Specifying Object Behavior

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

ABSTRACT

The procedural behavior, the only one usually available in object oriented languages, is necessary in the construction of programs. However, other well-known behavioral semantics in the conventional programming world have no counterparts in object orientation. In this paper we propose the inclusion, in classes, of operations to handle asynchronous messages, exceptions, and the production of sequences of values for loop control objects, offering examples of their uses.

KEYWORDS AND PHRASES

object orientation, object behavior, asynchronous message, exception handler, iterator, method

Specifying Object Behavior

INTRODUCTION

Object oriented programming systems are being considered at least as viable alternatives in the production of good quality software [1]. Such systems usually have a programming language, a program executor, a class library for the modeling of objects, and are also frequently associated with suggestions on how to design programs, as are the cases of Smalltalk [2, 3] and TOOL [4, 5, 6]. Also, several books have already been published on the analysis and design of object oriented systems [7, 8, 9, 10, 11, 12].

The central idea in an object oriented language is the concept of *class*. As is always the case in emerging disciplines, however, there are today several interpretations for this concept. Among the most common we find a class as a model for objects, to be considered mainly during compilation, and specifying both the *data structure* for modeled objects and the *behavior* they are allowed to have.

The data structure for objects is actually a composition structure, familiar to conventional programmers, and such that already existing classes model fields, or components, of the objects to be declared or created from the class being defined. The behavior of objects is specified by active programming units that are to be executed when a class object receives a stimulus.

In most object oriented languages today this behavior is exclusively procedural, meaning that:

- the execution of the operation generating the stimulus is suspended until the actions corresponding to this invocation, declared in the receiving object's class, are executed;
- the referencing environment corresponding to these actions is allocated on the top of the execution stack;
- control data, such as return addresses and dynamic links, are also saved on the stack;
- the flow of control is transferred to the operation to be executed;
- when this operation terminates, the flow of control returns to the invoking operation.

It is also sometimes the case that the discovery of the operation to be executed can only be made at execution time, since it may depend on the class at that time modeling the object.

In this paper we argue that:

- the procedural semantics, usually represented by *methods*, is necessary in the design of classes, but may not reflect sufficiently well all stimuli that class objects may receive;
- other operations, well-known in the conventional world, should also be present in object oriented languages;
- the class mechanism, essential in object oriented languages, is the perfect framework for the inclusion of these new operations.

Initially we propose a syntax and a semantics for classes, one in which the new operations to be proposed elegantly fit. We then present other behaviors for class objects: the passing and receiving of asynchronous messages, the handling of exceptions, and the production of new values for loop control objects. These are all behaviors well understood in

the conventional programming world; in this paper we simply try to place them in the object oriented world in such a way that conventional programmers can recognize them when changing environments. The familiarity we strive for is important in the opening of object orientation to conventional language programmers.

A VIEW OF CLASSES

For the purposes of this paper, we adopt Wegner's view that a class is basically an implementation of an abstract data type affected by inheritance [13], to be used by the language compiler to validate all operations applied to objects. This "static" or "passive" view of classes is extremely convenient for the popularity and ease of use of new object oriented languages: classes can thus be viewed as mere extensions of types, a familiar concept to conventional language programmers. Another advantage of classes as static structures modeling objects is the strong typing that can be obtained: language compilers can generate efficient, optimized code by discovering, each time a strongly bound object is used, to which class it belongs. Also, in this view, classes can be designed, compiled and tested separately, and can then be stored in libraries for further use. To consolidate this view, the following skeletal definition of class is suggested:

class class-name

inherits class-name,...

exports fields,... operations,...

imports class-name,...

structure field,...

behavior operation,...

end class

The **inherits**, **exports** and **imports** clauses constitute the class interface. The **exports** clause lists the fields and operations visible outside the class. In the **imports** clause we declare all external classes used to model, for example, structure fields or local operation objects.

In the **structure** section, we define the field composition necessary to describe the state of a class object. This composition is like that found in conventional language records, but here inheritance also plays a part: the data structure for objects of class A is found by adding to the fields declared in class A the fields found in the declarations of all superclasses of A, listed in class A's inheritance clause, and so transitively on. This is usually accompanied by disambiguating rules, applied when a same name field appears in more than one class in this inheritance chain.

Access to object fields is in general restricted to the class operations defined in the behavior section, described below. In this way an abstract view of the class is presented to users, who can then affect the state of an object solely through class operations.

In the **behavior** section we declare all operations applicable to the modeled objects. This is where we postulate greater variety, as discussed below. To represent the usual procedural behavior, the following syntax could be adopted:

method method-name [(parameter,...)][**returns** class-name object-name];

[local-declaration,...]

begin

statement,... -- possibly including **return** [**when** expression];

end method

The **returns** clause, the parameter list and the local declarations are optional. The possibly conditional **return** statement provides an abrupt method termination, destroying its referencing environment.

THE HANDLING OF ASYNCHRONOUS MESSAGES

The construction of programs presenting to their users a graphical control interface is becoming increasingly popular. The Microsoft Windows operating system [14] has been built specifically to handle this kind of application; other such platforms are the X-Window system [15] and MOTIF [16]. In a typical application, its user can for example press a button or make a menu choice, thereby asynchronously interfering with the execution of the application.

In operating systems such as these there usually exists a low level message handling mechanism, involving a message queue and a loop statement that continuously tests the existence of messages in the queue. If messages are found, one is taken out of the queue and its handler is invoked for execution. A mouse click over a button and a menu choice are examples of user actions that place messages on the queue.

The construction of an application at this level requires that the programmer controls the queue, and needs a typically unique routine that handles all messages sent during execution. This routine consists of a multiple options selection statement conditioned by the messages to be handled. In fact this kind of program was being built in the 60's, when languages for the construction of operating systems were developed.

The handling of asynchronous messages in an object orientation framework can improve considerably on that. In the next paragraphs we present the message passing and receiving facilities available in TOOL.

Classes in TOOL exist solely for the modeling of objects, defining their state and behavior. This behavior can be procedural, as described above, but it can also be asynchronous: TOOL objects can send and receive messages. Some of these objects are visual control objects, part of user interfaces. These are objects modeled by built-in library classes, and their asynchronous behavior is predefined. For example, pushbuttons send a message "Clicked". However, the message handling mechanism in TOOL is fully extensible: user-declared classes can model message passing and receiving objects. For example, in the construction of a payroll application, the production of reports can be handled asynchronously: when one such report is needed, a message is sent to an object of the class Payroll, which has in its behavior section a handler for this message. Asynchronism is in this case justifiable, since reports do not return information needed in the continuation of the application.

In this system both the queue and the centralized handling routine disappear. The queue and its control loop still exist at the operating system level, since TOOL runs over Windows 3.1; the routine, since TOOL is an object oriented system, is replaced by object behavior. This places at a much higher level the construction of graphical user interface applications.

The semantics of message handlers differs from that of methods in the following:

- the operation containing the message sending statement does not have its execution suspended;
- the message handler can only receive input parameters;
- the message handler does not return resulting objects;
- the flow of control does not continue with the statement following the message sending statement.

These precautions are necessary, since it is possible that the object sending the message (the one receiving the operation containing the message sending statement) is destroyed before the message is attended to. At a level invisible to application constructors, this semantics also contains queue and loop, but this is the concern of the compiler, generating adequate code.

To represent the ability objects have to respond to messages, the following syntax could be adopted:

message handler for message-name [(parameter,...)][**from** object-name,...];
    [local-declaration,...]
begin
    statement,...
end handler

The **for** clause in the handler signature designates a message that can be handled by objects of the class being declared. This clause is mandatory. The optional **from** clause

indicates objects that can send this message and that will cause this same message handling. For example, the class below could be used in the construction of an interface consisting of a window and two buttons:

```
class Interface;
    imports Window, PushButton;
    structure
        Window      w;
        PushButton  ok, cancel;
    behavior
        method  R;
        begin
            w <- CREATE;
            ok <- CREATE;
            cancel <- CREATE;
            ...
        end method
        message handler for Clicked from ok;
            -- local declarations and statements
        end handler
        message handler for Clicked from cancel;
            -- local declarations and statements
        end handler
end class
```

When the method R is applied to an object of the class Interface, the objects 'w', 'ok' and 'cancel' are declared, created, initialized and shown. As a result, a window and two buttons exist on the screen. When the interface user presses a button, the built-in message "Clicked" is sent. This message is trapped, sent to the queue, and when removed a handler for it must be found, otherwise nothing happens. This handler is found in the class where the button was declared and created: namely, the Interface class. Objects of this class can respond to this message; they have this asynchronous behavior. In the example above, since two buttons of obviously different meanings are present in the interface, we need two handlers, one for each button. This is the reason for the **from** clause in message handler signatures. An absent **from** clause indicates a handler for messages sent from objects outside the scope of the class.

Message handlers declared in classes specify the receiving behavior of class objects. The sending behavior, if needed, must also be specified. Syntactically this could be done in the **exports** clause of a class declaration, by simply including the signatures of messages to be sent. The semantics of message sending, however, needs to be defined with care. For example, we want to preserve the encapsulation facilities available in classes, to increase their reuse potential. If we state, in the signature of a message to be sent, the name of a receiving object belonging to another class, then we tie the class being declared with the one modeling the receiving object, and lower the reuse potential of our class. As a consequence of this, the signature of a message to be sent contains only the message name and a possible list of parameter classes. This being the case, which objects can receive these messages? The answer is that messages sent from objects of a certain class can only be received by their owners, or the objects that created them. In the Interface class example above, the owner is

an object of this class; so the "Clicked" message sent from an owned pushbutton, when taken out of the queue, is received by an Interface object. In this solution the object sending behavior specified in classes preserves encapsulation by allowing the definition of messages sent to generic owners; users of such classes need only know which messages are sent, and, if necessary, declare the corresponding handlers.

To exemplify, consider a reusable calculator, one that contains in its interface, besides buttons for digits and operations, a button labeled "SEND". The purpose of this button, when pressed by the calculator user, is to send to the owner of the calculator the current display value, without knowing whether this value goes to a spreadsheet cell, or whether it is a new salary for an employee. To accomplish this, the calculator class declares in the **exports** section, as sending behavior, the message Send, and as receiving behavior, the following handler:

```
handler for Clicked from s;
begin
        owner <<- Send ( disp );
end handler
```

where 's' is the name of the button "SEND", 'disp' is the string on the display, and **owner** is the generic designator for objects modeled by classes that include the calculator. These classes must in turn have:

handler for Send (STRING) **from** calc;

begin

-- what to do with the display value

end handler

where 'calc' is the name of the calculator object.

The message handling semantics described above has been implemented in the TOOL programming system [5, 6].

EXCEPTION HANDLING

The initial objective here is to attach to classes active units representing the exceptional behavior of objects. Furthermore, we want to specify this behavior in a manner recognizable by conventional programmers trying to embrace object orientation.

We begin by stating basic assumptions in the exception handling to be proposed. First we decide, as is the case in conventional languages like Ada, that the routine raising an exception must terminate, since it is incapable of fulfilling its contract, in Eiffel's terminology [17]. We also accept that this termination may be preceded by the execution of a special section of code, still in the environment of the raising routine: a local exception handler. If such a handler exists for a raised exception, then the exception is canceled out, and the execution of the application continues normally.

Next we also take from the conventional world that, if no local handler is available (and class handlers, proposed below, are also not available), then the responsibility for the handling of the exception raised lies with the operation that invoked the raising operation.

Local exception handlers could be declared as follows, for any kind of operation:

operation operation-name [(parameter,...)] ;

[local- declaration,...]

begin

statement,... -- including **raise** exception-name [(argument,...)];

exception -- local handlers:

when exception-name [(parameter,...)],...

then statement,... -- including **raise**;

...

end exception

end operation

In the absence of a local handler, the raising operation terminates, the exception is still raised, and a handler must be found at another level. This is where the class concept offers an exception handling level between that of local handlers and that of invoking operations: the class level exception handler. This handler is to be executed in the absence of a local handler, and, if existent, prevents the propagation of the exception to the calling operation. In a sense this handling is also local, since it takes place in the environment of the class, and treats the exception before it reaches the class' client. A handler like this could be used, for instance, to

define the actions to be taken when the bounds of an array that is part of the structure of class objects are violated.

The following syntax could be used to declare class level exception handlers:

exception handler for exception-name [(parameter,...)] ;

[local-declaration,...]

begin

statement,...

end handler

The semantics of this handling can be explained by supposing the application of an operation R to an object 'x', and considering the cases below:

- no exceptions are raised in R, which terminates normally;
- an exception 'e' is raised in R, which has a local handler for 'e': the code corresponding to this handler is executed in the environment of R, the exception is canceled, and R terminates normally;
- an exception 'e' is raised in R, which does not have a local handler for 'e', but a class handler for 'e' exists in R or in any of its superclasses: R terminates, the exception handler code is executed in the class' environment, the exception is canceled, and the execution of the application continues normally;
- an exception 'e' is raised in R, and neither local nor class handlers exist for 'e': R terminates, and the exception is propagated to operations expecting a return from R (we recall that message handlers do not return to the calling environment).

The propagation of an exception to some invoking operation may be considered as information being exported from the raising class; the **exports** clause in the class declaration should contain the signatures of all exceptions raised but not handled in the class. A detailed discussion of class level exception handlers can be found in [18].

ITERATORS AND NAVIGATORS

In this section we present two situations where the semantics of coroutines is useful in object oriented languages: in the construction of iterators for the production of values for loop control objects, as is done in CLU [19], and in the navigation of a dynamic data structure such as a list. In both operations the receiving object controls a loop. The operations are applied to this object to produce, outside of the loop's body, the next value for this object. This reduces the size of loop bodies, improving the readability of programs. Besides, the declaration of iterators and navigators in classes increase the reuse potential of these classes.

We further characterize iterators and navigators:

- an *iterator* produces the next value of a loop control object by *constructing* this next value from the existing value and its own execution context (referencing environment plus code);
- a data structure *navigator* obtains the next value of a loop control object by *displacing a cursor* through an existing data structure, using the current cursor position and its own execution context (referencing environment plus code).

In the latter case the data structure accessing discipline is a primary factor: the next value obtained by a single link list navigator must obey the linking discipline adopted by the single link list.

As an example of an iterator, we offer the arithmetic progression iterator present in most conventional programming languages: the initial-value-to-final-value iterator, which produces, without programmer intervention, the next value in the range specified. As an example of a data structure navigator, consider a first-to-last single link list navigator, which at each loop iteration produces the next list element for processing.

Just as we did in the previous sections, the purpose here is to place, in classes, operations that implement the constructions and traversals mentioned above. We note that the classes modeling objects that can receive the application of a navigator must define dynamic data structures.

The following syntax could be used to declare iterators:

iterator iterator-name [(parameter,...)];

[local-declaration,...]

begin

statement,... -- including **yield** [**when** expression];

end iterator

With respect to semantics, we observe that iterator parameters must be exclusively of input mode, since iterators exist with the well defined purpose of generating a series of values for loop control objects; this being the case, side effects should not be created with the application of iterators. For this same reason, iterators do not return objects; they simply change the value of the loop control object, which they freely access in their statements through the generic receiving object designator **self**.

The basic semantics for iterators is that of coroutines, described below in contrast with that of methods. First we consider the **yield** iterator statement, as opposed to the **return** statement found in methods. The latter, when executed, destroys the method's environment and returns the flow of control to the operation invoking the method; the former, on the other hand, halts temporarily the execution of the iterator, providing another value to the controling object. The environment of the iterator is retained, since it is important to obtain the next value; also, the iterator's access to the receiving object is maintained.

When the execution of a method terminates, the address of the next instruction to be executed is known, being saved in the method's referencing environment. If the method is applied again, a new referencing environment is created, a new return address is saved, and the execution starts from the first statement in the method's body. In iterators, when a **yield** statement is executed, the flow of control is sent to the first instruction in the loop's body; this address is saved in the iterator's environment. When the iterator resumes its execution, each time the flow of control goes through the loop's header, the instruction to be executed is the one following the last yield statement executed; this address is also saved in the iterator's environment, kept between value producing cycles.

An iterator terminates normally when the last value in the sequence specified by its statements is produced. An abrupt termination can also exist, when a **return** statement is executed. In both cases this termination also signals the end of the loop.

The semantics of coroutines is described in detail in [20]. A proposal for the inclusion of iterators in TOOL can be found in [21].

The following loop structure could be associated with iterators:

loop for object-name <- iterator-name [(parameter,...)];

statement,...

end loop

The use of the loop control object in the loop's body should be defined with care. Since the use of an iterator in a loop defines a discipline for the production of values for the loop control object, it seems reasonable to suppose that these values cannot be changed inside the loop. Also, the last value produced by the iterator is the one the control object has when the loop terminates.

To exemplify the use of iterators, consider a class modeling points represented by a pair of cartezian coordinates. Consider also the existence of an iterator in this class that produces points, from the origin to a given point in the first quadrant, being given also a horizontal displacement. This iterator can be used in the loop below, supposing the declaration of points 'p' and 'q':

```
loop for p <- OriginToGiven ( q, 0.5 );
        statement,...
end loop
```

and the iterator could be declared as follows:

```
iterator OriginToGiven (in Point p, in REAL delta_x);
        REAL theta;
begin
        -- calculate the segment's slope:
        theta := p.y / p.x;
        -- produce the first control object value:
        self.x := 0.0;        self.y := 0.0;
        loop  -- to produce new values:
                yield; -- current value to control object.
                -- prepare new value:
                self.x := self.x + delta_x;
                exit when self.x > p.x;  -- last value already produced.
                self.y := theta * self.x;
        end loop
end iterator
```

Navigators are special coroutines, available in classes modeling dynamic data structures, not to produce new values to control objects, as is the case of iterators, but to

access the next data structure element, following a discipline defined by the structure. The semantics of iterators can be applied to navigators as well. However, since the loop control object in the case of navigators is a structure and not a single element, a generic designator must exist in the loop's body to access the current element.

To exemplify, if we have a class modeling single link lists, and if the object 'pl' is declared as a list of persons, for example, then in the loop below the class navigator FirstToLast could be used to produce, one at a time, the next person in the structure:

```
loop for pl <- FirstToLast ();
        statement,...
end loop
```

Each person in the loop body statements could be referred to by the designator **current**.

CONCLUSIONS

In conventional programming languages, several unit level control structures have been implemented and are commonly used. Exception handlers and iterators are examples of such units. In object oriented languages, however, the only semantics typically available is that of procedures, possibly late bound to receiving objects. In this paper we propose the inclusion of asynchronous message handlers, exception handlers and coroutines in object oriented languages, as operations declared in classes, representing respectively the message passing and receiving, exceptional and iterative behavior of objects. Furthermore, we do this in a way we

hope will be familiar to the professional programmers active today and trying to embrace object orientation.

To achieve this effect we decided that we would bring from the conventional world mechanisms implementing the semantics we required, thus extending object orientation without violating the nature of its main concept, the class, and still providing familiarity. The proposed exception handling and iteration semantics presented above fall in this category. The passing and receiving of asynchronous messages in a general purpose programming language, however, is still coded today typically with the help of primitives more adequate to operating systems. In this respect the inclusion of message handlers as class operations represents a higher abstraction level in the construction of programs.

It can be seen from the syntax and semantics presented above for the operations proposed that a simple class mechanism acts as a more than adequate host for the new operations. In fact the programming language TOOL has already been implemented with the procedural and asynchronous semantics for object behavior. Exception handlers and iterators will be added in the future.

BIBLIOGRAPHY

1. B. Cox, There is a Silver Bullet, BYTE 15 (10): 209-218, 1990.

2. A. Golgberg and D. Robson, SMALLTALK-80: THE LANGUAGE AND ITS IMPLEMENTATION, Addison-Wesley, Palo Alto, CA 1983.

3. G. Krasner and S. Pope, A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk 80, JOOP 1 (3): 26-49, 1988.

4. SPA Sistemas Planejamento e Análise, TOOL: THE LANGUAGE, Rio de Janeiro, 1992.

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

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

7. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, OBJECT ORIENTED MODELING AND DESIGN, Prentice-Hall, Englewood Cliffs, NJ, 1991.

8. G. Booch, OBJECT ORIENTED DESIGN, Benjamin Cummings, Redwood City, CA 1991.

9. P. Coad and E. Yourdon, OBJECT-ORIENTED ANALYSIS, Prentice-Hall, Englewood-Cliffs, New Jersey, 1991.

10. J. Martin and J. Odell, OBJECT ORIENTED ANALYSIS AND DESIGN, Prentice-Hall, Englewood-Cliffs, New Jersey, 1992.

11. S. Shlaer and J. Mellor, OBJECT-ORIENTED SYSTEM ANALYSIS, Prentice-Hall, Englewood-Cliffs, New Jersey, 1988.

12. R. Wirfs-Brock and B. Wilkerson, DESIGNING OBJECT ORIENTED SOFTWARE, Prentice-Hall, Englewood-Cliffs, New Jersey, 1991.

13. P. Wegner, Classification in Object Oriented Systems, SIGPLAN Notices, 21 (10): 1986.

14. C. Petzold, PROGRAMMING WINDOWS, Microsoft Press, Redmond, Wash, 1990.

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

16. Open Software Foundation, OSF/MOTIF STYLE GUIDE, 1990.

17. B. Meyer, OBJECT-ORIENTED SOFTWARE CONSTRUCTION, Prentice-Hall, Englewood Cliffs, NJ, 1988.

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

19. B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. Schaffert, R. Scheifler and A. Snyder, CLU REFERENCE MANUAL, Springer Verlag, Berlin, 1981.

20. C. Ghezzi and M. Jazayeri, PROGRAMMING LANGUAGE CONCEPTS, Wiley, 1982.

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