PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 41/92

Introducing Iterators in TOOL

Stella C. C. Porto
Sérgio E. R. Carvalho

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 41/92
Editor: Carlos J. P. Lucena Março, 1992

Introducing Iterators in TOOL*

Stella C. C. Porto
Sérgio E. R. Carvalho

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

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio — Departamento de Informática
Rua Marquês de São Vicente, 225 — Gávea
22453-900 — Rio de Janeiro, RJ
Brasil
Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645
E-mail: rosane@inf.puc-rio.br
techrep@inf.puc-rio.br (for publications only)

Introducing Iterators in TOOL

Stella C.S. Porto
Depto. Informática – PUC-Rio
Depto. Telecomunicações – UFF
e-mail: stella@inf.puc-rio.br

Sérgio Carvalho
Depto. Informática – PUC-Rio
e-mail: sergio@inf.puc-rio.br

ABSTRACT

TOOL is an object-oriented, event-driven programming system, designed to simplify the task of building application programs for Windows and similar environments. Its class construct has been designed to allow the definition of behaviors of different natures for the modelled objects. In this report we propose the inclusion of iterators in TOOL, as coroutine-like operators, which provide the programmer with the ability of using control loop objects of any class.

RESUMO

TOOL é um sistema de programação orientado a objeto e dirigido por eventos projetado para simplificar a tarefa de construção de programas aplicativos para Windows e ambientes similares. Sua estrutura de classes foi projetada para permitir a definição de comportamentos de diferentes naturezas para os objetos modelados. Neste relatório propomos a inclusão de iteradores em TOOL, como operadores semelhantes a corotinas, que proveêm o programador com a capacidade de usar objetos de controle de repetição de qualquer classe.

KEYWORDS AND PHRASES

object-orientation; iterators; TOOL programming system; loop structures and loop control objects

1 Introduction

TOOL [1] is an object-oriented, event-driven programming language, designed to simplify the task of building application programs using the extensive facilities provided by Windows and similar environments. It incorporates some of today's most important features for the development of modern software, such as object orientation, message passing and graphical user interfaces.

The object orientation programming paradigm has as its fundamental motivations the concepts of extensibility, encapsulation, protection and reutilization. Its basic ideas of classes and objects evolved from the well-known procedural programming environment.

- Extensibility is the ability to define new data models (classes) and new operations which are used to express object behavior and are contained in classes.
- Encapsulation is achieved through a syntactic entity (class), which contains the definition of the data structure and also the operations performed on the objects modelled by this class.
- Protection guarantees the data integrity of modelled objects, allowing users to access objects only through the operations defined in their corresponding classes.
- Reutilization is achieved through the inheritance concept. Class hierarchies can be created and then stored in libraries for reuse, cutting down on software development costs.

Iterators are a generalization of the iteration facility available in programming languages. They are user-defined control units, which provide to the programmer the ability of using loop control variables of any abstract data type. The iterator can be seen as a special operation encapsulated within a certain class. When applied to an object in a loop statement, it produces a sequence of values for this object, thus controlling the iteration. This kind of operation abstraction is another important step towards full extensibility and software reutilization, promoting instances of user-defined abstract data types to first class entities.

This work presents the basic concepts concerning iterators and object orientation in TOOL. It describes syntax, implementation restrictions and details for the inclusion of iterators as a new control structure in TOOL. The next section overviews some important concepts, which provide the basis for understanding the TOOL environment and the proposed iterator facility. In section 3, iterators are introduced through its basic concepts. In section 4, the inclusion of iterators in TOOL is proposed and discussed. The final section presents some concluding remarks.

2 TOOL – The Language

The main TOOL feature related to iterators is object orientation. This section focuses on the fundamental concepts of the object orientation feature of TOOL, namely: classes, objects and methods. More details on TOOL may be found in [2, 3, 1].

2.1 Classes in TOOL

The class is the main abstraction mechanism in object oriented systems. A class contains the data representation of the objects it models, and the operations applicable to those objects.

A class in TOOL can be described as a program unit, serving two essential purposes:

- the modelling of objects, defining basically their data structure, and the operations that can be applied to them;
- the creation of more specialized classes.

The binding between classes and objects in TOOL is very strong, usually performed at compilation time. This means that once declared as being an instance of a certain class, an object must behave according to the operations described in that class. Therefore, the compiler is able to, at translation time, verify the correctness of all operations applicable to an object.

The TOOL system provides the user with a set of built-in classes and a mechanism for the construction of user-declared classes. There is no nesting of classes in TOOL, and classes cannot be locally declared in methods. From now on, the general term class will stand for user-declared classes, and built-in classes will be explicitly referenced in the text when necessary.

Classes can contain:

- an inheritance clause;
- a section defining the structure of class objects;
- a class data structure to be shared among objects of the class;
- a section defining constant values for objects of the class;
- private and/or public class operations.

The inheritance mechanism in TOOL establishes a subclass relationship between classes. Inheritance in TOOL is simple: each class has only one direct superclass.

2.2 Object Behavior

Classes are passive entities, used in programs in the creation of other classes and in the modelling of objects. Objects (instances of classes), on the other hand, are dynamic since they exist during program execution.

TOOL supports the existence of variable and constant objects. This text deals specifically with variable objects. These can be introduced in the class declaration sections concerning object representations and shared representations. They may also appear as method or handler parameters, as method results, and as entities local to methods or to handlers. Objects can also be used as elements in structure declarations.

In the design of object oriented languages to be compiled, such as TOOL, all objects must be declared. This indicates to the compiler the expected behavior for the object, and also the amount of memory it will need at execution. At declaration, besides binding an object to a class, an object is associated with a *base class*, which defines the data structure for the object (its state) and the operations that can be applied to the object (its behavior). The duration of this association depends on the *lifetime discipline* of the object, also established at declaration. *Initial values* may also be explicitly given to objects.

The discipline of an object involves both the binding strength and the lifetime of the declared object. The binding strength measures the facility objects may have of changing classes at execution-time. The lifetime represents the span of execution-time during which the object actually exists, and may be referenced. There are three possible lifetimes for TOOL objects: automatic, dynamic and polymorphic.

In the *automatic* discipline (assumed as the default discipline when none is specified at declaration) the lifetime is that of the enclosing method. Automatic lifetime objects are very efficiently handled by the TOOL system, and should be used whenever possible.

Dynamic objects are not allocated/deallocated on method entry/exit; instead, the programmer controls their existence using the built-in methods CREATE and DISPOSE. Dynamic objects are useful in the representation of data structures which have their sizes increased/decreased at execution time. The dynamic lifetime discipline affects the existence of objects, but not their associations with base classes: throughout their programmer controlled life cycles, dynamic objects are associated exactly with the base classes in their declarations.

Objects declared with the *polymorphic* lifetime discipline also have their existence controlled by programmers, again through the application of the methods CREATE/DISPOSE. Their class association, however, may vary at execution time. The base class of the polymorphic object plays an important role in the definition of the class variation range for the object. The subclass structure whose root is the base class of an object, defines a set of subclasses. The object is allowed to change from its base class to any class of this set.

Polymorphic class variations occur when a polymorphic object receives the application of the built-in method NOW_IS. An argument must be provided when this application occurs: an already created dynamic or polymorphic object of a subclass of the declaration class of the object, or exactly this same declaration class. The application of NOW_IS to a polymorphic object changes the class of this object, if the class of the argument is different than the class of the receiver. When this change occurs, the behavior of the polymorphic object changes accordingly: a new set of methods is applicable to it. In TOOL, polymorphism is a feature of objects, established at declaration, which means that the cost of polymorphism is only paid when due. This advantage results from the combination of late binding with the redeclaration facility for operations in subclasses.

2.3 Methods in TOOL

Class operations define the behavior of class instances (objects), being the only operations applicable to those instances.

Two kinds of operations can be currently declared in TOOL: methods and message handlers. Methods are the TOOL equivalent to procedures and functions in languages like Pascal and FORTRAN, being bound at execution-time only when applied to objects declared as polymorphic. Message handlers implement the asynchronous operation feature in TOOL. There is no synchronism between the message handler and the invoking unit. Several message handlers may be active at the same time, perhaps along with methods.

Results are obtained by the application of operations to receiving objects. A method may change the object's internal state, produce side effects, or produce a result to be used in an expression. In fact, the receiver is a distinguished argument, in TOOL actually outside the argument list:

$x < -P(\cdots)$ -- x is the receiver of method P

or

$y << -M(\cdots)$ -- y receives the message M.

When the execution of a method terminates, the next instruction to be executed is the one following the method application statement. Only the most recently applied method is active. Other methods

are either suspended or inactive.

Methods in TOOL can be either built-in or created by class constructors. User declared methods, located in user declared classes, are needed to supplement the built-in behavior.

The possible attributes for TOOL methods are **inline**, **private**, and **virtual**. The **inline** attribute is actually a request to the compiler, to replace method application statements with the code for the method itself. This is done in order to obtain better execution time efficiency. This attribute is valid for both public and private methods.

The **private** attribute indicates a method which can only be invoked from within methods and/or handlers in its own class; a method not visible to users of the class. In all other respects, private methods are no different than "public" methods.

The **virtual** attribute is associated with methods which cannot be implemented in a class A, and which, for every subclass of A, must either be implemented (through redeclaration), or in turn declared **virtual**. In other words, for a virtual method in a class, only the method interface (name, parameters and result, if any) is given.

Classes containing virtual methods (virtual classes) can only be used in the declaration of objects of polymorphic lifetime discipline, which furthermore, in order to posses full behavior, must receive a NOW_IS application.

A method declaration may contain the declaration of parameters. Parameters can be either full-fledged objects, or merely argument identifiers.

Object parameters are locally known objects, can be of any class, and can have any lifetime discipline suitable to that class. Three object parameter modes are available for TOOL methods: **in**, **out**, and **inout**. In each case, the correspondence between argument and parameter is made with the passing of object values.

Argument identifiers, on the other hand, are simply place holders for the corresponding arguments in the method application. Argument identifiers are designated by the mode **id**. Parameters for **inline** methods can only be argument identifiers.

In TOOL the reserved word **self** designates the receiving object, a distinguished parameter. Access to the receiver's components is possible through **self**, with dot notation. **self** is not an object; it is merely an identifier, to be used within operation bodies. The binding of **self** to a receiving object is automatically made at the time the operation is applied, and remains until operation termination.

Objects of any class and suitable lifetime discipline may be returned from methods. Returning objects are specified in the **returns** clause of a method's signature. The returning object is, as

is the receiving object, a distinguished parameter. Its local value may be repeatedly changed by assignment, when the method's body is being executed. As far as the overall execution is concerned, however, only the last such value matters, since this is the one passed back when the execution of the method ends.

Single objects and/or structures of objects may be locally declared in methods. This being the case, their life cycle is bound to that of the enclosing method. Automatic objects behave like local variables in Pascal or C: they are created on method entry, and destroyed on method exit.

The dynamic and polymorphic lifetime disciplines can also be attached to local objects, or to components of local object structures. In this case explicit creation must occur, and explicit destruction may occur.

The execution of a method terminates either normally, after the execution of the last statement in its body, or abruptly, when a possibly conditional **return** statement is executed. In either case, the last value associated with the return object, if there is one, is returned to the calling environment. Several **return** statements may exist in a method's body, placed anywhere in its statement sequence.

3 Iterators – Underlying Concepts

Iterators are a control abstraction available in some programming languages, notably CLU [4]. They permit users to iterate over arbitrary types of data in a convenient and efficient way, preserving abstraction by specification [5].

In order to clearly understand the utility and functioning of iterators, we may start from analysing repetition structures. High-level languages provide control structures that allow the programmer to specify looping over a certain set of instructions. We are essentialy interested in counter-driven iterative control structures[1], used to model the common case in which the number of iterations is determined by a finite sequence of values assigned to a so-called loop control variable. Frequently, before the iteration begins, the lower and upper bounds of the control variable are determined. In FORTRAN, the control variable can only be of type INTEGER. Pascal allows counter-driven loops where the counting variable is of any ordinal type.

Pascal also prescribes that the control variable is not to be altered in the loop. The value of the variable is also assumed to be undefined outside the loop. These restrictions, far from placing arbitrary constraints on the programmer, contribute to the readability and maintainability of the

[1] As opposed to the condition-driven repetition structures.

resulting programs by limiting the scope of the loop counter [6].

Further abstraction is provided by languages such as CLU [4], which allow the programmer to have control variables of any abstract data type (cluster in CLU) and provide constructs for specifying how the sequence of values of such control variables is to be produced. These constructs are called *iterators*, and are declared in the cluster modelling the control variable. An *iterator* yields the sequence of values a given loop is to span over when applied to the loop control variable. The elements of the collection are provided one at a time. The policy that selects the next element of the collection is hidden from the user of the iterator and implemented by the iterator itself. The iterator is responsible for producing the items, while the module containing the loop defines the actions to be performed.

In the following example from a CLU program [6], *atom*, the loop control variable, is of the user-defined type *node*, and *list* is an iterator. The instructions are meant to retrieve and manipulate all the nodes belonging to a *list*(x).

```
for atom : node in list(x) do
   perform an action on atom
```

The iterator used above can be specified by a module that takes a parameter of the abstract type *linked list* and delivers a parameter of abstract type *node*.

```
list = iter (z : linked list) yields (node)
        ...
        yield(n)
        ...
   end list
```

At each iteration of the above loop, the module *list* is activated, and a node yielded by it is assigned to *atom*. As a consequence of yielding, the iterator is suspended, its local environment is retained, and control flows back to the loop. At each loop iteration, the iterator is resumed with its previously saved local environment and executes from the instruction following the last **yield**, much like a coroutine. When the iterator indicates that the sequence of objects is exhausted, the **loop** statement terminates.

Iterators are efficient to execute. Yielding from an iterator is like a call: the body of the **for** loop is "called" from the iterator. Resuming an iterator is similar to a return: the loop body "returns" to the iterator. Therefore, the cost of using iterators is at most one procedure call per execution of the loop body; the cost may be less because of compiler optimizations [5].

With the addition of iterators, variables of user declared types can really be considered first-class

entities.

4 Iterators in TOOL

In the last section, iterators were presented in the environment of CLU – not an object oriented language. In order to understand iterators in an object orientation framework, a few adjustments must be made. Here iterators are units declared inside *user-declared classes* (while in CLU, they were built inside clusters), so they are only to be applied to objects belonging to this specified class, just as any ordinary method. This means, that instead of referencing a control variable, we have a *control object* whose sequence of values is in the same way responsible for controlling the number of repetitions of a certain loop.

We first briefly present how repetition structures work today in TOOL, before introducing iterators in TOOL. A single **loop** statement is used to specify repetition in TOOL (the syntax is presented bellow). The execution of a loop statement can be controlled by conditional **exit** and **repeat** terminators, placed anywhere in the loop's body, and in any number.

loop

    –

    –

    exit [**when** *boolean expression*]

    –

    repeat [**when** *boolean expression*]

    –

    –

end loop

The **exit** terminator terminates the loop's execution, transferring control to the statement immediately following the loop. The **repeat** terminator terminates a loop iteration, transferring control to the beginning of the loop's body. Both terminators are only valid in loop bodies.

4.1 Syntax

The introduction of iterators in TOOL, follows the syntax used in CLU [5], and is illustrated in the skeleton bellow:

```
loop
   for obj ← itername[(arguments)];
                    -- obj is the control object to which itername is applied
                    -- An iterator may be declared without parameters
       -
       -
       exit ···
       -
       repeat ···
       -
       -
end loop
```

The reserved word **for** indicates that the loop is controlled through the application of an iterator. *obj* is the object to which the iterator, named *itername*, is applied. The iterator *itername* must be implemented in the object's class or superclass thereof, with the following syntax:

```
iterator itername [(parameters)];
   -
   -- local declarations
   -
begin
       -
       -
       yield [when boolean expression];
       -
       -
end iterator
```

The reserved word **iterator** indicates the nature of this active unit. The **begin-end** iterator pair delimits the iterator's body.

The statement **yield [when** *boolean-expression*]; may be substituted for **yield**'s in **if** statement branches.

Single objects and/or structures of objects may be locally declared within iterators, just as in any method. Their life cycle is bound to that of the enclosing iterator. The dynamic and polymorphic lifetime disciplines can also be attached to local objects, or to component of local object structures. In this case, explicit creation must occur, and explicit destruction may occur.

4.2 Semantics and Restrictions

In order to discuss semantics and implementation issues, it is useful to compare iterators with procedures (methods, in TOOL). We begin by considering the **yield** iterator statement, as opposed to the **return** statement that can be used to terminated methods.

The execution of a method terminates either normally, after the execution of the last statement in its body, or abruptly, when a possibly conditional **return** statement is executed. In either case, the last value associated with the return object, if there is one, is returned to the calling environment. Several **return** statements may be placed anywhere in a method's body.

At each **yield** the iterator is suspended, providing a new value for the control object. The iterator's context is not lost. On the next iteration, the iterator will continue from the point where it stopped, executing the first instruction just after the most recently executed **yield** statement. The sequence of control object values produced by the iterator until its termination is encountered, defines the number of iterations of the active loop.

No **return** statement is allowed in the body of an iterator. When the last value is yielded, a flag in the iterator's environment is set signaling the end of the iterations (see section 3). This flag is tested each time the iterator is resumed, determining if the iteration should proceed or not.

The only permitted atributtes for iterators are **private** or **virtual**. When no attribute is present at declaration, the iterator is considered to be public, which means that it can be applied to an appropriate object (an instance of the iterator's class or subclass thereof).

The control object is read-only in the loop's body. This means that:

- it can be used only as an input parameter to other methods;
- it cannot be assigned to;
- it cannot be subject to method applications, to avoid side effects.

However, the control object is freely accessed in the iterator's body, where its new values are produced.

The value of the control object after loop termination is the value obtained after the last activation of the corresponding iterator, even if loop termination is caused by **exit** statements, which may be found in any number in the loop's body. In the case of a **repeat** statement, the iterator is called again and the control object will have the value obtained in this last activation of the iterator.

Iterator's parameters are exclusively of input mode. This restriction is due to the main motivation for the inclusion of iterators in TOOL, which is to produce a sequence of values to control an iteration process. There is no sense in generating side effects through the activation of iterators. These input parameters may be of any lifetime discipline, following the same rules established for ordinary methods in section 2.3.

The control object may be of any lifetime discipline. Dynamic and polymorphic objects must have obviously been explicitly created before the application of the corresponding iterator.

The redeclaration and the inheritance of iterators follow the same rules given for ordinary methods in TOOL, which means that iterators defined in a certain class will be inherited by its subclasses if not redeclared. Late binding of the iterator code to be executed may occur when the control object follows a polymorphic discipline.

Loop control objects with polymorphic discipline may not change classes during the execution of a certain iterator. This is rather a consequence of the nature of iterators, which when used should be the only providers of values to loop control objects. (Recall that no assignments to loop control objects are allowed in the loop's body.)

Nested loops with identical or different iterators are allowed. If identical iterators are used in nested loops, the control objects used in each **for** clause must be distinct, because the control object of a certain iterator must not be changed in the loop's body, for example by another iterator. In the iterator's body, loops with or without other iterators are also allowed. Recursion is allowed in the iterator's body.

4.3 Implementation

The representation structure of the object is the structured set of values of all data components of the object, which are declared in the object representation section of the object's class. With the inclusion of iterators, it is necessary to maintain a pointer to the iterator's environment created when the iterator is applied to the control object. This environment includes the following information concerning the iterator:

- parameters, local objects;
- *entry point*;
- termination flag.

As mentioned before, at each **yield** the iterator is temporarily suspended. After the loop's body is executed, the control is passed to the iterator, which resumes from the instruction following the last executed **yield**. This instruction is given by the *entry point.*

When the final **yield** of the iterator is executed, the *termination flag* is set to signal the end of the iteration. The loop's body is executed for the last time. The iterator's environment is abandoned and the execution continues from the first instruction following the **end loop** delimiter.

In TOOL, the way objects are implemented in memory depends on their lifetime discipline. Automatic objects have their representation structure in the stack as automatic variables in languages like C and Pascal. Also for automatic objects, the pointer to the iterator environment is maintained in the stack. On the other hand, a dynamic or polymorphic object is implemented as a pointer in the stack, which points to a descriptor in the heap. The descriptor has pointers to the representation structure of the object and to the current iterator's environment (while the iterator is active), both of them located in the heap. This organization is shown in Figures 1 and 2.

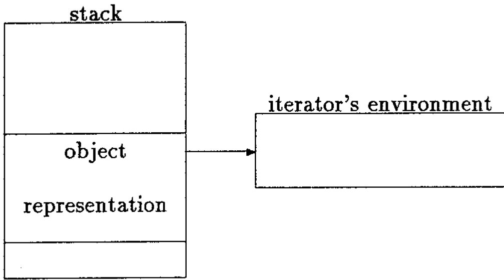

Figure 1: Memory Organization for Automatic Control Object

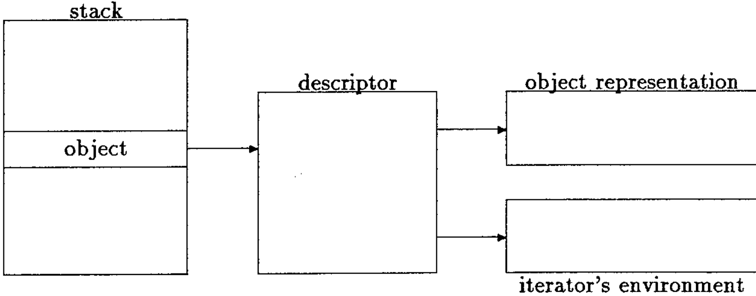

Figure 2: Memory Organization for Dynamic or Polymorphic Control Object

5 Concluding Remarks

This work describes the most important features concerning the inclusion of iterators in the object oriented language TOOL. An iterator is introduced as a special method, which provides the programmer the ability of controlling a certain loop. The number of iterations of the loop will depend on the sequence of values of the control object.

At first glace, iterators seem to be an important operation abstraction for an object oriented language. However, it is worth noting that iterators are essentially application dependent, and this fact represents a fundamental contradiction towards the underlying concepts of object orientation. The dependency on the application seems to determine severe restrictions on the extensibility and reutilization of these constructs, fundamental features of the object orientation paradigm.

Although this issue may tend to qualify iterators as restricted tools in the object oriented programming environment, they are in fact essential constructs for the development of a flexible and rational object oriented programming technique, since they promote objects of user declared classes to first-class objects. In this sense, the inclusion of iterators in an object oriented language like TOOL is not only completely justified, but also necessary.

References

[1] Sérgio Carvalho. TOOL: a Short Description. Monografias em Ciência da Computação 25/92, Depto. de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 1992.

[2] Sérgio Carvalho. *TOOL for Windows - The Object Oriented Language - The Language.* SPA - Sistemas, Planejamento e Análise.

[3] Sérgio Carvalho. *TOOL for Windows - The Object Oriented Language - Fundamentals.* SPA - Sistemas, Planejamento e Análise.

[4] B.H. Liskov, E. Moss, C. Scheffert, R. Scheiffer, e A. Snyder. *CLU Reference Manual.* MIT Laboratory for Computer Science, Cambridge, MA, Julho 1981. In Computation Structures Group Memo 161.

[5] Liskov, Barbara and John Guttag. *Abstraction and Specification in Program Development.* The MIT Electrical Engineering and Computer Science Series. The MIT Press and McGraw-Hill Book Company, 1986.

[6] Ghezzi, Carlos and Medhi Jazayeri. *Programming Language Concepts.* John Wiley & Sons, Inc., 1982.