Asynchronous Behavior in the TOOL Programming System

Sergio Eduardo Rodrigues de Carvalho

Departamento de Informática, Pontifícia 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

It is expected that a large portion of the software systems to be constructed in this decade will involve object orientation, message passing and graphical user interfaces. In this paper we present an object and event driven programming system, TOOL, which takes full advantage of object orientation, by considering the handling of asynchronous messages as part of object behavior. In this way low-level and error prone concerns with message queues and centralized handling routines are eliminated. Moreover, we can achieve full decoupling between interface and implementation in a GUI application, and reuse stand-alone applications as components in other applications.

KEYWORDS AND PHRASES

object orientation, asynchronous behavior, programming language, GUI application, interface-implementation decoupling, reuse.

INTRODUCTION

Object orientation is being considered at least as a viable alternative in the construction of good quality, reusable software [1, 2, 3, 4, 5]. Several features contribute to this belief: encapsulation, genericity, abstraction, and inheritance are just a few. Most of these features are also present in conventional programming systems, but they seem to be combined best in an object orientation framework.

Encapsulation is of particular interest in this paper: this feature allows the definition of syntactic structures, usually called *classes*, which concentrate all attributes affecting their instances, or *objects*. Among these attributes we find a data structure definition that models the state of each class object during its life cycle, and a collection of operations that model the behavior of such objects.

One of the advantages of encapsulation is the protection that is usually provided as a side effect: the data structure fields of each object are visible only to the operations in its modeling class. This saves the user of an object from concerns about its internal details. As a consequence object users deal at a higher level of abstraction, and this tends to improve software quality. Another advantage of encapsulation is that once a class is defined, implemented and tested, in short considered ready to be used in applications, it is made available in some library.

The role of such library classes, when used in some application, is usually that of models for server objects: their services, or operations on objects, are needed in the context

of some client object. We can illustrate this by considering a class Button in a library of visual objects for the construction of graphical user interfaces. The services provided by this class usually include operations that show, hide, enable and disable, initialize and move buttons, for example. The client of a button 'b', for example an object of a class modeling some user interface, may request that the button be hidden, since at a particular moment during execution its presence on the screen is undesirable. For this to occur, the client must know the server, or, in other words, the identity of the button being considered must be available to the interface. This is accomplished by declaring the button in the data structure of the class modeling the interface; once this identity is known, the interface may request the button 'b' to Hide.

The communication between client and server is usually implemented through input and output parameters declared in the signature of the server operation. The result of the operation may also be revealed through side effects that alter the overall state of the application. The semantics of operation application is usually that of procedure calls in conventional languages: a high degree of synchronism exists between the callee and the called operations. In more detail, the referencing environment of the called operation is saved on top of the execution stack; control data such as return addresses, static and dynamic links are also saved on the stack; and finally control is transferred to the code generated for the called operation. While this operation is executing, the callee is suspended, and is active again upon termination of the called operation.

As mentioned above, this is basically the semantics of procedure calls in strongly typed conventional languages. In object orientation, however, the binding of the operation called to the receiving object may only be possible at execution time. In this case the cost of applying an operation to a receiving object increases, since for example a table of all possible operations that can be applied to the object must be kept at execution time, and each object must be accompanied by a descriptor that informs the runtime system the object's current class.

Applications involving graphical user interfaces, however, must also deal with a different operational semantics, since users may asynchronously interfere with the execution by selecting a menu option or by pressing a button, for example. Going back to the Button library class used above, we can say that buttons provide another kind of service beyond those provided to application programs: they may be pressed by interface users that expect some reward for their efforts. It is as if the client is now the user, an not the application program. However, it is the latter that must react to the action provoked by the user.

The semantics of procedure calls described above is not adequate to this situation: the user of the application program may go on activating interface controls, totally unconcerned about the termination of the first actions requested. For this reason modern platforms for the development of graphical user interface applications, such as Microsoft Windows [6] and X-Window [7], contain primitives that implement the semantics of asynchronous message passing, usually involving a message queue with accessing operations. The use of these primitives in the construction of an application is not a simple task, however; one must

typically dominate some conventional programming language, plus a large and unstructured set of new functions and data models in order to code message passing.

Our contention in this paper is that the construction of object oriented software can benefit from the inclusion, in new object oriented languages, of class operations that handle non-procedural semantics, as for example that of asynchronous messages. We propose, furthermore, that the choice of behavior rest with the class constructor, who must decide whether modeled objects react synchronously or asynchronously to service requests. To exemplify this need for full extensibility, consider a payroll application where reports must be generated. Since reports do not as a rule contribute to the continuation of the application, returning no values, they may be produced asynchronously. To achieve this effect one may devise a ReportWriter class with message receiving operations, not methods, to have reports printed out.

In the next section we shortly describe the TOOL programming language, where the semantics of messages is implemented at the level mentioned above. We then discuss in detail the message passing and receiving facilities available in the language, and show how they can be used first to decouple interface and implementation in a graphical user application, and then to allow the reuse of stand-alone applications as components in other applications.

THE TOOL PROGRAMMING SYSTEM

TOOL is a new object and event oriented programming system, of general purpose and fully extensible, consisting basically of a programming language, an optimizing compiler, a library system, an abstract machine to allow portability through different environments, a friendly environment for the construction of programs, and a visual application constructor. The current system is implemented as a software layer above Microsoft Windows, version 3.1.

Our main goal, when designing TOOL, was to simplify the task of building application programs that use the extensive facilities provided by Windows and similar environments. To accomplish this, considerable effort went into the design of programming language constructs that, besides implementing modern software features such as object orientation, message passing, and graphical user interfaces, seem also familiar to conventional language programmers. Thus, TOOL classes are presented as type extensions; the handling of asynchronous messages, essential to graphical interfaces, is compared to the familiar procedure call, and considered as just another kind of object behavior; polymorphism as an ability some objects have of changing classes at execution time.

We now present a short introduction to the TOOL programming language, shortly described in [8].

CLASSES

The main programming unit in TOOL is the *class*. A class models objects and other classes, being exclusively a compile-time entity. Classes can be *basic*, *conventional,* or

extended. A collection of basic classes is supplied with the system. They include, for example, implementations for INTEGER, REAL, CHARACTER, BOOLEAN, STRING. Conventional classes are not significantly different from classes found in conventional object oriented languages. Extended classes model message passing and receiving objects, and are the main concern of this paper.

Conventional classes may contain an inheritance clause, an object representation section, a class representation section, class constants and methods. We now describe each class section.

Inheritance in TOOL is simple, establishing both a type-subtype relationship and also visibility regions. Thus, if B is a subclass of A, every B object is also an A object and may appear wherever an A object is expected; and every A attribute is visible in B. A class and its descendants constitute a *family*.

In the *object representation section* the data structure for class objects is declared. This is basically a record consisting of object and structure declarations. Unless declared as public, representation components are only visible inside their family. For every component an initial value is given: either explicitly, or transitively obtained by default, since every basic class in TOOL has an initial value for its objects. The ordered collection of such initial component values is a constant value of the class (see below).

In the *class representation section* the data structure for the class itself is declared. Class representation components, objects and/or structures, are only visible inside their family and can be used as common storage among objects of the class, being visible only to operations in the family.

Special and immutable values for class objects may be declared in the *class constants* section. A constant object declaration associates a name with a multiple value. Such values may be assigned to and compared with variable objects of the class.

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.

OBJECTS

Objects are instances of basic, conventional or extended classes. Objects can be declared in the object and class representation sections of a class declaration, as parameters to methods or message handlers (to be discussed below), as method results, and as locals to methods or message handlers.

An object declaration, always required, associates an object with a *base class*, which then models this object. A declaration also attaches a *discipline* to objects. This discipline affects both the lifetime and the class flexibility of objects. The possible disciplines are *automatic*, *dynamic* and *polymorphic*.

The *automatic*, or default, discipline is that of languages like Pascal: space for the object's representation, as defined in the object's base class, is allocated on the execution stack. The object's lifetime is that of the enclosed method/handler, or, for object components, that of the enclosing object.

Objects with the *dynamic* discipline have their lifetimes controlled by the programmer, through the application of built-in methods for creation and disposal. In addition, a set of special methods to safely control dynamic objects exists in the language, avoiding dangling references and simplifying garbage collection. In all other respects dynamic objects behave exactly like automatic objects.

The lifetime of *polymorphic* objects is also programmer controlled. Unlike automatic and dynamic objects, however, they have the ability of changing classes during execution. The set of possible new classes for such objects is the object's family. Polymorphism results from the application of a special, built-in method to a polymorphic object. The method's argument must be an already created dynamic or polymorphic object of the receiving object's family. After the method's application, the receiving object shares the representation of the argument, and is of the same class as the argument.

METHODS

Methods have a signature, local declarations and statements. The signature of a method, besides the method's name, optionally contains the specification of parameters, of a

result object, and of a method attribute. Attributes exist for in-line code expansion, for private methods, and for methods deferred to its subclasses.

Parameters and *arguments* correspond according to position, family, mode and discipline. Parameters can be objects or merely argument identifiers. Object parameters correspond with their arguments through the assignment of a value (input, output or both), or through a reference. Subtyping is enforced: a subclass object value or reference may be passed to a superclass object. Object parameters can have any suitable discipline. Argument identifiers are exactly that: a local reference to an argument, which exists in some other context.

The *result object* is locally expressed with the declaration of an object, which can receive values during the method's execution. The last such value is the result of the method's application. Any suitable discipline can be attached to the result object.

Methods can be redeclared in subclasses. In this case the same signature is required.

Methods are applied to *receiving objects*, specially designated in method body statements. Receiving object components are accessed with dot notation from this special designator. The components of other local objects of the same class are also directly accessed with dot notation from their names. Components of other class objects, unless declared public in the declarations of those classes, are not visible inside the method.

A set of built-in methods is available in any non-basic class: for object creation and disposal, and for polymorphic transformation, for example. Certain built-in methods can only be applied to objects satisfying a certain declared discipline. Examples include creation/disposal methods, applicable to dynamic or polymorphic objects only, and the polymorphic transformation method.

OTHER FEATURES

A small and conventional looking set of statements is available in TOOL: assignment, selection, repetition, application (of methods and message handlers), and termination (of loops, methods and message handlers).

Object structures can be constructed with the *array*, *dynamic array* and *union* mechanisms. Constructors are merely compositions of objects; they are not classes.

ASYNCHRONOUS BEHAVIOR

Extended classes model objects capable of responding to asynchronous calls. This modeling is done at a very high level, taking full advantage of the class mechanism: TOOL programmers are not concerned with message queues and centralized message handling routines. Instead, we consider the reception of an asynchronous message by an object of a class as part of the behavior of objects of that class, to be defined in the class by routines with the same status, but different semantics, as that of methods. In fact, instead of saying that "a message M is sent to an object 'x'", as is common in object orientation, we state that "the handler for the message M is applied to 'x'", as we say that "the method R is applied to 'x'".

The basic message passing organization in TOOL has to do with the declaration of a special kind of operation, a *message handler*, in an extended class. The declaration of a handler for a message M, in a class modeling an extended object 'p', states that 'p' is able to receive the message M; this is part of the declared behavior for 'p'. The reception is done with the creation of a local environment for the handler and with the execution of the handler's statements. This execution is asynchronous, moreover: the operation (method or handler) that was executing when the message M was applied to 'p' continues after the message is sent, never expecting any results from the execution of the corresponding handler.

To illustrate, consider a class P containing a handler for a parameterless message M:

```
XCLASS P;
        ...
        HANDLER FOR M;
                -- local declarations, the handler's environment.
        BEGIN
                -- statements, the handler's body.
        END HANDLER
        ...
END XCLASS
```

If we also suppose the existence of an object 'p' modeled by the extended class P, and the message application statement

p <<- M;

then the handler for M declared in P would eventually be executed on the receiving object 'p', thus completing the message sending/receiving cycle.

The cycle explained above, which could be called the named message cycle, involves the sending of messages to known objects, or objects whose names are known in the message sending context. An important feature of the full message passing mechanism in TOOL could, in contrast, be called the generic message cycle, this one involving a sending object and its owner. In order to explain the correspondence between an object sending a message and its owner, the *ownership relation* must be presented. We do this with an example.

```
XCLASS Interface;
        REPRESENTATION
                Window      w;
                PushButton  ok, cancel;
        END REPRESENTATION
        METHOD  R;
        BEGIN
                w <- CREATE;    ok <- CREATE;    cancel <- CREATE;
                w <- Show;    ok <- Show;    cancel <- Show;
                ...
        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, then the message is lost.

Again in the example above, when the user presses a button, say 'ok', the PushButton class 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 extended 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
                w <- Show;     ok <- Show;      cancel <- Show;
                ...
        END METHOD

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

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

With this organization, the Interface class can ultimately serve the user by reacting to user actions on the screen. From the example we can see that:

- the PushButton class is brought in from a library to serve the Interface class. Its objects 'ok' and 'cancel' are created and shown, being the recipients of methods declared in the PushButton class.

- however, one can argue that the Interface class in turn serves the pushbuttons, reacting to messages they send, by executing the corresponding handlers.

The PushButton example above is representative of all classes modeling visual and control objects. They serve the application by behaving according to method application requests; they in turn send to the application messages responding to user actions, messages that the application must handle.

In order to properly encapsulate and make such classes reusable as black-boxes [1], the recipients of the messages sent from objects of these classes must be generic. When a button is pressed by the user, the message Clicked is sent, and it is the responsability of the execution environment to figure out who its recipient must be. As seen above, in the TOOL system this is solved by the ownership relation, established at execution time.

As another message passing example, this one using the ownership relation, and showing the extensibility of the message passing mechanism in TOOL, consider the need for buttons that know their positions; buttons used, for example, to represent chess or checker board cells. The extended class below simulates this situation.

```
XCLASS GameButton;

        REPRESENTATION
                SHORT  pos;  -- of button in board.
                PushButton   b;
        END REPRESENTATION

        MESSAGE   -- sent to owners
                GBClicked ( SHORT );
        END MESSAGE

        METHOD GBInitialize (...);
                b <- CREATE:  -- establishing ownership.
                b <- Show;
                ...
        END METHOD

        HANDLER FROM b FOR Clicked;
                OWNER <<- GBClicked ( SELF.pos );
                b <-  SetText ( "played" );
        END HANDLER

END XCLASS
```

In this model, the following behavior is specified:

- When a gamebutton is initialized in the application, its pushbutton component 'b' is created and shown, as seen in the method GBInitialize. This establishes ownership, so it is in the GameButton class that messages sent from the pushbutton 'b' are to be handled.
- When the user of the application presses a pushbutton, a message Clicked, declared in the built-in PushButton class, is sent to the gamebutton that owns it.
- When this message is received, the handler for this message is activated, resulting in turn in the sending of the user-declared message GBClicked, with the button's position, to the owner of the gamebutton, and in changing the button's text to "played".

In this last example, a message section was added to the GameButton class. In this section all messages sent from objects of this class to their owners are declared.

It is now possible to see the generic message cycle in TOOL. In extended classes, one can declare not only the behavior of objects when a message is received, but also the sending of messages to generic owners, other objects established when some application is being run. We note that the message sending behavior of extended objects is thus independent of the application where they are placed; they merely send a message to the current owner, responsible for its handling. This promotes encapsulation and class reutilization; gamebuttons, for example, could be reused wherever the need for their behavior is felt, provided that in their new context, the classes modeling their owners possess handlers for the messages they send.

We now summarize the features of extended classes in TOOL. To model asynchronous behavior, extended classes contain, besides the sections existing in conventional classes, a *message* section and the declaration of *message handlers*. The message section contains the signatures of all messages sent from extended class objects to their owners, or the objects that created them. Message signatures have a name and an optional set of parameters.

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 extended class being declared can respond to, by creating the corresponding handler environment and by executing its statements.

The sender identification, when present, indicates a set of 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 extended class. Handlers without a sender identification clause treat messages originating from non-owned objects.

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 behavior of objects can be specified independently of the context in which they are placed, truly favoring encapsulation.

We see that the message section and the declaration of handlers split the responsibility for the asynchronous behavior in TOOL: in an extended class, the former tells the system

about the messages that objects of this class can send to their owners; the handlers of this class tell the system in what way objects of this class treat messages they in turn 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 handler application statement can continue with its execution, as opposed to method application statements, which suspend the execution of the sending operation until the method terminates.

The implementation of this asynchronous behavior, in TOOL, takes full advantage of the primitives offered by the Windows operating system. For a handler application statement, the TOOL compiler generates code that, when executed, sends the corresponding message to the queue managed by Windows. When this message is removed from the queue, by another Windows operation, it is sent back to the TOOL level, where its owner is identified and the corresponding handler executed.

DECOUPLING AND REUSE

In this section we illustrate the power of the asynchronous object behavior in TOOL by addressing two important concerns in the development of object oriented software: the decoupling between application interfaces and application implementations, and the reuse of stand-alone applications as components in other applications.

We begin with the description of a TOOL program. A TOOL program is a collection of one or more conventional and/or extended classes. Some extended class in the collection must be identified as the main class (this is done through the program construction interface, a Windows application supporting the construction, compilation, linking and execution of TOOL programs). The main class must contain a method called Main. Execution starts with the application of Main to an extended object of the main class, automatically created by the system.

Actually one or more extended 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 cannot change.

To illustrate the decoupling and the reuse mentioned above, we use a calculator as an example. A calculator interface usually contains buttons for the ten digits and for the operations, and a display area for the value being constructed by the user. A corresponding implementation may contain a stack and its accessing methods, and other methods to execute the operations requested by the user.

In order to completely decouple the interface from the implementation, we can construct a TOOL calculator program as a collection of three classes: a calculator manager, a calculator interface and a calculator implementation. The main class in the collection is the manager, which declares and owns an object of the interface class and an object of the

implementation class. The manager's function is to control all communications between interface and implementation. This is done by declaring, in this manager, not only a method Main, which creates its component objects, thus assuring ownership, but also handlers for messages to be received, during execution, from the owned interface and implementation objects.

For example, the calculator user, after creating a string of digits in the display area, may press the ENTER button. When this is done, the interface object sends a message to its owner, passing the string displayed. The manager captures this message, translates the string to a real value, and invokes the implementation object to stack up this value. If the user presses the '+' button, again the interface sends a generic message, which when treated by the owner passes to the implementation a value corresponding to the string displayed and a character representing the operation chosen. With this architecture the interface and the implementation do not communicate directly; neither has to know the other, and if they are replaced in the application only the manager may need updating. This kind of program design is being proposed to implement domain change applications [11,12]. A detailed discussion of this program design technique can be found in [9].

To illustrate the reuse of a stand-alone application as a component in other application, again let us consider the calculator, and use generic message passing one more time. To create such a dual-role application we need first to add to the calculator interface, for example, another button, labeled SEND, which when pressed by the user sends the string currently displayed to its owner, again the manager class. The manager must have a handler for this message, which when executed sends a corresponding real value to the manager's owner,

another application where this value is important (the salary of some employee, or the value of a spreadsheet cell, to the manager this does not matter, since the message it sends is generic).

Secondly, we need to add to the manager class another method, called for example Initialize, which is to be invoked when the calculator is a component in another application, and is not the main class in the collection. This method looks very much like the Main method still present in the manager class, in the sense that it creates and displays the interface, and creates and prepares internal implementation data structures, as the stack. If we also take the precaution of adding to the interface class a method that hides all interface objects needed in the component mode, as for example the button SEND, and modify the Main method in the manager to invoke this HideSend interface method, then the architecture is complete.

With this organization, in the stand-alone mode the calculator's manager is the main class: a manager object is created by the TOOL system, and Main is applied to it. As a consequence, the button SEND is hidden. If, on the other hand, the calculator is a component, then after its creation by its owner, the Initialize method can be applied to it, displaying the interface, which now includes the SEND button, and preparing the implementation. The new collection of calculator related classes, the manager, the interface and the implementation, now form a black-box application [1]. We discuss the construction of dual-role applications in [10].

RELATED WORK AND CONCLUSIONS

The discussion of message passing mechanisms as programming language features has its main origins in the CSP [13] and DP [14] proposals. Since then several other conventional programming languages (as opposed to object oriented programming languages) include such features, for example Ada [15] and SR [16].

As Gehani [17] points out, however, this discussion has taken place mainly in the context of operating system design, and not in the context of programming language design. We may add to this remark the observation that the lack of message passing features is even more pronounced in the design of object oriented languages. In fact in Smalltalk [18], Eiffel [19] and C++ [20], object behavior is expressed as (possibly late bound) procedural behavior; unblocking, queue administered, behavior level message semantics is not a native feature of those languages.

This feature can be found in extensions being proposed to conventional languages, especially to treat concurrency, as is the case in [21], where message features are added to C++. In this case, however, the option chosen was to add new type specifiers to the language, implementing for example coroutines and monitors; message passing and receiving are not specified as object behavior, declared in classes.

In the design of TOOL we took advantage of operating system primitives (the queues in Windows) to implement a fully extensible asynchronous behavior mechanism. In this sense our work is more related to that described in [22], where multiple inheritance was used in the

design of classes that implement generic interface tools. In TOOL, we use composition instead of multiple inheritance to achieve this effect. We also relate to the work of Ghelli [23], which proposes linguistic constructs to capture the notion of message passing. As seen above, a set of such constructs is offered in TOOL.

In our view the main contribution in the design of TOOL resides in the mapping of message passing, a necessary feature in the construction of graphical user interface applications. By stating that message handling is also object behavior, asynchronous but still behavior, we have eliminated the low-level concerns programmers usually have with old-fashioned message queues and message dispatching.

Still with respect to this mapping, we recall that, though asynchronous messages are a built-in feature of interface related objects, in the construction of a TOOL application the programmer can usually decide which semantics is best suited to each particular situation faced, due to the extensibility of the language.

We also feel confortable in stating that incorporating asynchronous behavior to classes is equivalent to taking full advantage of the object oriented programming style, provided one considers that the class mechanism exists for modeling objects - because if this is the case, where else would object behavior, of any kind, be placed? At the moment, in fact, other useful behaviors are being studied and will probably be added to the TOOL language in the future:

- object iterators, to be used in association with loops, with co-routine semantics, have been proposed [24];
- exception handlers have also been proposed [25];
- synchronous message handlers, with handshake semantics, and dynamic data structure navigators, with co-routine semantics, are being studied.

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. .

BIBLIOGRAPHY

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

2. 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.

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

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

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

6. C. Petzold, *Programming Windows*, Microsoft Press, Redmond, Wash, 1990.

7. R. Scheifler and J. Gettys, 'The X Window System', *ACM Transactions on Graphics*, 5(2):79-109, Apr 1986.

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

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

10. S. E. R. Carvalho, 'On the Reuse of Applications as Components', Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 7/93.

11. D. Cowan, R. Ierusalimschy, C. Lucena, and T. Stepien, 'Abstract Data Views', *Structured Programming*, 14(1):1-13, Jan 1993.

12. C. Lucena, D. Cowan, R. Ierusalimschy and T. Stepien, 'Application Integration: Constructing Composite Applications from Interactive Components', *Software Practice and Experience*, 23(3):255-276, March 1993.

13. C. A. R. Hoare, 'Communicating Sequential Processes', *CACM*, 21(8): 666-677, 1978.

14. P. Brinch Hansen, 'Distributed Processes: a Concurrent Programming Concept', *CACM* 21 (11): 934-941, 1978.

15. *Reference Manual for the Ada Programming Language*, United States Department of Defense, Jan 1983.

16. R. A. Olsson, 'Using SR for Discrete Event Simulation', *Software Practice and Experience*, 20 (12): 1187-1208, Dec 1990.

17. N. Gehani, 'Message Passing in Concurrent C: Synchronous versus Asynchronous', *Software Practice and Experience*, 20 (6):571-592, June 1990.

18. A. Goldberg and D. Robson, *Smalltalk-80, the Language and its Implementation*, Addison-Wesley, 1983.

19. B. Meyer, *Object Oriented Software Construction*, Prentice Hall, 1988.

20. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

21. P. A. Buhr, G. Dithfield, R. A. Stroobosscher, B. M. Younger and C. R. Zarnke, 'uC++: Concurrency in the Object Oriented Language C++', *Software Practice and Experience*, (22)2:137-172, Feb 1992.

22. N. Guimarães, 'Building Generic User Interface Tools: an Experience with Multiple Inheritance', *Proceedings OOPSLA 1991*, 89-96, 1991.

23. G. Ghelli, 'A Static Type System for Message Passing', *Proceedings OOPSLA 91*, 129-145, 1991.

24. S. C. Porto, and S. E. R. Carvalho, 'Introducing Iterators in TOOL', Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 41/92.

25. S. E. R. Carvalho, 'Exception Handling in Object Oriented Languages: a Proposal', Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 23/92.