Nordic Journal of Computing 1(1994), 77–93.

EXTENSIBILITY IN THE OBERON SYSTEM

HANSPETER MÖSSENBÖCK*
Institute for Computer Systems
ETH Zürich
CH–8092 Zürich
Switzerland

Abstract. We show how an object-oriented system-and in particular the Oberon System-can be used to write software that is extensible by end users even while the software is running. Extensibility instead of completeness may be a way out of the unpleasant situation in software industry where applications still tend to become bigger every year. Oberon is both an object-oriented programming language and an operating system with new concepts such as commands and dynamic loading. The language and the system make up an environment that is similar to Smalltalk in its flexibility but offers static type-checking and is much more efficient.

CR Classification: D.2.2, D.1.5

1. Introduction

Ambitious software systems with large functionality and thousands of users can hardly be designed in a form that meets all future needs from the beginning. It is impossible to foresee all the ways in which such systems will be used so that requests for extensions will arise naturally.

Current software industry tries to cope with this problem by including as many features as possible into a software system. This leads to huge monolithic software that offers amazing functionality but still leaves some users disappointed because it typically lacks just their favorite feature.

The opposite approach is to design a system only as a minimal kernel and to provide means to extend it. This allows vendors and even users to add more functionality later without bothering other users which do not need it. It leads to small systems that are easier to handle and still more flexible because the number of future extensions is potentially unrestricted.

The Oberon System [12] follows the latter approach. It supports extensibility both at the language level and at the system level. This paper describes how it can be used to build small systems that can be adapted by users on demand.

* Author's current affiliation: University of Linz, Institute for Computer Science, A–4040 Linz, Austria

Received January 1994. Accepted April 1994.

2. What is Oberon

Oberon is both a programming language and an operating system. It was designed by Niklaus Wirth and Jürg Gutknecht at ETH Zürich and follows the tradition of Pascal and Modula-2. Although the Oberon Language can be used without the Oberon System, its full power only comes to fruition when both are used together. Oberon is small and efficient: the whole operating system including editor and compiler easily fits on a single diskette. The system is available for most common platforms via ftp. The usage, design, and implementation of Oberon are described in [6], [7], [12], and [4].

The most important features of the Oberon language are *modules* with separate compilation, *strong type checking* (even across module boundaries), and *type extension* which makes Oberon an object-oriented language. The most recent language version is Oberon-2 [5] but in the context of this paper we will refer to it as Oberon. In spite of its simplicity Oberon is not an academic language but has all the features that are necessary to build large real-world systems.

The Oberon operating system is an open system consisting of a set of modules sharing the same address space. They can be imported in user-written modules that extend the basic system. There is no difference between a user-written module and a system module. Programming in Oberon therefore always means extending the operating system.

The Oberon run-time system offers facilities for writing extensible software: the most important among them are the so-called *commands* (procedures that can be invoked like programs) and the ability to dynamically add modules to a running system (*dynamic loading*). Oberon has also a *garbage collector* which-together with strong typing-guarantees memory integrity.

Oberon grew out of the experiences with Pascal and Modula-2. Inspired by the Cedar system [9], which Wirth and Gutknecht studied during a sabbatical at Xerox Parc, Oberon was designed and implemented by just two persons from 1985 to 1988. Since then it has been improved and ported to most modern platforms (e.g., SparcStation, DECstation, RS/6000, HP workstations, Intel 386, Macintosh, etc.) [1].

3. Extensibility at the Language Level

A programming language supports software extensibility if it allows adding new operations and new data types to a program without invalidating existing code. A program should not be "aware" when it is extended. Only then is it possible to add new functionality without modifying existing software. This kind of extensibility is provided by object-oriented languages.

Object-oriented programming is based on the following principles: Data and their access operations are combined into units called *classes* (abstract data types). A class T0 can be extended to a class T1 that *inherits* the data and operations from T0 and can add more. The point is that T1 and T0 are compatible, i.e., clients can work with T1 objects instead of T0 objects

without noticing any difference. Operations on objects are performed by sending *messages* to them. A message is a request that tells an object which service is desired but not which procedure is to be invoked in order to implement the service.

Oberon is a hybrid object-oriented language in the sense that there are ordinary types (integers, arrays, etc.) besides classes. Static type checking guarantees that objects will understand messages sent to them. However, it is also possible to send messages that are only checked at run time. Both ways of message sending have their benefits and are explained below.

3.1 Classes and Type-Bound Procedures

In Oberon a class is a record type with associated procedures (*type-bound procedures or methods*). For simplicity we call a pointer to a record type also a class. Let us consider a class *Text* that maintains a sequence of characters with operations such as *Insert* or *Delete*. The interface of such a class may look as follows:

```
TYPE
  Text = POINTER TO TextDesc;
  TextDesc = RECORD
    length: INTEGER;
    PROCEDURE (t: Text) Insert (pos: INTEGER; s: ARRAY OF CHAR);
    PROCEDURE (t: Text) Delete (from, to: INTEGER);
    ...
  END;
```

Note that the interface hides the actual implementation of the text. Only the length is visible; other data fields are hidden. This prevents clients from directly accessing the data possibly invalidating certain invariants. It also allows changing the implementation of the data without requiring modifications in the clients.

To insert characters in a text *t* one can write *t.Insert(0, "abc")*. The compiler knows that *t* is of type *Text* and can therefore check that *Insert* is indeed a method of *TextDesc*. A message *t.Print* would result in a compile-time error since *Print* is not a method of *TextDesc*.

The object to which a message is sent is called the *receiver*. It is a parameter of every method. To distinguish it from other parameters it is written in front of the method's name. In contrast to other object-oriented languages, where the receiver is usually predeclared with a special name such as *self*, Oberon requires an explicit declaration. This avoids hidden mechanisms and allows choosing a meaningful name for the receiver.

The interface of a class is an extract of its actual implementation which is contained in a module. While the interface lists only the fields and methods visible to clients, the implementation gives their full declarations inluding the code of the methods. The implementation of the class *Text* may look as follows:

```
MODULE Texts;

  TYPE
    Text* = POINTER TO TextDesc;
    TextDesc* = RECORD
      length*: INTEGER;
      buffer: ...
    END;

  PROCEDURE (t: Text) Insert* (pos: INTEGER; s: ARRAY OF CHAR);
  BEGIN
    ... code for this method ...
  END Insert;

  PROCEDURE (t: Text) SetInsertPoint (pos: INTEGER);
  BEGIN
    ... code for this method ...
  END SetInsertPoint;

  ... other methods ...

END Texts.
```

Names that are to be exported are marked with an asterisk in their declaration. Thus the field *length* is visible in client modules while the field *buffer* is not. Similarly the method *Insert* is exported but the method *SetInsertPoint* is not. Note that the interface of a class is not written down explicitly but is extracted from the implementation by a browser. This avoids the need for a special definition module and keeps redundancy small.

Now let us assume that we want to have texts that maintain not only characters but also fonts. We can derive such a class *StyledText* from *Text* by writing:

```
TYPE
  StyledText = POINTER TO StyledTextDesc;
  StyledTextDesc = RECORD (TextDesc)
    fonts: FontList;
    PROCEDURE (t: StyledText) SetFont (from, to: INTEGER; font: Font);
    ...
  END;
```

Writing the name *TextDesc* behind the keyword RECORD means that *StyledTextDesc* extends *TextDesc*, i.e., it inherits all the fields and methods from *TextDesc* just as if they were redeclared at this point explicitly. In addition to that, new fields and methods such as *fonts* and *SetFont* may be declared. Since the record *StyledTextDesc* extends *TextDesc* we also say that the pointer *StyledText* extends *Text*. *StyledText* is called a *subclass* of *Text* and *Text* a base class of *StyledText*.

Oberon supports single inheritance, i.e., a class may be derived only from a single base class. Other languages allow multiple inheritance where a class can be derived from several base classes. We did not find many convincing examples for the usefulness of multiple inheritance. It rather tends to complicate class hierarchies and sometimes imposes a run-time penalty on programs even if it is not used [8]. In many cases multiple inheritance can be modelled reasonably with single inheritance as is shown in [11].

StyledText inherits also the methods *Insert* and *Delete* from *Text*. However, these methods only work on plain characters and not on fonts so they have to be overridden in *StyledText*. This can be done by redeclaring them in *StyledText* with the same signature but a different implementation. For example, *Insert* may be redeclared as follows:

```
PROCEDURE (t: StyledText) Insert* (pos: INTEGER; s: ARRAY OF CHAR);
BEGIN
  t.Insert^(pos,s); (*call the inherited method fromText*)
  ... update the font information of t ...
END Insert;
```

Note that it is possible to call the inherited method from within the overriding method. This allows building on existing code and avoids rewriting everything from scratch.

Since *StyledText* was derived from *Text* it is a special kind of *Text*. Therefore it is possible to assign a *StyledText* object to a *Text* variable *t*. If we now send an insert message to *t*, e.g., *t.Insert(0, "abc")*, the *Insert* method from *StyledText* will be invoked and not the one from *Text* as above. This is correct because *t* now holds a *StyledText* object. This mechanism is called *dynamic binding*: a message always invokes the method from that object that is stored in the receiver at run time.

What is this all good for? The big benefit is that all programs that were written to work with *Texts* can now also work with *StyledTexts* without having to be modified. By simply storing *StyledText* objects in *Text* variables all messages to these variables will invoke *StyledText* methods. We may even decide to derive more types from *Text* such as *StructuredText* or *HyperText*. The existing programs can work with all these types without distinguishing between them. In other words: they can treat all variants uniformly. This is something which is hard to do in non-object-oriented languages.

3.2 Message Records

Methods are just one possibility to implement messages. Another possibility is to take the term "sending a message" literally, and to regard a message is a piece of data (a record) that is passed to an object for being handled. This requires one record type per message and one method in every object

to handle the messages. Since this handler must accept various messages all message records must be derived from a common base type:

```
TYPE
  Message = RECORD END; (*common base type for all messages*)

  InsertMsg = RECORD (Message) pos: INTEGER; s: ARRAY 256 OF CHAR END;
  DeleteMsg = RECORD (Message) from, to: INTEGER END;
```

The parameters of the methods become fields of the message records. The interface of *Text* may now look as follows:

```
TYPE
  Text = POINTER TO TextDesc;
  TextDesc = RECORD
    length: INTEGER;
    PROCEDURE (t: Text) Handle (VAR m: Message);
  END;
```

Sending a message becomes:

```
VAR insert: InsertMsg;
...
insert.pos := 0; insert.s := "abc";
t.Handle(insert);
```

The implementation of *Handle* must inspect the run-time type of the received message and react according to it. Oberon provides a WITH statement that performs such a test:

```
PROCEDURE (t: Text) Handle (VAR m: Message);
BEGIN
  WITH
    m: InsertMsg DO ... (*insert m.s at m.pos into t*)
  | m: DeleteMsg DO ... (*delete m.from..m.to from t*)
  | ...
  ELSE ... (*unknown message*)
  END
END Handle;
```

If m holds an *InsertMsg* at run time the first branch of the WITH statement is executed and m is regarded as an *InsertMsg* variable; if m holds a *DeleteMsg* at run time the second branch is executed, and so on. If none of the type tests is satisfied the ELSE branch is executed.

Message records have several advantages over methods:

- Message records are pieces of data that can be stored and sent later on. In principle they can even be sent over a network to a distant computer.
- The same message can easily be distributed to more than one object. This can be used to implement *broadcasts*.
- An object can be sent a message that it does not understand. It may ignore the message, forward it to another object, or report an error. For example, a *SetFont* message may be broadcast to several texts although only *StyledTexts* will understand it. It is sometimes simpler and more flexible to send the message to all objects and let the objects decide if they want to react. With methods this is not possible because the compiler checks if a message is understood by the receiver.
- It is possible to implement the message handler as a procedure variable rather than as a method. Then the handler can be exchanged at run time to dynamically exchange the behavior of an object. With this approach an object may assume different roles during its lifetime.

Message records also have some disadvantages:

- It is not immediately clear which operations belong to a class, i.e., which messages an object understands. To find that out, one has to look at the implementation of the message handler.
- Message records are interpreted at run time using a WITH statement that checks the variants sequentially. This is slower than a method invocation, which is usually implemented as a table look up.
- Sending a message record is somewhat clumsy. First the input parameters have to be packed into the record, then the handler has to be called, and finally output parameters can be obtained from the record.
- What was considered an advantage above can also be a drawback: the compiler cannot check if an object understands a message. If a message is not understood this might lead to a run time error. The error can show up only after months and is difficult to find then.

Thus message records have advantages and disadvantages. In general, methods are preferable because they are more efficient, safer, and better readable. Message records should be used where special flexibility is needed, e.g., for broadcasting a message or for cases where it is important to add new messages to a class later without changing the interface of that class.

4. Extensibility at the System Level

How can an operating system contribute to the extensibility of programs? First, it must provide means to add new modules with new code and new data types to already loaded programs. The new modules should share the same address space with the already loaded modules so that they can access

their data and call their procedures. Moreover, it must be possible for the new modules to install procedures and objects in the old modules in order to get the new code invoked by the old system (up-calls).

The Oberon System provides these features together with a couple of facilities such as *commands*, *run-time types*, and *garbage collection* that help in writing extensible programs.

4.1 Prerequisites

Commands

In common operating systems like Unix or MS-DOS the unit of execution is a program. The user first has to invoke a program and can then perform further actions by selecting from menus (mode-less input) or by responding to prompts (modal input).

In Oberon the unit of execution is a procedure. Every parameter-less *procedure* can be invoked interactively just like a program. If a module M exports a procedure P, the user can activate it as $M.P$. In most Oberon implementations this is done by typing $M.P$ and clicking at it. In fact, every occurrence of the text $M.P$ on the screen can be used to activate the procedure. Such procedures are called *commands*.

What happens when a command $M.P$ is called? First, module M is loaded (if not already in memory) and linked to the already loaded system. Then procedure P is looked up in a dictionary of commands associated with M and is executed. When P terminates, module M stays loaded with all its global data. A later activation of $M.P$ (or of any other command of M) can still access this data to read or modify it.

Commands therefore allow writing programs with multiple entry points. Implementing menus becomes unnecessary, since every action of the system can be *directly* invoked as a command. Modules assume the role of data capsules that maintain data which can be manipulated by commands in an arbitrary order. Commands can exchange information via memory data structures and do not have to resort to files or channels. This promotes efficiency and convenience.

As an example consider a module *Counter* that maintains a numeric value and provides two commands *Add* and *Print* to modify and inspect it.

```
MODULE Counter;

  VAR val: INTEGER;

  PROCEDURE Add*;
    VAR x: INTEGER;
  BEGIN
    ... (*read a value x*)
    val := val + x
  END Add;
```

```
  PROCEDURE Print*;
  BEGIN
    ... (*print val*)
  END Value;

BEGIN
  val := 0
END Counter.
```

Since a command is a parameter-less procedure there must be some way to pass arguments to it. A command is free to expect arguments from anywhere. Usually the arguments follow the command in the text at the point of its invocation, but they could also come from a different window or from a file. A user could now type

```
Counter.Add 10
```

and click at it. *Counter* will be loaded, its body will be executed, and *val* will be initialized to 0. *Next, Add* will be called, setting *val* to 10. The user may now click at the above command again, causing *val* to become 20, 30, and so on. Finally the user may decide to print the value by invoking

```
Counter.Print
```

Note the similarity between modules with commands and object-oriented programming. A module corresponds to an object with local state. Invoking a command of the module corresponds to sending a message to the module "object". This model of user interaction is known from interpreted languages like Smalltalk and Lisp but it is usually not available in compiled languages like C++ or Pascal.

Commands are particularly useful for adding new functionality to an existing system. This will be explained in Section 4.2.

Dynamic Loading and Linking

In many operating systems, a *linker* is used to put all modules of a system together, resolve external references, and write the result to a file that can be loaded by the *loader*. As soon as the system has been linked, however, it is a monolithic piece to which nothing can be added any more. It is not extensible.

Recently, operating systems have relaxed this limitation by the concept of *shared* or *dynamic link libraries*. These libraries allow modules to be added to a linked and even loaded program. In Oberon, this concept is called *dynamic loading* and is particularly straight-forward, since no special libraries are needed. Every module can be added at run time.

Oberon uses a *linking loader*. There are no pre-linked files on the disk but every module is compiled to a separate object file. When a command from a module M is invoked, M's object file is loaded and linked with all modules imported by M. If some of these modules are not yet in memory, they get loaded, too.

Fig. 1: Dynamic loading of modules

Fig. 2: Every module is loaded only once

After system startup, the loaded modules are just those of the operating system. Then, while commands are activated, new modules are possibly added. After some time, all modules needed for a typical session will be in memory and the system will reflect the user's work set. Command execution becomes very fast then, since no new modules have to be loaded any more (Figure 1).

Because all modules share the same address space, every module is loaded only once. Thus, if a system consists of several modules, only those are loaded that are not already in memory (Figure 2).

Dynamic loading is an indispensable prerequisite for extensibility. It results in short loading times and little memory consumption and soon leads to a situation in which no new modules have to be loaded any more. The system has been adapted to the user's needs.

Instead of invoking a command interactively, it can also be invoked from a module via a library procedure (*Oberon.Call* in most implementations). This allows the system to extend itself. The following example shows how this works: A module *Base* may read the name of a command—*Ext.P*, say—into a variable *cmd* and invoke *Oberon.Call(cmd, ...).* This will add module *Ext* as described above. *Ext* may now install a private object in a variable *obj* of module *Base*. When *Base* sends a message to *obj*, dynamic binding will cause the corresponding method in module *Ext* to be executed (Figure 3).

Fig. 3: A module *Base* loads a module *Ext* that it does not know

Note, that *Base* is invoking code of which it does not even know that it exists. In fact, *Ext* may have been written long after *Base* was designed and implemented.

A practical example of a self-extending program is a module *Base* that reads a data file containing objects of a yet unloaded module *Ext*. In order to be able to apply operations to these objects, *Ext* must be loaded. To do so, the data file contains also the name of a command from *Ext* that is read and invoked by *Base*. Thus *Ext* is loaded and is extending *Base*.

Garbage collection

Garbage collection is a generally useful mechanism that automatically reclaims unreferenced memory and relieves a programmer from explicitly deallocating data structures. Explicit deallocation is dangerous since it may lead to dangling pointers which are a common source of errors and hard to track down.

In an extensible system, garbage collection is even more important than in a static system because programs may work with object extensions which they do not know. Therefore, if a data structure is deallocated, it may still be referenced by an invisible pointer from some object extension. This pointer then becomes dangling.

The garbage collector in the Oberon System—together with the strong typing concept of the language—ensures that a pointer is either NIL or points to a valid object. Memory integrity is therefore guaranteed, even in the case of extensions.

4.2 Adding New Commands to a Program

So far, we have discussed some prerequisites for extensibility: object-orientedness, commands, dynamic loading, and garbage collection. Let us now look at an example that shows how commands can be used to add new functionality to a program.

Assume that we have a text editor and want to add a spelling checker to it. The spell-checking algorithm is straight-forward: read every word of the text and look it up in a dictionary. If the word is not found it is assumed to be misspelled. But how can we add this functionality to the existing editor?

In Oberon, we can add functionality in the form of a new module and activate it by a command of this module. Therefore we design a module *Spell* with a command *Check*.

What do we have to do in this command? First, we have to get the text from the desired window (called *viewer*). The Oberon System provides procedures to do that. Since all modules share the same address space, our new module can access the text of the editor and read it. For reading, the Oberon System provides special *scanners* that can be set to a text and used for reading symbols like identifiers or numbers. Finally, we have to implement the dictionary and its look up mechanism. A sketch of module *Spell* might look as follows:

```
MODULE Spell;
  IMPORT Oberon, Viewers, Texts, TextFrames;

  ... declaration and implementation of the dictionary ...

  PROCEDURE Check*;
    VAR viewer: Viewers.Viewer; frame: TextFrames.Frame;
    text: Texts.Text; scanner: Texts.Scanner;
  BEGIN
    viewer := Oberon.MarkedViewer();          (*get the marked viewer*)
    frame := viewer.dsc.next(TextFrames.Frame); (*get the text frame in
                                               this viewer*)
    text := frame.text;                       (*get the text from this frame*)
    Texts.OpenScanner(scanner, text, 0);      (*set a scanner to the
                                               beginning of the text*)
    Texts.Scan(scanner);
    WHILE ~ scanner.eot DO                    (*read words and look them up*)
      IF scanner.class = Texts.Name THEN
        ...                                   (*a word has been read and
                                               stored in scanner.s;*)
```

```
        ...                                    (*look it up in the dictionary*)
      END;
      Texts.Scan(scanner)
    END
  END Check;

  ... other commands ...
END Spell.
```

The user may now mark a text viewer and invoke the command *Spell.Check*. The module *Spell* will be loaded and added to the editor. The *Check* command will look up every word of the text in the dictionary. Our new module will probably contain further commands such as *SelectDictionary* to select one of several dictionaries, *LearnWord* to add a new word to the dictionary, or *ForgetWord* to remove a word from the dictionary. All these commands can be invoked individually and in any order. They share the dictionary as a common data structure.

Note that the spell-checking facility need not be loaded from the beginning. The editor can be started as a small and simple program. This makes its loading time short and provides the user with an easily comprehensible system. Spell-checking is only loaded on demand. Not every user has to carry it along and pay for functionality that he possibly never uses. In the same way, other functionality such as regular expression search, word counting, or special formatting can be added in the form of new modules with commands.

Compare this to the usual way an application is delivered. Commercial products are usually huge packages several megabytes in size which contain every conceivable feature. Yet they can never be complete and since they are not extensible they are of limited value to some clients.

4.3 Adding New Objects to a Program

We will now look at another example of extensibility. This time we want to add new objects to a program, i.e., we want to make the program work with data types that it did not know before. As the reader might guess, this is accomplished with object-oriented techniques.

Let us stay with the example of the text editor. Assume that we want to have pictures, tables, formulas, or clocks in the text (Figure 4). The editor should handle all these objects uniformly, i.e., it should display them on the screen, load and store them on a file, and allow mouse clicks on them. Every object should handle mouse clicks in its own way: a picture should react by interpreting the click as an editing request while a clock should allow setting the time, etc.

The objects are like big characters. They float with the text while it is edited and can be selected and deleted much like ordinary text. An important requirement is that it should be possible to add new kinds of objects such as hypertext buttons or pop-up menus later. These objects

Fig. 4: Text with objects such as pictures, tables, or clocks

should behave exactly like the existing ones. They should be displayable, loadable and storable, and they should react to mouse clicks.

How can we plan for such extensions? The only way for the editor to cope with future object variants is to not distinguish between variants at all. Objects are treated as abstract entities. The editor knows what operations can be applied to them (e.g., displaying, loading, storing, etc.) but not how these operations work. Every object is itself responsible for implementing them. We therefore define an abstract base class *Element* from which we later derive all the specific classes such as pictures, tables, and formulas.

```
TYPE
  Element = POINTER TO ElemDesc;
  ElemDesc = RECORD
    width, height: INTEGER;
    PROCEDURE (e: Element) Display (frame: Display.Frame);
    PROCEDURE (e: Element) Copy (VAR copy: Element);
    PROCEDURE (e: Element) HandleMouse (x, y: INTEGER; buttons: SET);
    PROCEDURE (e: Element) Load (VAR r: Files.Rider);
    PROCEDURE (e: Element) Store (VAR r: Files.Rider);
  END;
```

Fig. 5: A piece list with 3 text pieces and 2 element pieces

Class *Element* defines the fields and operations common to all elements but not more. The methods are still empty because we cannot say how an abstract element should be displayed or stored.

The specific elements are defined as subclasses of *Element*. They may add new data fields and must override the inherited methods. A subclass for graphic elements might look as follows:

```
TYPE
  GraphicElement = POINTER TO GraphicElemDesc;
  GraphicElemDesc = RECORD (ElemDesc)
    figures: ...
    PROCEDURE (e: GraphicElement) Display (frame: Display.Frame);
    PROCEDURE (e: GraphicElement) Copy (VAR copy: Element);
    PROCEDURE (e: GraphicElement) HandleMouse
        (x, y: INTEGER; buttons: SET);
    PROCEDURE (e: GraphicElement) Load (VAR r: Files.Rider);
    PROCEDURE (e: GraphicElement) Store (VAR r: Files.Rider);
  END;
```

The text in the editor is organized as a so-called *piece list* [3]. A piece denotes either a stretch of characters or an element. Figure 5 shows the data structure for a text containing two elements. Since graphic elements and clock elements are subclasses of abstract elements they can be stored in fields which are of type *Element*.

The editor may now forget about the specific element kinds. Whenever it wants to redraw the text it traverses the piece list and sends every element a *Display* message. Graphic elements will react by drawing a picture and clock elements by drawing a clock.

But how can we add new elements like hypertext buttons? We have to provide a new module *M* containing the declaration of a *class HyperElement* which is a subclass of *Element*. We also have to write a command *Insert* that generates a new hypertext element and inserts it in the piece list using a special procedure of the editor.

```
MODULE M;
  IMPORT Texts;

  TYPE
    HyperElement = POINTER TO HyperElemDesc;
    HyperElemDesc = RECORD (Texts.ElemDesc)
      ...
    END;

  PROCEDURE Insert*;
    VAR e: HyperElement; t: Texts.Text;
  BEGIN
    NEW(e);
    ... initialize e ...
    ... t := text in which the element is to be inserted ...
    ... insert e into t ...
  END Insert;

END M.
```

Executing *M.Insert* will now insert a hypertext button in the text. The editor is able to handle this new kind of element like all the others although it does not know about the existence of hypertext elements.

The extensibility in this example comes from the following characteristics of the Oberon System:

(1) The object-oriented nature of the language allows us to extend the type *Element* and to treat the extensions in the same way as their base type.

(2) Dynamic loading makes it possible to add the new hypertext module to the editor at run time.

(3) Commands allow us to invoke the new functionality directly without having to modify the menus in the existing editor.

(4) Since all modules share the same address space, the hypertext module can insert elements in the piece list which belongs to a module of the base system.

Note that this kind of extensibility has to be planned right from the beginning. The base system must contain slots into which the extensions can be plugged. Such a slot is the abstract class *Element*. Although the editor does not have to know which element kinds will exist in the future, it must know that there are such things like elements at all. It has to specify the operations that can be applied to them and has to provide means to store elements in the base system. Thus, extensibility does not come from an object-oriented language automatically. It has to be planned.

An editor of this kind has been implemented in the Oberon System [10] and has proved highly useful. Many users implemented their own element extensions and made them available to others. While the base system remained small and simple the total functionality offered by elements is amazing. The

editor is now even used in situations for which it was not intended (e.g., as a framework for user interfaces).

5. Conclusions

This paper showed how programs can be extended in the Oberon System using its most prominent features: object-orientedness, commands and dynamic loading.

Although all object-oriented languages claim to be suitable for writing extensible software this is often not true because the systems in which they are embedded lack dynamic loading. Some systems provide dynamic loading in the form of dynamic link libraries but still these systems lack commands so that there is no easy way for the user to invoke the extensions interactively. The Smalltalk System [2] is an exception: it allows adding objects to a running system and sending messages to them interactively (which corresponds to invoking commands). Smalltalk is therefore known for its power and flexibility. However, Smalltalk is an interpreted and dynamically typed language which makes it less efficient and less safe. Oberon provides the same flexibility while it is statically typed and efficient.

References

[1] Brandis M., Crelier R., Franz M., Templ J.: The Oberon System Family. Technical Report 174, Department of Computer Science, ETH Zürich, 1992.
[2] Goldberg A., Robson D.: Smalltalk-80 - The Language and its Implementation. Addison-Wesley, 1983.
[3] Gutknecht J.: Concepts of the Text Editor Lara. Communications of the ACM, Vol. 28, No. 9, Sept. 1985, 942-960.
[4] Mössenböck H.: Object-Oriented Programming in Oberon-2. Springer-Verlag, 1993.
[5] Mössenböck H., Wirth N.: The Programming Language Oberon-2. Structured Programming, Vol. 12, No. 4, 1991.
[6] Reiser M.: The Oberon System. User Guide and Programmer's Manual. Addison-Wesley, 1991.
[7] Reiser M., Wirth N.: Programming in Oberon. Steps beyond Pascal and Modula-2. Addison-Wesley, 1992.
[8] Stroustrup B.: Multiple Inheritance for C++. Proceedings of the EUUG Spring Conference, Helsinki, May 1989.
[9] Swinehart D.C., et al.: A Structural View of the Cedar Programming Environment. ACM Transactions on Programming Languages and Systems, Vol. 8, No. 4, October 1986, 419-490
[10] Szyperski C.A.: Write-ing Applications: Designing an Extensible Text Editor as an Application Framework. Proceedings of the TOOLS'92 Conference, Dortmund, Prentice Hall, 1992.
[11] Templ J.: A Systematic Approach to Multiple Inheritance Implementation. SIGPLAN Notices, Vol.28, No.4, April 1993, 61-66.
[12] Wirth N., Gutknecht J.: Project Oberon. The Design of an Operating System and Compiler. Addison-Wesley, 1993.