PUC

Monografias em Ciência da Computação
nº 25/92

TOOL: a Short Description

Sérgio E. R. Carvalho

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22454-970
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, Nº 25/92
Editor: Carlos J. P. Lucena Julho, 1992

TOOL: a Short Description *

Sérgio E. R. Carvalho

* This work has been sponsored by the Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

TOOL: A Short Description.

Sergio E. R. Carvalho
(PUC Rio-BR, Depto. Informática)

PUCRioInf-MCC25/92

Abstract: TOOL is a programming system designed to simplify the task of building application programs running on graphical user interface platforms. The current version runs above Microsoft Windows. In this report the programming language component of the system is summarized. This is an object and event oriented language, designed to be easily understood by conventional language programmers.

Resumo: TOOL é um sistema de programaçao projetado para simplificar o desenvolvimento de programas construidos sobre plataformas oferecendo interfaces gráficas. A versao corrente funciona sobre Microsoft Windows. Neste relatório a linguagem de programaçao do sistema é sumariamente apresentada. Esta linguagem é orientada a objetos e eventos e foi construida para ser facilmente entendida por programadores convencionais.

Palavras-chave: interfaces gráficas, orientaçao a objetos, eventos.

TOOL: A SHORT DESCRIPTION

S. Carvalho

INTRODUCTION

Software development platforms for the construction of applications driven by graphical interfaces are becoming increasingly popular (Windows for DOS and MOTIF for Unix are examples). Application users benefit considerably from this technique: for example, the fact that application interfaces tend to be similar speeds up the learning process, allowing users more productivity in less time.

What is nice for users, however, is not necessarily so for programmers. The task of developing a Windows application in C, for example, has been recognized as at least non-trivial. For this reason new programming systems for Windows-like platforms are being developed and marketed today. All support graphical user interfaces and most advertise object orientation, a definitive trend for the 90's. We can roughly classify most of these new systems in two categories, described in some detail below:

- common knowledge seekers;
- new paradigm advertisers.

In the first category we place those systems developed as extensions to well-known programming languages, notably C and Pascal. Most conventional or procedural languages, however, have not been originally designed to accommodate asynchronism, or event-driven programming, essential to graphical user interfaces. As a result, when using one such language to construct an application for a modern software platform, one usually has to learn a new, large and unstructured set of functions, in order to code message passing. Moreover, one frequently has to cope with inelegant syntactic and semantic language extensions, which often violate original design principles.

In the second category we place those systems emphasizing a Smalltalk-like approach to object orientation, and even a Smalltalk-like syntax, both unfamiliar to most programmers today. As a result, procedural language programmers trying to embrace object orientation often feel alienated, thus failing to benefit from the obvious advantages offered by Windows-like platforms.

Neither approach seems adequate to us. What seems to be needed are new languages, certainly built for the resources and needs of today (thus including object and event orientation for example), and yet positively utilizing the extensive body of programming knowledge existing today (thus welcoming all conventional language programmers). In this paper one such language, TOOL, will be presented. TOOL accommodates, through object orientation, also the message passing and procedural paradigms. In *classes*, the main language construct, both the synchronous (procedural) and the asynchronous (event-driven) behavior of objects may be specified.

TOOL is actually an object and event oriented programming system, consisting basically of a programming language, an optimizing compiler, a library system, and an abstract machine to allow portability through different environments. The current version of the system is implemented as a software layer above Microsoft Windows.

Our main goal, when designing TOOL, was to simplify the task of building application programs using the extensive facilities provided by Windows and similar environments. To accomplish this, considerable effort has gone 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; message passing, as a special case of the familiar procedure call; polymorphism, simply as an ability objects have of changing classes at execution time.

The main features of the TOOL programming language are presented next. The TOOL programming units are *classes*, for modelling objects and other classes; *methods*, to implement synchronous object behavior; message *handlers*, to implement asynchronous behavior. Methods and handlers are located within classes.

CLASSES

Classes are the meeting grounds for all properties associated with the objects they model. Classes can be *basic*, *conventional*, or *extended*.

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

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

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, perhaps with qualification, if a same name attribute is declared 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 (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.

In the *class representation section* the data structure for the class itself is declared. Class representation components (objects and structures) are only visible inside their family and can be used as common storage among objects of the class.

Special 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 method/handler parameters, as method results, and as locals to methods/handlers.

At declaration (always required) an object is associated with a *base class*, and is modelled by this class. At declaration, a *discipline* is also attached to objects. This discipline affects both the lifetime and the class flexibility of objects. The possible disciplines are *automatic*, *dynamic* and *polymorphic*.

The *automatic* (default) discipline is that of languages like Pascal: space for the object's data structure is allocated on the execution stack, and the object's lifetime is that of the enclosed method/handler, or, for object components, that of the parent 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 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.

Basic class objects are always automatic. Extended class objects cannot be automatic. Any discipline is suitable for conventional class objects.

STRUCTURES

Object structures can be constructed with the *array*, *dynamic array* and *union* mechanisms. Constructors are merely compositions of objects; they are not classes. Structures may appear in the object and class representation sections of a class declaration. They may also be locally declared in methods and handlers. They may not, however, appear as parameters or results in method or handler interfaces.

Arrays are homogeneous fixed-size compositions. Access to array components is done by indexing the array's name with integer expressions. Array components can be of any class and have any suitable discipline. *Dynamic arrays* are created during execution, where special methods are able to set and then get dimension limits. *Unions* are probably heterogeneous compositions where all elements share the same memory positions. Union elements (basic class objects or one dimensional arrays thereof) are accessed with dot notation.

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 which cannot be fully declared in a class, but only in its subclasses.

Parameters and *arguments* correspond according to position, family, mode and discipline. Parameters can be objects or merely argument identifiers. Object parameters correspond through the assignment of a value (input, output or both). Object parameters can have any suitable discipline. Dynamic and polymorphic parameters are automatically created by the system.

The method's *result* 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 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.

STATEMENTS

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

In an *assignment* a left hand side object receives a value that is either obtained from an expression or is the value of an object (variable or constant). In all cases the right hand side value must be of the left hand side declaration class (or descendant thereof). Only common declaration class fields are updated.

Selection is expressed by conventional **if** ... {**elsif** ...} [**else** ...] , and **case** ... {**when** ...} [**else** ...] statements.

A single **loop** statement is used to specify *repetition*. The execution of this statement can be controlled by conditional **exit** and **repeat** statements placed anywhere in the loop's body. The exit statement terminates the loop's execution, transferring control to the statement immediately following the loop. The repeat statement terminates a loop iteration, transferring control to the beginning of the loop's body.

Application statements force operators (methods and handlers) on receiving objects. For readability, class names may qualify operator names. Object returning methods, when applied, produce object values to be used in expressions. Handlers do not produce values. A conditional **return/quit** statement may be used in method/handler bodies to abruptly terminate execution.

EXTENDED CLASSES

Extended classes model objects capable of responding to asynchronous calls. This modelling is done at a very high level, taking full advantage of the class mechanism: TOOL programmers are not concerned with message queues, for example.

To encompass message passing, extended classes contain, besides the sections existing in conventional classes, a *messages* section and the declaration of message *handlers*. In the messages section, the signatures of all messages sent from extended class objects to their owners (the objects that created them) are given. Message signatures have a name, an optional set of parameters, and an optional tag. This tag allows programmers to directly relate messages and their handlers.

Handler declarations contain a signature, an optional message related tag, 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 modelled by the same extended class. Handlers without a sender identification clause treat messages originating from non-owned objects.

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.

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.

A PROGRAM

A TOOL program is a sequence of one or more conventional and/or extended classes. Some extended class in the sequence 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.