reusing stand alone applications as components, carrying along, as an illustrative example, the realization of a calculator.

REUSE IN OBJECT ORIENTATION

Here we focus on classes as object models, and comment on composition and inheritance. Using the *composition* mechanism, the data structure for objects in a class being constructed can be expressed as a function of other classes already available, as for example in the partial declarations below:

```
CLASS Point;
        REPRESENTATION
                REAL x, y;
        END REPRESENTATION
        ...
END CLASS
```

```
CLASS Person;
        REPRESENTATION
                STRING name, address;
                DATE birth;
        END REPRESENTATION
        ...
END CLASS
```

which describe the structure of an object of class Point as being a pair of instances of the already available class REAL, and the structure of an object of class Person as a composition of two strings and a DATE.

This kind of reuse is certainly much older than object orientation; the composition mechanism for the construction of records, or structures, in languages like COBOL, Pascal and C is well-known, also establishing a *has-a* relationship between an instance of the structure being defined and each of its components. The object orientation contribution to this mechanism lies:

- in the encapsulation of this record-like declaration within the confines of a class, close to the declarations of all operations dealing with this internal structure;
- in the protection usually given to the internal components from any outside access, making them visible only to the class operations.

The *inheritance* mechanism also allows us to bring back from libraries classes already available, but for a different kind of reuse. In systems where strong typing is enforced, the declaration

```
CLASS Student FROM Person;
        ...
END CLASS
```

relates classes Student and Person in such a way that every instance of Student is also an instance of Person, and can be used wherever a Person is expected. From the reuse point of

view, inheritance can be likened to a sort of macro-expansion, where the contents of the existing class (Person) start out the definition of the new class.

With the composition and inheritance mechanisms available, one can construct an object oriented application, or a program P, to satisfy the needs of a user. Much less common in both the procedural and in the OO world is the ability to reuse this program P, without modifications, as a component in another application, one in which the functionality of P is considered useful [3]. The difficulty with this kind of black-box reuse seems to lie in the fact that P is typically constructed in the procedural way, and OO reuse is based on classes, or static object models.

There are both semantics and syntactic aspects to be considered when this cultural barrier is to be crossed. From the semantics point of view, it is obvious that the execution of the application must start somewhere, for example in a main program (or procedure) in some way recognizable to the runtime system. This is typically done, in the procedural world, by attaching a different syntax to this main program, thus allowing a compiler to create a special code for this special procedure.

In OO, however, we talk about the reuse of classes, which are entities completely different, in nature, from main programs. The TPS solution to this problem, to be discussed in detail below, consists basically in creating a "main program" that is also a class, and therefore able to go to storage and be reused, either as a component in another application or as a superclass in a new class declaration.

THE TOOL PROGRAMMING SYSTEM

The TOOL language is described in [9]. In this section we present the related notions of classes and application programs, and discuss the passing of messages among TOOL objects in an application.

The main programming unit in TOOL is the class. *Classes* are the meeting grounds for all properties associated with the objects they model. Among these properties we find a data

structure describing the composition of class objects, and a collection of operations describing the behavior of class objects.

Objects in TOOL can behave either procedurally, receiving method applications, or asynchronously, handling messages. *Methods* are active program units used in the implementation of synchronous operations on objects. They correspond to the notions of procedures and functions in conventional languages. They have a name, a possible list of parameters, a possible return object, local declarations and a body. When a method is applied to a receiving object, the execution of the operation where the application was issued is suspended until the method application terminates, when it is resumed from the statement following that application.

To discuss message passing in the TPS, it is important to understand the *ownership relation* that may be established between two TOOL objects. We do this with an example, partially declaring a class Interface that displays on the screen a window 'w' and two buttons, 'ok' and 'cancel'.

```
XCLASS Interface;
        REPRESENTATION
                Window      w;
                PushButton  ok, cancel;
        END REPRESENTATION
        METHOD  R;
        BEGIN
                w <- CREATE;
                ok <- CREATE;
                cancel <- CREATE;
                ...
        END METHOD
END XCLASS
```

In the example above, when the method R is applied to an object of the class Interface, the window 'w' and the two buttons 'ok' and 'cancel' are created. This states that this interface object is the owner of 'w', 'ok' and 'cancel'. This is important, since the owner of an object that is able to send a generic message is the one responsible for handling the message sent; in other words, its class must contain the declaration of a message handler that is executed when that message is received. If no such handler exists, the message is lost.

Again in the example above, when the user presses a button, say 'ok', the PushButton message 'Clicked' is sent. In the example, it is the responsibility of the class Interface to contain a handler for this message. Actually, since two buttons with obviously different purposes have been declared in the Interface class, we would need two handlers for the same 'Clicked' message, one for each button. With this new behavior attached, the class Interface would look like this:

```
XCLASS Interface;

    REPRESENTATION
        Window      w;
        PushButton  ok, cancel;
    END REPRESENTATION

    METHOD  R;
    BEGIN
        w <- CREATE;
        ok <- CREATE;
        cancel <- CREATE;
        ...
    END METHOD

    HANDLER FROM ok FOR Clicked;
        -- locals and actions
    END HANDLER
```

```
        HANDLER FROM cancel FOR Clicked;
            -- locals and actions
        END HANDLER
        ...
END XCLASS
```

We now describe the message passing features of the TPS, referring back, when convenient, to the example above. To encompass message passing, classes in TOOL contain a message section and the declaration of message handlers. In the *message section*, the signatures of all messages that can be sent from class objects to their owners (the objects that created them) are given. Message signatures have a name and an optional set of parameters. In the Interface class above no message section was declared, indicating the fact that objects of this class do not need to send messages to their owners.

Handler declarations contain a signature, local declarations and statements. Handler signatures contain a message identification clause and an optional sender identification clause. The *message identification*, always required, identifies a message that objects of the class being declared can respond to, by creating the corresponding handler environment and by executing its statements. In the example above, the clause "FOR Clicked" identifies, in both handlers, a message Interface objects may respond to.

The *sender identification*, when present, indicates one or more owned objects that are possible senders of the message to be handled, and which will cause this same message handling. All objects in this clause must be modeled by the same class. Handlers without a sender identification clause treat outside messages to objects of this class. In the example above, the senders are the owned buttons 'ok' and 'cancel'.

We see that the message section and the declaration of handlers split the responsibility for the asynchronous behavior specified in a class A: the former tells the system about the messages that objects of class A can send to their owners; the handlers of class A tell the system in what way objects of this class treat the messages they receive.

Handler applications are syntactically similar to method applications. The semantics differ, however: handlers do not return to the calling environment, they just eventually quit executing; handler parameters are input only, and handlers do not return values. Besides, the operation containing the message sending statement can continue with its execution, as opposed to method application statements, which suspend the execution of the sending operation until the method terminates.

In order to generalize the passing of a message from an object to its owner, a generic owner identifier exists in the language. In this way the message sending behavior of objects can be specified independently of the context in which they are placed, truly favoring encapsulation. The calculator example presented below illustrates this mechanism.

We conclude this section by describing the organization of an application program in the TPS. A TOOL program is a collection of one or more classes, which are submitted to the compilation process. The classes in the collection typically use other classes already stored in libraries. After compilation, the linking of all relevant classes must take place. At this time, the TPS program construction interface asks the application constructor to name some class in the collection as the main class, one which is going to head off the construction process. This main class must contain a method called Main. The execution of the now constructed program starts with the application of Main to an object of the main class, automatically created by the system.

Actually one or more classes in the collection defining a TOOL program may have methods called Main. Due to this a TOOL program may have a set of entry points. Once one of these classes is identified as the main class, however, the program construction is set and execution may start.

We see that organizing a program as a collection of classes and having the TPS create an object of the designated main class, which automatically receives the application of a special method, removes the procedural aspect usually seen in an object oriented program. In fact the TPS requires the application constructor to designate a library where the code

generated for the application is going to be stored; this library, once created, behaves as any other library in the system, and can, for example, be reused in another construction.

However, this program organization in itself is not enough. To fully realize the potentialities of reuse, it is also necessary that an application constructed to function in the stand-alone mode be useful in new contexts, being able to communicate with new applications [3, 11, 12]. In the next section we begin to show the TPS solution to the development of black-box components by discussing the construction of TOOL applications, where the separation between interface and implementation is maximized to facilitate reuse.

DECOUPLING INTERFACE AND IMPLEMENTATION

This is a common goal in the design of object oriented applications having GUI's [13, 14]. The more mutually independent interface and implementation are, the higher the possibilities of reuse of any of these parts. In the TPS the message passing facilities described above are used to separate interface and implementation [10]. To show how this separation is achieved, let us consider the construction of a simple, stack-based calculator, reasoning first about its interface. Let us keep in mind that in what follows we will be constructing a stand-alone calculator, unable to be reused as component in another application. The reuse features will be added later.

A calculator interface typically contains:

- buttons representing the ten digits and the decimal point;
- buttons representing the operations, for example ENTER, +, /, etc.;
- an area for the display of values;
- a window to contain all of the above, to be shown when the calculator is needed.

We can declare a class, CalcInterface, having in its REPRESENTATION section declarations for all the visual objects above. In this class we place a method, say SetUp, which when applied to an object of the class CalcInterface displays the window and the visual objects contained therein on the screen. This class would now look like this:

```
XCLASS CalcInterface;
    REPRESENTATION
        -- declarations for the buttons, display area, window.
    END REPRESENTATION

    METHOD SetUp;
    BEGIN
        -- create buttons, display area, window, thus establishing ownership;
        -- initialize all visual objects with style, screen positions, etc.;
        -- show the calculator interface.
    END METHOD
END XCLASS
```

Since we are talking about the interface of an application, asynchronism will occur: the calculator user will press buttons, constructing values and operating upon them to get results. We now discuss how the interface relates, through messages, with the rest of the application. Consider first the digit buttons: when each is pressed, the resulting action should be the display of a newly composed string of digits in the display area. Since the CalcInterface class owns all buttons, we declare in this class handlers from all digit buttons for the message Clicked; the action defined within each handler is the display area composition mentioned above. These handlers would look like this:

```
    HANDLER FROM one FOR Clicked;
    BEGIN
        -- attach the digit '1' to the display area;
    END HANDLER
```

(Actually, a single handler could manage all digits; let us however suppose ten digit handlers for the purposes of this paper.) With this new behavior given to CalcInterface objects, when the calculator user presses a digit button, the interface itself manages to construct a new string on the display.

The nature of the operation buttons is somewhat different. When the ENTER button is pressed, for example, a real value obtained from the display area must be stacked up; when the '+' button is pressed, the converted display value must be added to the value on the top of the stack, and the result of this operation must be shown on the display area. We decide, when designing the calculator, that the keeping of a stack and the execution of operations are not the responsibility of the interface, but rather of another class, called CalcImplementation, where internal data structures and operations are declared. Before discussing this class, however, let us complete the message protocol for the CalcInterface class.

Again, since this class owns all operation buttons as well, handlers from all these buttons for the message Clicked must be declared in the interface. The difference between these handlers and the digit handlers introduced above is that the interface is not able, by itself, to take care completely of the actions required by the user; the interface knows nothing about a stack, for instance. So the actions defined within the operation handlers declared in the interface consist in sending messages to the interface owner, passing on the operation chosen and the current string on the display area. This is in fact what the interface must do, since it alone owns the buttons; this is also all it can do, since it knows nothing else.

A handler for a calculator operation would be declared thus:

```
HANDLER FROM add FOR Clicked;
BEGIN
        OWNER <<- OperationIs ('+', dispvalue);
END HANDLER
```

In this code we see a message being sent to the (unknown) owner of the CalcInterface object. As a consequence, as mentioned above, a message section must be added to the CalcInterface class:

```
MESSAGE
        OperationIs (CHAR, STRING);
        -- other messages, if needed.
END MESSAGE
```

The design of the calculator interface is more or less complete. It owns all visual objects the user may act upon, and must therefore be responsible for handling the messages sent when the user acts on those visual objects. Some actions are simple enough for the interface itself to take care; others the interface cannot handle, passing them on to its owner, through the OperationIs message.

It is time to discuss this owner. This is another class in the collection composing the calculator, one that receives messages from the interface and the implementation and handles these messages, managing in fact all communications in the system. Its organization is summarized below:

```
XCLASS Calculator;

    REPRESENTATION -- structure of a calculator:
        CalcInterface  f;
        CalcImplementation  m;
    END REPRESENTATION

    METHOD Main;  -- execution starts here, in the stand-alone mode:
    BEGIN
        f <- CREATE;    -- to establish ownership;
        f <- SetUp;     -- to display the calculator on the screen;
        m <- CREATE;    -- to establish ownership;
        m <- SetUp;     -- to set up internal data structures, as the stack;
    END METHOD

    -- message handlers treating messages received from the owned objects f and
    -- m; for example, the message OperationIs received from f:
```

```
        HANDLER FROM f FOR OperationIs ( CHAR op, STRING disp );
            REAL realdisp;
        BEGIN
            -- convert disp to a real value and invoke the implementation;
            m <- Compute ( op, realdisp );
        END HANDLER

        -- and the message ResultIs received from m:
        HANDLER FROM m FOR  ResultIs ( REAL value );
            STRING strval;
        BEGIN
            -- convert value to a string;
            f <- Display ( strval );
        END HANDLER
    END XCLASS
```

Due to the last handler we see the need for a method Display in the class CalcInterface.

The last class in the stand-alone calculator system would be CalcImplementation. Its details, however, are not needed in this paper and will be omitted; we just mention in passing the need for the declarations of a message ResultIs and a method Compute.

Summarizing the contents of this section so far, we have defined a calculator as a collection of three classes: the calculator itself, which manages all domain change information passed in the system; the interface, in contact with the user and handling messages sent when visual objects are activated by the user; and the implementation, where computations take place. The presence of the calculator manager decouples interface and implementation, both of which do not even know who they are working for, simply sending messages to a generic owner (OWNER) when out of their depths.

The main class of the system is the Calculator class, or the information manager, which possesses a method Main, as seen above. We go back to the declaration of this method,

adding important information: the position of the calculator window on the screen, and the window in which the calculator is to appear. For the stand-alone calculator, the method Main now looks like this:

```
METHOD Main;
        Window  w;
BEGIN
        f <- CREATE;
        f <- SetUp ( 10, 10, w );
        m <- CREATE;
        m <- SetUp;
END METHOD
```

The window 'w', local to Main, is the containing window for the calculator. The upper left corner of the calculator window itself is placed ten pixels to the right and ten pixels down from the upper left corner of the containing window. The method SetUp declared in the interface must be redeclared to account for the new parameters.

This completes the definition of a stand-alone calculator. When compiled and constructed, a library is set up to contain the resulting code. When executed, a calculator interface shows up and computations can take place. When the calculator is not needed anymore, its window can be closed, the interface disappears, memory is released and another application can take over the system. But keep in mind that the calculator can be run again, in the stand-alone mode.

APPLICATIONS AS COMPONENTS

Let us now suppose that, in the design of another application, the need to compute values during the execution of this new application arises. Furthermore, these values are to be incorporated to this new application; they are important in reaching a larger goal. We may for example have to compute the value of a spreadsheet cell, or calculate on the fly a new salary for an employee. We already have a calculator; what we need is a way to incorporate this calculator in any other applications where it might be useful.

We emphasize that our purpose is not to tailor the calculator each time its need arises; rather, we want to have calculator classes that allow it to be used anywhere, independent of the application where placed, and without further changes.

We note that all mechanisms needed to accomplish this goal in the TPS have already been presented above. In fact what we need in order to have a calculator either as a component or as an application by itself is:

- the addition, to the interface, of a button SEND, which, when pressed by the user, sends to the calculator owner the current display value;
- the addition, also to the interface, of a method HideSend, applied to the interface object owned by the manager when the calculator is in the stand-alone mode;
- the addition, to the manager, of a handler for the new interface message ValueSent, which sends to the interface owner the display value;
- the addition, also to the manager, of a message section, where the message CalcResultIs sent from the calculator to its owner is declared;
- the addition, also to the manager, of a method Initialize, to be used instead of Main to set up the interface and the implementation, when the calculator is a component in another application.

The new calculator classes would look like this:

```
XCLASS Calculator;

        REPRESENTATION  -- as before
                -- as before
        END REPRESENTATION

        MESSAGE  -- to be sent to calculator owners;
                CalcResultIs ( REAL );
        END MESSAGE
```

```
METHOD Main; --  to be used in the stand-alone mode;
        Window  w;
BEGIN
        f <- CREATE;
        f <- SetUp ( 10, 10, w );
        f <- HideSend; -- to hide the SEND interface button;
        m <- CREATE;
        m <- SetUp;
END METHOD

METHOD Initialize ( IN INTEGER x, IN INTEGER y, ID Window parent );
-- to be applied to a calculator object when a component in another
-- application;
BEGIN
        f <- CREATE;
        f <- SetUp ( x, y, parent );
        m <- CREATE;
        m <- SetUp;
END METHOD

HANDLER FROM f FOR ValueSent ( STRING val );
-- to pass on the result of the computation;
BEGIN
        OWNER <<- CalcResultIs ( val <- REAL );
END HANDLER

-- other handlers from f and m, as before;

END XCLASS -- Calculator.
```

```
XCLASS CalcInterface;

    REPRESENTATION
        -- as before, plus new button SEND;
    END REPRESENTATION

    MESSAGE
        -- as before, plus:
        ValueSent ( STRING );
    END MESSAGE

    HANDLER FROM send FOR Clicked; -- user pressed the button SEND;
    BEGIN
        OWNER <<- ValueSent ( disp );
        -- terminate the calculator, since value already sent;
    END HANDLER

    METHOD HideSend;  -- to hide the button SEND, when stand-alone;
    BEGIN
        send <- Hide;
    END METHOD

    -- other handlers and methods, as before;

END XCLASS

XCLASS CalcImplementation;
    -- as before;
END XCLASS
```

With the organization above we have given our calculator the ability to function in both worlds. If we use the calculator by itself, the button SEND is hidden; the class

Calculator has to be named as the main class in the construction process, since this is where the method Main resides. As a component in another application, the calculator could be used like this:

```
XCLASS AnotherApplication;

    REPRESENTATION
        -- whatever is needed, including a containing window for the
        -- calculator (say parent), plus:
        Calculator calc;
    END REPRESENTATION

    METHOD Main;
        -- locals, if needed, including possibly the coordinates x, y
        -- of the calculator with respect to its containing window;
    BEGIN
        -- typically create and initialize all components, including:
        calc <- CREATE;
        ...
        IF calculator is needed THEN
            calc <- Initialize ( x, y, parent ); -- to show the calculator;
            ...
        END IF
        ...
    END METHOD

    HANDLER FROM calc FOR CalcResultIs ( REAL val );
        -- what to do with the the value computed by the user;
    END HANDLER

END XCLASS -- AnotherApplication.
```

CONCLUSIONS

The construction of applications that can, without adaptations, either stand alone or be used as components in other applications is a worthy goal [3], eliminating the adapting code phase mentioned in [15,16] and satisfying the software representation requirements mentioned in [12]. In this paper we have shown, through an example, how this goal can be reached in the TPS, which allows the creation of OO software for GUI applications where the separation of the interface part from the implementation part is maximized: both work for a generic owner, and do not communicate directly.

There are three modern features in the TPS that allow the construction of reusable, dual-role, and fully encapsulated applications:

- the ability to represent, in user-declared classes, message passing and receiving object behavior;
- the ownership relation established between objects;
- the generic designator OWNER, which allows an owned object to behave as a pure server, communicating results to an unknown client.

We can summarize the design of such dual role applications by observing the principles below:

- Define the dual role application as a collection of at least three classes: the information manager, the interface and the implementation.

With respect to the manager:

- The manager contains and owns an interface object and an implementation object, and consequently handles all messages sent from these two components.
- Interface messages are of two possible natures: they indirectly require some action from the implementation part, or they pass to the manager information to be in turn passed to the manager's owner, when the dual role application is a component.
- The manager has at least two important methods: Main, to be used in the stand-alone mode, and Initialize, to be used when the dual role application is a component.

- The methods Main and Initialize create and set up both the interface and the implementation; the method Main, in addition, hides all interface objects used when the dual role application is a component.

With respect to the interface:

- The interface contains and owns all visual objects necessary to the user of the dual role application. Some of these objects are to be used only when the application is a component.
- The interface class handles all messages received from these visual objects. Some of these messages are handled entirely within the interface, being simple in nature and having to do with the interface itself. The other messages are sent to the unknown owner of the interface.
- The interface contains a method that hides all objects needed only when the dual role application is being used as a component.

ACKNOWLEDGEMENTS

The TOOL Programming System is the result of a joint research and development project conducted by SPA (Sistemas, Planejamento e Análise) and a group from the Departamento de Informática of the Pontificia Universidade Católica, both in Rio de Janeiro. This report must be understood as another result from this same team effort, although the author takes sole responsibility for its contents. Thanks are due to Otavio P. Coelho, Nelson Gorini, and Werther J. Vervloet, early day team members, and to Mario S. Cardoso, Eduardo Galucio and Toacy C. Oliveira. The author also acknowledges the numerous improvements to this paper suggested by Julio C. Leite.

BIBLIOGRAPHY

1. M. D. McIlroy, "Mass-produced Software Components", in *Software Engineering Concepts and Techniques*, eds. Buxton, J., Naur, P., Randell, B., 88-98, 1976.

2. T. Biggerstaff and A. J. Perlis, (eds), "Special Issue on Software Engineering", *IEEE Trans. Soft. Eng.*, SE-10 (5): 474-609, Sept. 1984.

3. R. Prieto-Diaz, "Status Report: Software Reusability", *IEEE Software*, 10 (3):61-66, May 1993.

4. J. Lewis, S. Henry, D. Kafura and R. Schulman, "An Empirical Study of the Object Orientation Paradigm and Software Reuse", *Proceedings OOPSLA 1991*, Nov 1991.

5. B. Meyer, "Reusability: the case for Object-Oriented Design", *IEEE Software*, 4(2):50-64, Feb 1987.

6. D. Taenzer, M. Ganti and S. Podar, "Object-Oriented Software Reuse", *JOOP*, 2 (3): 30-35, Sept/Oct 1989.

7. B. Meyer, "Lessons from the Design of the EIFFEL Libraries", *CACM* 33(9):68-88, Sept 1990.

8. P. Johnson and C. Rees, "Reusability Through Fine-grain Inheritance", *Software Practice & Experience*, 22 (12):1049-1068, Dec 1992.

9. S. E. R. Carvalho, "The Object and Event Oriented Language TOOL", Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 6/93.

10. S. E. R. Carvalho, "Decoupling Interface and Implementation: the TOOL Solution", Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 37/92.

11. R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability", *IEEE Software*, 4 (1): 6-16, Jan 1987.

12. T. Biggerstaff and C. Richter, "Reusability Framework, Assesment and Directions", *IEEE Software*, 4 (2): 41-49, March 1987.

13. Cowan, D., Ierusalimschy, R., Lucena, C., Stepien, T., "Abstract Data Views", *Structured Programming*, 14(1):1-13, Jan 1993.

14. Lucena, C., Cowan, D. Ierusalimschy, R., Stepien, T., "Application Integration: Constructing Composite Applications from Interactive Components", *Software Practice and Experience*, 23(3):255-276, March 1993.

15. K. Huff and R. Thomson, "Supporting Understanding and Adaptation in Software Reuse", *Proceedings of the First International Workshop on Software Reusability*, 45-50, Dortmund, Germany, July 1991.

16. V. Basili and H. Rombach, "Support for Comprehensive Reuse", *Proceedings of the First International Workshop on Software Reusability*, Dortmund, Germany, July 1991.