Oberon, Gadgets and Some Archetypal Aspects of Persistent Objects

Jürg Gutknecht, ETH Zentrum, CH–8092 Zürich

February 18, 1996

Abstract

Oberon is both a programming language in the Pascal-Modula tradition and a modern operating system for single-user personal workstations. Its highlights are (a) an integrated and modular programming environment and (b) a versatile textual user interface. Oberon System 3 is an evolution of the original Oberon system that features (c) a built-in management of persistent objects and (d) a sophisticated component framework called *Gadgets* for the interactive and programmed construction of visual objects, model objects and entire graphical user interfaces. Henceforth in this text, we shall use the terms *Oberon* and *Oberon system* synonymously with *Oberon System 3, expressed in the language Oberon.*

The main objective of this article is a systematic presentation of the complete Oberon system architecture whose cornerstones are several well-defined, separate and extensible hierarchies that interact harmoniously. In detail, the topics dealt with in this article are: (a) Modular structures and their relation to class libraries, (b) a suitable kernel support for the management of persistent objects and (c) concepts and structure of the *Gadgets* package. In particular, we make an attempt to classify the different activities in the relatively new and complex field of component construction. In a brief interlude, we comment on a current study of some kind of concurrent objects, called *active* objects.

Keywords: Oberon, System design, Modular systems, Object-oriented languages, Object-oriented systems, Persistent objects, Component frameworks.

1 Oberon as a Hierarchy of Modules

Perhaps the best short characterization defines Oberon as a well-organized hierarchy of *modules*, as depicted in Figure 1. In this context, modules are collections of logically connected types, data and functionality, and intermodular dependencies represent *client-server* relations or, more concretely, use of data or functionality. However, the use of ingredients of a module is restricted to a subset that is presented as an *abstract public interface* or, to put it differently, private contents of a module are inaccessible to clients. The benefits of such an explicit concept of module interface are obvious: (a) Private data structures

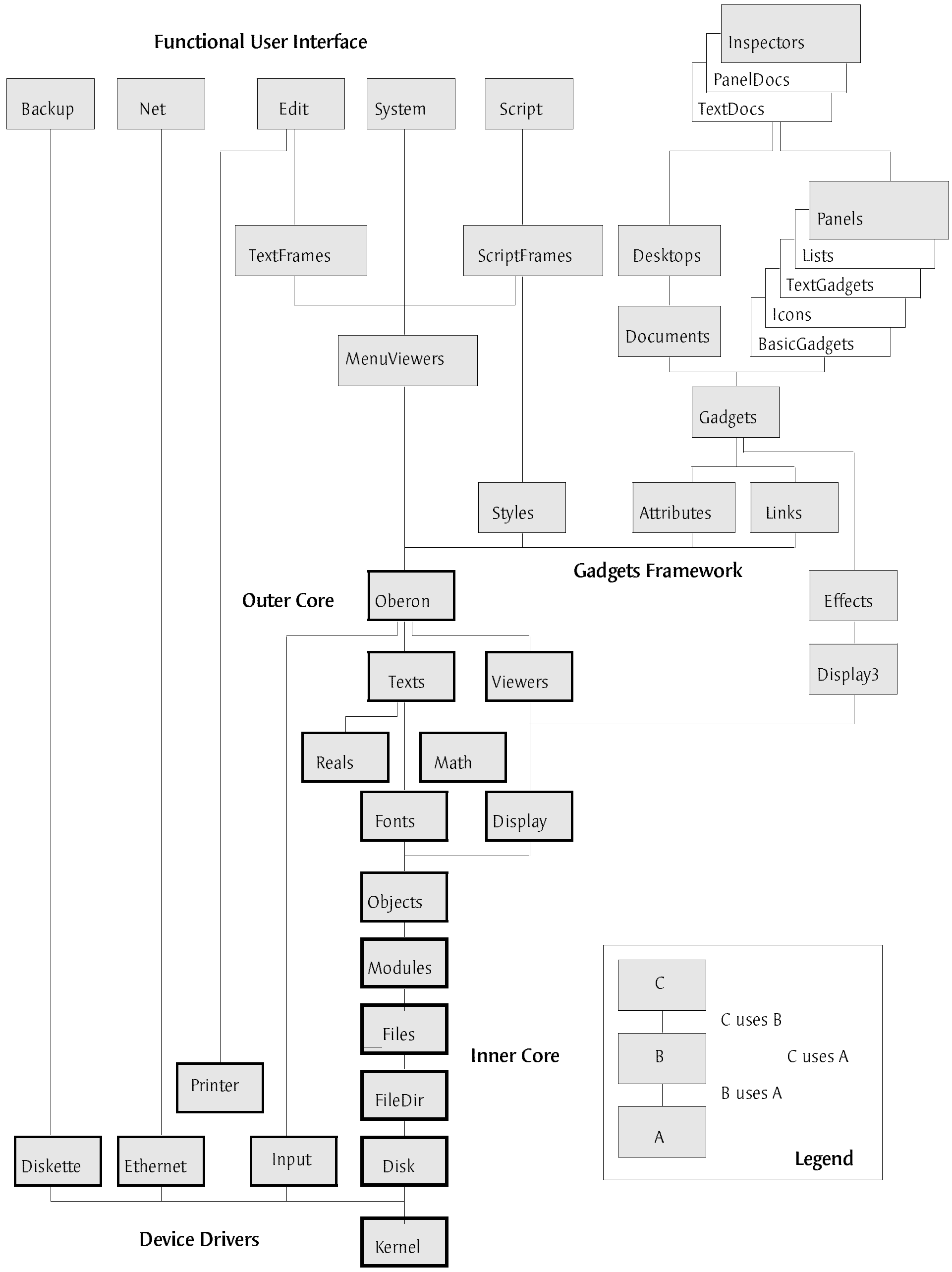


Figure 1: The Oberon System as a Hierarchy of Modules

are protected from corruption by clients and (b) clients are insensitive about changes of private details in server modules.

While most of the modules in the hierarchy are used by other (client) modules, the topmost modules are used by interactive users instead. Their interfaces are collections of *commands* that can be called by name (*M.P*) directly at run time. In their entirety, the topmost modules in the hierarchy therefore represent the *functional user interface* of the system.

In a system with such a rigorous modular structure, individual modules are sensibly declared as both compilation units and loading units. This has two important technical implications. First, remembering the strong typing tradition of the Pascal-Modula-Oberon line, a compiler is needed that does rigorous consistency checking across module boundaries. Second, because programs appear as packages of *dynamically linked libraries* (DLL) much more naturally than as statically linked monoliths, it seems reasonable to use a dynamic linking loader that loads modules on demand, i.e. if and when they are used for the first time.

Taking a global and abstract view, Oberon thus appears as a completely uniform hierarchy of abstract specifications of all participating resources. This is remarkable, because the possible variety of such resources is considerable, as is easily illustrated by some typical examples like hardware devices (for example keyboard, mouse and display screen handled by device driver modules *Input* and *Display* in Figure 1 respectively), abstract data types (for example fonts and texts handled by modules *Fonts* and *Texts*), functional libraries (for example math libraries) and application packages.

The abstract data type *Text* has not been mentioned accidentally. Its presence in the system base can be considered as a keypoint of the Oberon architecture, witnessing the major and integrating role played by texts. Undoubtedly the most profitable application is Oberon's unique and very effective *textual user interface* (TUI) that, in essence, relies on the ability to interactively call commands by their name (M.P) from any location in any text. The crucial technical tool making this possible is an integrated *scanner facility* (concept borrowed from compiler construction) that is able to recognize tokens of a defined set of universal types, including the types *Name* and *Number* that typically occur in textual command specifications and parameter lists.

Text is an excellent example of an *abstract data type*, because its abstract presentation is simple and closed (complete set of operations, including *delete, insert* and *read/write access*), while any efficient implementation is sophisticated and depends on rather complex auxiliary dynamic data structures [1]. However, a similar statement is not true for all object types that are used in an environment like Oberon. For example, the definition in the system base of any type that describes *viewers* (windows) on the screen must remain incomplete and open until the exact kind of contents and functionality of the viewer is known, which, of course, can well be years after system implementation time.

2 Oberon as a Hierarchy of Object Types

It is immediately clear that the above mentioned case of *viewers* is archetypal for truly extensible systems in the sense that existing system layers are often entrusted with the management of objects of a future type (for example specific viewer) on some basic level of abstraction (for example black box). ¿From this, we conclude that a suitable language for the programming of *extensible systems* must necessarily offer some construct for *subtyping*, i.e. for deriving specialized types (for example specific viewer) from a base type (for example black box).

Subtyping is expressed in Oberon in terms of a simple and very natural construct called *type extension*. It allows Oberon programmers to derive new record types from an existing record type by simply adding components. Obviously, such derived types can be considered as specializations or variants of their *base type* and, consequently, objects of a derived type are accepted at run time whereever objects of the corresponding base type are. This kind of type compatibility is known as *polymorphism*. Its safe implementation requires some runtime type support that is not offered by Modula-2, for example.

Subtyping can be considered as a bridge from extensible systems to object-oriented environments. Perhaps to our own surprise, we soon recognize that the new construct of type extension in combination with the old concept of procedure variable provides an absolutely sufficient language framework for the creation of amazingly rich and flexible object-oriented sceneries as the one that we are now going to explore.

Let us start with some sample declarations and an attempt to draw up a small dictionary for the translation from ordinary object-oriented (OO) terminology into Oberon terminology.

Sample declarations

```
TYPE
  Object = POINTER TO ObjDesc;
  ObjDesc = RECORD (* base type *)
    a: A; (* state variable *)
    P: PROCEDURE (me: Object; s: S) (* procedure variable *)
  END;

VAR obj: Object; (* instance *)
```

and (possibly declared in a different module)

```
TYPE
  MyObject = POINTER TO MyObjDesc;
  MyObjDesc = RECORD (ObjDesc) (* derived type *)
    b: B; (* added state variable *)
    Q: PROCEDURE (me: MyObject; t: T) (* added procedure variable *)
  END;

VAR myobj: MyObject; (* instance *)
```

Dictionary

OO Terminology	**Oberon Terminology**	**Oberon Sample**
class	record type	Object
subclass	derived record type	MyObject
superclass	base record type	Object
object	instance of record type	obj, myobj
instance variable	state variable in record	a, b
method	procedure variable in record	P, Q
message send	call of procedure variable	obj.P(obj, s)
		myobj.P(myobj, s)
		myobj.Q(myobj, t)
message	actual parameters	s, t

Dictionary (continued)

OO Terminology	Oberon Terminology	Oberon Sample

object creation	allocation & initialization	NEW(obj);
		obj.P := myP0;
		obj.a := mya
		
		NEW(myobj);
		myobj.P := myP1;
		myobj.a := mya;
		myobj.Q := myQ;
		myobj.b := myb

inheritance	reuse of base type part	myobj.a
supercall	reuse of module functionality	myP0(myobj, s)

Some clarifying and consolidating comments are in order. We first point out that Oberon uses an *instance-centered* approach, i.e. method implementations are instance-specific and bound to objects at creation time. In contrast to *class-centered* systems that require method implementations to be class-wide and specified at programming time, instance-centered approaches are more flexible (method implementations might even be changed during an object's life time), but less economic memorywise (one memory word is used per method and object).

In passing we note that class-wide methods are offered under the name of *type-bound procedures* by a variant of the Oberon language called *Oberon-2* [2]. The problem with the Oberon-2 solution is a stylistic inconsistency originating from the fact that (in contrast to record components) type-bound procedures are overwritable in derived types and thus reintroduce all the problems of method inheritance through the backdoor. In the section on active objects later in this text, we shall revisit type-bound procedures in a different context.

Let us now focus on object interfaces. We first recall that the interface of an object is typically defined by some class-wide and static collection of state variables and methods and is therefore (a) class-centered and (b) closed. While (a) is generally desired, (b) is sometimes too restrictive as, for example, in the case of context-oriented message propagation in composite objects, a situation that we shall encounter later in this text. Remarkably, the Oberon object model is able to type-safely express *open* object interfaces. The key idea is simple: Apply type-extension to *messages*.

For example, let us take the following Oberon declarations:

```
TYPE
  Object = POINTER TO ObjDesc;
  Message = RECORD END; (* message base type *)

  ObjDesc = RECORD
    a: A;
    H: PROCEDURE (obj: Object; VAR msg: Message) (* message handler *)
  END;
```

and (perhaps in a different module)

```
TYPE
  MyMessage = RECORD (Message) (* later defined message type *)
    u: U (* message contents and return values *)
  END;
```

and (perhaps in a different module)

```
TYPE
  YourMessage = RECORD (Message) (* later defined message type *)
    v: V (* message contents and return values *)
  END;
```

Remembering the rules of polymorphic type-compatibility, objects of type *Object* are now prepared in principle to accept and handle messages of type *MyMessage*, *YourMessage* and of any other type that is (at any later time) derived from base type *Message*. In other words, the interface of type *Object* is generic and open.

Almost needless to add that the actual handling of all potentially arriving messages must still be provided by the concrete message handler that is bound to the object at creation time and that, in case of new message types to be handled, the message handler must be extended accordingly. The important point, however, is that such extensions are a pure matter of implementation and do not at all affect or even invalidate clients.

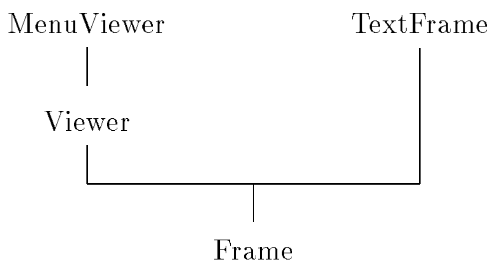
We can now give a sketch of a typical generic message handler. Note that it makes use of *type guards* and *type tests*, i.e. of the runtime type support that we mentioned earlier:

```
PROCEDURE MyHandler (obj: Object; VAR msg: Message);
BEGIN
  IF msg IS MyMessage (* type test *) THEN
    WITH msg: MyMessage (* type guard *) DO (* handle it *) END
  ELSIF msg IS YourMessage THEN
    WITH msg: YourMessage DO (* handle it *) END
  ELSE (* do some default handling *)
  END
END MyHandler;
```

With our next and last comment on Oberon's object-oriented scenery we flash back to modular hierarchies and to the very beginning of our discussion in the previous section. Bearing in mind the new concept of type extension, we are now in a position to refine the relation of intermodular dependency by putting the new relation of specialization/subtyping ("is"relation) beside the conventional client-server relation ("use"-relation). With this, Oberon appears as a hybrid of a traditional modular system (for example based on *Modula-2* or *Ada*) and a purely object-oriented environment (for example *Smalltalk*).

Thinking as specially colored all "is"-dependencies within the total module hierarchy, we can easily recognize a set of disjoint, tree-structured and relatively flat subhierarchies. In contrast to the often deep and complex monolithic class libraries of pure object-oriented systems, our hybrid system thus presents itself as a disentangled two-level structure with a clear distinction between the levels of subtyping and "ordinary" reuse. Also, questionable constructs like "multiple inheritance" are avoided a priori by our concept.

A real and concrete example of a type hierarchy would certainly do no harm at this point. We already mentioned type *Viewer* as a type with an inherent demand for extensibility. Remember that viewers are essentially black boxes on the screen of some arbitrary contents. A first specialization of general viewers in Oberon is called *menu viewers*. By definition, a menu viewer is partitioned into two *frames*, a title/menu frame and an arbitrary frame of contents (see Figure 4). In Oberon, a frame is again a black box on the display screen, in fact the most elementary black box at all. Our collection of types would not be complete without frames of some concrete contents, for example *text frames*.
And this is the hierarchical representation of the set of types just introduced:



Notice that (in this case) an individual module is associated with each type in the diagram. Module *Display* defines type *Frame* as a base type. Module *Viewers* is the viewer manager. It is responsible for the placement of viewers on the tiling screen and keeps track of their exact state, location and extent. Module *MenuViewers* handles menu viewers on an abstract basis and module *TextFrames* implements both a view for texts and a mouse command interpreter for text editing.

Before concluding this section, let us for a moment consider Oberon as a case study in system design and programming in the large. Two widely accepted design principles are (a) unification of similar concepts and (b) separation of different concerns. However, these principles are somewhat controversial, because their application depends crucially on the interpretation of "similar" and "different". Consequently, it is no surprise that different systems differ considerably in the matter of concept unification and separation respectively, as is documented convincingly by the following pairs of concepts: type $\leftrightarrow$ class, specialization $\leftrightarrow$ subclassing and reuse of functionality $\leftrightarrow$ inheritance. In Oberon, the first pair is unified in one concept, while the other two are separated. In C++, the situation is just the other way round.

Pair of concepts	**in C++**	**in Oberon**	**Benefits in Oberon**
type $\leftrightarrow$ class	separated	unified	familiar concepts lean language
specialization $\leftrightarrow$ subclassing	unified	separated	open object interfaces
reuse of functionality $\leftrightarrow$ inheritance	mixed	separated	disentangled hierarchy

Let us recapitulate the benefits resulting from the Oberon solution: (a) Thanks to reuse and reinterpretation of familiar concepts (record type, procedure variable) language extensions can be kept on a minimum, (b) the context-free definition of type-extension (decoupled from classes) as a general tool for specialization comes in handy with the definition of open object interfaces and (c) the clean separation of subtyping and reusing functionality leads to a disentangled two-level hierarchy.

3 Persistent Objects in Oberon

Text is undoubtedly an important data type, and advanced textual user interfaces are remarkably versatile and effective. However, in the age of multimedia and high-speed networks, operating environments are expected to handle persistent documents and objects of an extensive variety. In this context, the term *persistent* is used to indicate that the documents and objects ought to be *portable*, more precisely, transportable from one internal memory to another one or to some external memory (typically a disk).

We first note that, under the condition of full integration or, synonymously, under the condition of unrestricted *object linking and embedding* (OLE), any acceptable basic management of persistent objects must be part of either the programming language or the system kernel. Encouraged by earlier experiments with topics like input/output and concurrency that were successfully removed from the language and put into modules, we decided to delegate the entire management of persistent objects to the system kernel, i.e. not to provide any language or compiler support.

In Oberon, the basic framework of persistent objects is defined by a single module called *Objects*. This module introduces the two abstract concepts *object* and *library (of objects)* that are represented by two base types *Object* and

Library. A library is an indexed collection of objects and is either *public* or *private* (to some host). Public libraries are named and accessible from any authority in the system. The member object O of public library L can be referenced invariantly by a qualified name L.O. As Figure 2 shows symbolically, libraries can refer to each other. In their entirety, they build a hierarchy that, in a sense, is dual to the module hierarchy. Private libraries are anonymous and encapsulated in some higher authority, typically a document. This is also depicted in Figure 2.

Object libraries take a major and very versatile role in the management of persistent objects. Not only do they serve as logical organizers, but they also provide a powerful tool for the crucial tasks of sequentializing (externalizing) and desequentializing (internalizing) of objects and collections of objects. Correspondingly, the functional interface of a library comprises a variety of operations for retrieving, adding and removing objects (at runtime) and for storing and loading the contents of the library to and from a sequential file.

Remember that objects are typically composed (recursively) of components and represented internally as a network of linked nodes. From this, it immediately follows that the algorithms for storing and loading objects must be generic enough to sequentialize and desequentialize any arbitrary dynamically linked heterogeneous data structure. It is therefore interesting to study these algorithms in some detail.

For the sake of simplicity, we assume a simple, full-or-nothing library storing and loading scheme. However, the definition of type *Library* allows different implementations of its functional interface such as, for example, a smarter partial storing and loading strategy in combination with buffering.

The Store library algorithm

This is a two-pass process that relies on a (recursive) preprocessing *binding phase*:

```
Bind(object) = {
* for all components of object do Bind(component) end;
  if object unbound then assign index to object end }

Store(library) = {
  for all objects in library do Bind(current object) end;
  for all indexes in library do
    with object to this index do store generator;
*     store main node with internal links replaced with indexes
    end
  end }
```

The marked statements ("*") cannot be executed directly by a library method, because the internal structure of an object is unknown to the library. Instead, these statements must be object method calls.

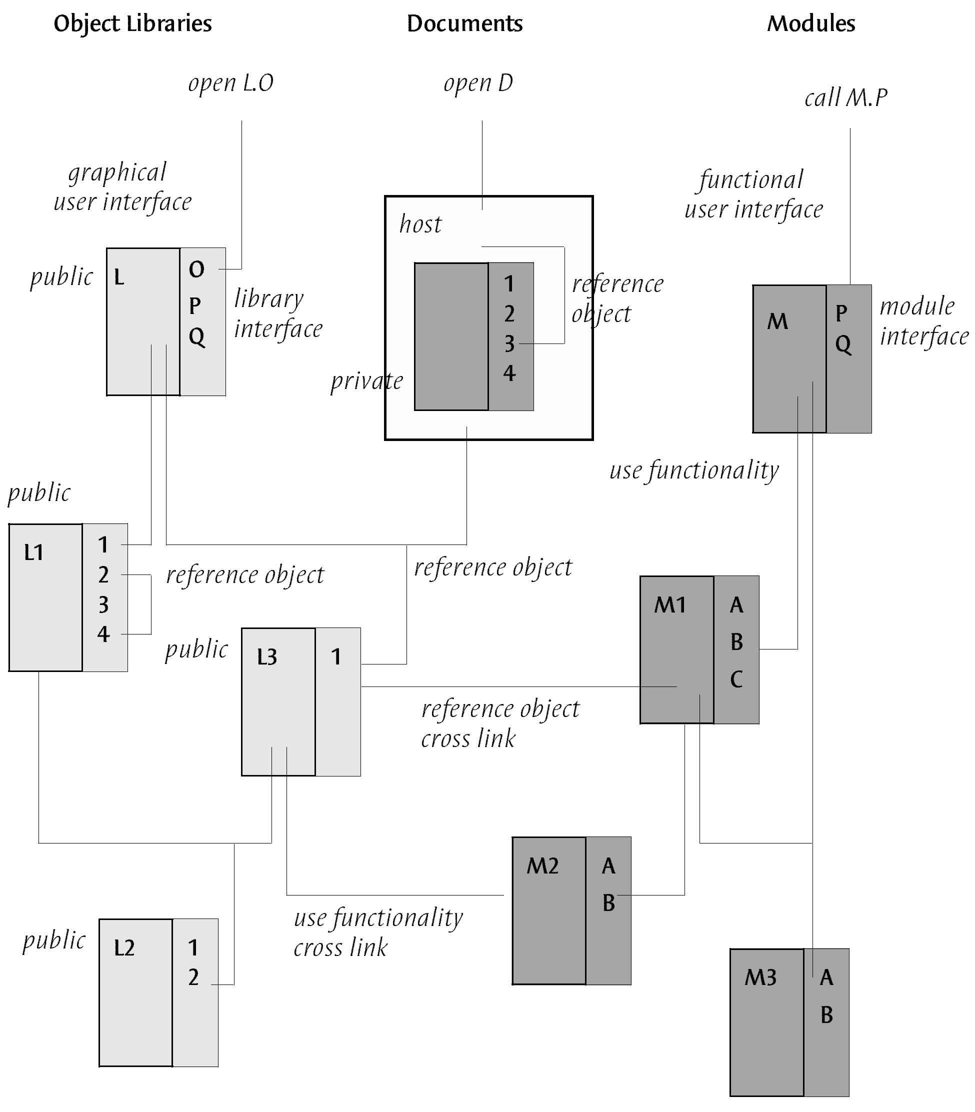


Figure 2: The Dual Hierarchies of Modules and Object Libraries

The Load library algorithm

This is again a two-pass process:

```
Load(library) = {
  for index := 0 to max do load generator; generate main node end;
  for index := 0 to max do
*   load main node with indexes replaced with internal links
  end }
```

Again, the marked statement must be an object method call. Also note that indexes in loaded object nodes might refer to different libraries, so that the loading process can get recursive.

Let us now turn our attention to objects. On the level of their definition, objects are abstract (or "virtual") and have no concrete functionality. Nevertheless, any participating object is expected to be prepared to implement a basic and predefined *system message protocol* that, in a sense, defines exhaustively the abstract concept of *persistent objects* in Oberon. This architecture can be looked at as a software analogy to the familiar *hardware bus*: If they comply with the given bus protocol, participating components of any kind simply plug in.

This is the complete set of basic message types:

{ Bind Message, File Message, Attribute Message, Link Message, Find Message, Copy Message }.

The table on page 13 briefly explains the semantics of each message type and assigns some archetypal topic to it. Note that the first two types are familiar to us already from the above discussion of the *Load* and *Store* algorithms.

So far, the discussion has been rather abstract. However, from the previous section we already know a very important and very concrete class of persistent objects: Frames on the display screen. Frames are *visual objects* because they are assumed to provide some functionality for a visual representation within a rectangular area on the screen (or printer). Typically, frames are *views* of some *model object* and come with a built-in interpreter for interactions, in which cases we can identify frames with the *View-Controller* part of the famous *Model-View-Controller* scheme [3].

¿From a technical view, frames are instances of type *Frame* that is a subtype of the base type *Object*. As specialized persistent objects, frames are expected to obey the basic message protocol plus some extension that is defined by a set of special *frame messages*. This set comprises requests to display itself, to change state (visible $\leftrightarrow$ invisible), size or location, to consume an other object or some text caption, to return selected contents, to mark itself as selected and to update consistency with the underlying model.

In a sense, it is natural to regard the display area itself as one global visual object that is hierarchically composed of ever smaller visual components or, to

Message type	**Topic**	**Explanation**
Bind Message	Grouping	Used to bind objects to a given library. More precisely, if we call *loose* an object that is either unbound or bound to an anonymous library, the bind message requires the primary object and all its loose components to bind themselves.
File Message	Transporting	Used to load and store objects from and to a sequential file.
Attribute Message	Communicating	Object attributes are specified by their name (a string) and their value (typically string or number). Using the *Attribute Message*, attributes can be added, and their value can be retrieved or changed. *Gen* and *Name* are predefined attributes. They specify the object's generator (a procedure) and its intrinsic name respectively.
Link Message	Linking	Used to create and retrieve named links to other objects.
Find Message	Locating	Used to retrieve a component object by its name within the scope of the recipient.
Copy Message	Cloning	Used to create an exact copy ("clone") of the recipient. We distinguish between *shallow copies* and *deep copies*. For shallow copies, as many of the components of the original object as possible are reused (by reference), whereas for deep copies the components are also copied (recursively).

put it differently, it is natural to regard all individual visual objects as mere components of one global *display space*. This way of looking at the situation has some interesting consequences. First, as depicted in Figure 3, it leads to a coherent hierarchical data structure whose first two levels correspond exactly to the typical tiling Oberon display screen with (vertical) *tracks* and (horizontal) *viewers*.

The second consequence is a convention, according to that all messages for visual objects must be addressed primarily to the display space as a whole, with an implicit forwarding obligation. The exact forwarding strategy depends on the kind of message. A *target-oriented strategy* is used if the message is directed to some specific object in the display space (the target), while a *broadcast strategy* is used in cases of an unknown final recipient or an unknown number of final recipients. A typical application of the broadcast strategy is view-update requests by model objects, with the substantial benefit of dispensing models from the burden of knowing about or even registering their views (for example in the form of call-back lists).

The two forwarding strategies are similar in the sense that they are *context-oriented*. However, they are different in detail. While the broadcast strategy simply spreads the message in the display space, the target-oriented forwarding strategy aims at passing down the message along the paths that lead to the desired target object. We should clarify at this point that message forwarding in either case is not a centrally controlled process but is distributed amongst the objects in the display space. However, the extended message protocol (that is compulsory for all members of the display space) defines a set of rules that, in the end, governs the process of message passing.

Let us now take the view of a message travelling through the display space and finally arriving at its destination. We know that, in the moment of its arrival, the message has passed the entire context, step by step. Interestingly enough, we can reap the benefits of this fact in two respects: (a) Any context-oriented processing can be done incrementally and (b) context-dependent message handling is possible. Typical uses of (a) are accumulation of relative coordinates and computation of overlapping masks in the context of a visual object. A typical use of (b) are visual objects with a different behaviour in a developer context and in a user context.

On just a cursory examination we could think that the display space is tree-structured. However, this is not quite correct, if we allow visual objects as models as well, i.e. if we allow views of views. In this case, paths may join, and we can only assert the display hierarchy to be a directed acyclic graph (dag). In combination with a context-oriented forwarding strategy, this may lead to complications due to possibly undetected multiple arrivals of a message at the same object. For example, a *copy* message arriving twice at a shared component of a composite object could lead to the creation of two different copies of this component and could therefore fail. In order to avoid problems of this kind, messages are time-stamped in Oberon, and recipients in the display space are requested to detect multiple arrivals of one and the same message by comparing time-stamps.

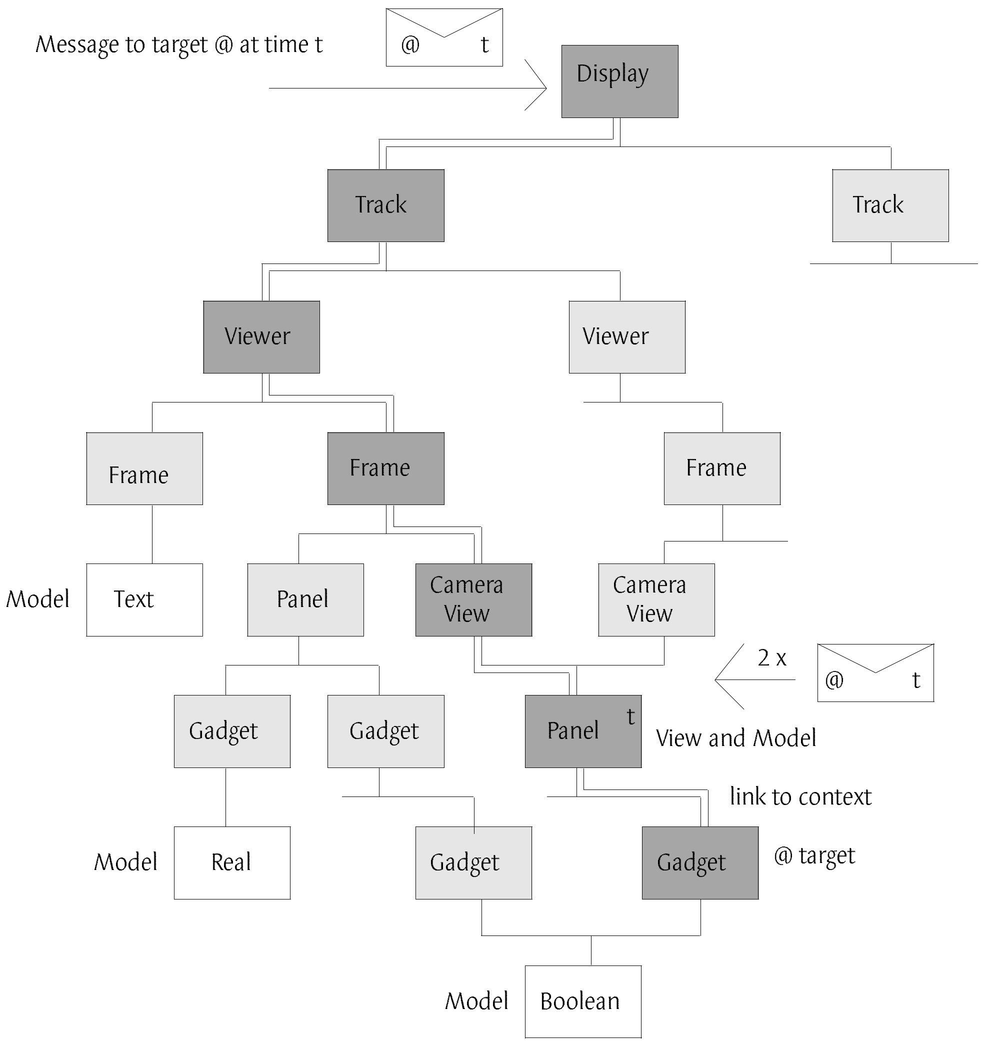


Figure 3: The Global Display Space as a Composite Visual Object

We can best summarize the rules of message handling and message passing in the display space by presenting a rough sketch of a message handler:

```
Handle message M received by frame F = {
  save pointer to context in F; update context pointer in M;
  if timestamp of M > backup timestamp in F then
    save timestamp in F;
    accumulate coordinates in M;
    if target frame of M = F THEN (* target frame is me *) handle M
    else
      if target frame of M = NIL THEN (* broadcast *) handle M end;
      while more descendants do
        pass M to next descendant
      end
    end
  else special handling in new context
  end }
```

In reality, message handling is slightly more intricate because (for the sake of optimization) recipients may decide to (early) terminate the handling of a message and to stop any further propagation.

We now briefly come back to a remark that we made in the previous section on the value of open object interfaces. As we can easily see, any context-oriented forwarding strategy requires open object interfaces. The reason is that intermediate stations on the message paths must be able to pass through (and even to preprocess) messages of a possibly unknown type such as, for example, view-update requests for exotic components.

Our last topic in this section is *object embedding*. We can distinguish two cases: (a) Embedding of objects in objects and (b) embedding of objects in text. Obviously, case (a) is subsumed under Oberon's highly integrated concept of composite objects that culminates in the construct of the display space. For case (b), a different but no less elegant solution exists. To the purpose of its explanation, we first recall Oberon's text model. An Oberon text is a sequence of attributed characters or, slightly simplified, a sequence of pairs (character code, font).

The key idea of our solution is a shift of emphasis in the interpretation of the font attribute. By simply reinterpreting *font* as a collection of characters, we reach a new view of text as a sequence of pairs (character code, collection of characters). It is now a small step from collection of characters to collection of objects and to object libraries. With this, our generalized texts are now sequences of pairs (index, object library), i.e. sequences of general objects or, more precisely, references to objects. Depending on the kind of library, each object is either public (and possibly "contained" in other documents as well) or private to the text. Typical examples of non-character objects (i.e. embedded objects) are pictures, formulae and arbitrary visual objects. However, completely other kinds of embedded objects are conceivable, for example *formatting controls* and *smart links* that are interpretative rather than visual [4].

4 Active Objects

Objects that have occurred so far in this text (and most of the objects in any object-oriented environment) are *passive* in the sense that they are remote controlled by some system process. A better term than passive is *re-active*, suggesting that objects are passive unless they react on an arriving message. However, we actually want *active* objects like videos, moving sprites, animations and simulations, i.e. objects with complete local control of their process of life.

Supposing the support of a sufficiently powerful data protection mechanism, most of the concepts that we have presented so far in connection with persistent objects are adaptable in principle to active objects. It is therefore a tempting idea to extend (smoothly) our system of persistent objects so to include active objects. However, the development in this area has not progressed far yet. For this reason, many details in this section will be omitted intentionally or kept rather vague.

At the moment, we are experimenting with an upgrade of record types towards instantiatable modules that optionally allow *type-local procedures* and a *type-body*. Type-local procedures represent *entries*. Entries can be guarded by some condition (typically a Boolean expression in braces). Calls of guarded entries automatically wait for the guarding condition to be true and then lock the object during execution time for all other clients, i.e. they protect its data from mutual access. Note that type-local procedures are intended to be used for protected access rather than as methods. Therefore, unlike the type-bound procedures in Oberon-2, they are not overwritable in derived types.

If a type-body is present, the object is assumed to be active and controlled by the statement sequence in the body. In this case, an extra light-weight process is created and started for the newly created object as a side-effect of the *NEW* procedure call. Depending on the options (in braces), the object process is given a special priority and is time-sliced or not. Object processes are scheduled centrally by a smart object scheduler in the system kernel.

The following is a rough sketch of a possible szenario, consisting of a group of (passive) resources and a group of concurrent actors. Note that both object types are derived from base type *Kernel.Object*. This reflects the fact that both types make use of the concurrency facilities provided by the system kernel (mutual exclusion in the case of type *Resource* and scheduling in the case of type *Actor* respectively).

```
TYPE
  Resource = POINTER TO ResourceDesc;
  ResourceDesc = RECORD (Kernel.ObjDesc) (* passive object *)
    a: A; b: B; (* local variables *)

    PROCEDURE P (s: S); (* non-guarded entry *)
      VAR ...
    BEGIN ...
    END P;

    PROCEDURE { a > 0 } Q (t: T); (* guarded entry *)
      VAR ...
    BEGIN ...
    END Q;

    PROCEDURE { b < 0 } R (u: U); (* guarded entry *)
      VAR ...
    BEGIN ...
    END R;

    (* passive object *)
  END;

  Actor = POINTER TO ActorDesc;
  ActorDesc = RECORD (Kernel.ObjDesc)
    r: Resource;
    t: T;

    PROCEDURE { systemTime >= t } P; (* guarded entry *)

  BEGIN { p, q } (* active object with priority p and options q *)
    LOOP
      ...
      r.Q(t); (* use resource r *)
      ...
      P; (* suspend myself until time t *)
      ...
    END
  END;
```

5 The Gadgets Component Framework

However indispensible a well-established low-level support in an integrated object-oriented environment may be, as useless it is without the support of some high-level companion. In fact, two companions are needed, one for the assistance of users and one for the assistance of programmers. In Oberon, the *Gadgets* package serves both purposes simultaneously. Looked at it functionally, *Gadgets* is a powerful object toolkit and application framework for the

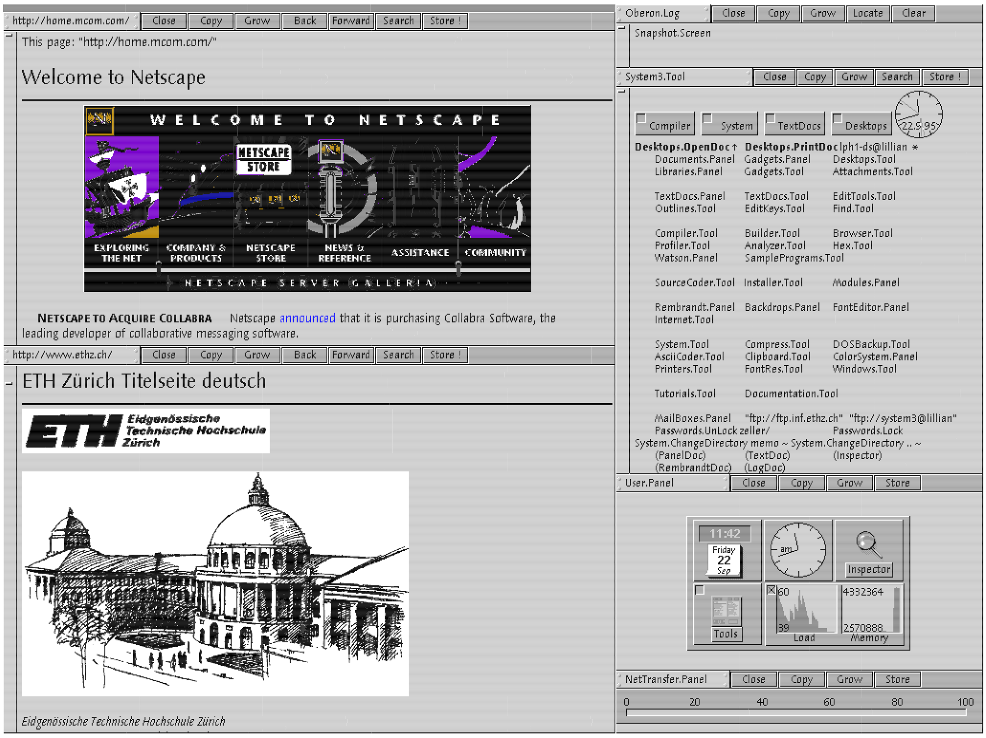


Figure 4: Documents on the Tiling Screen

construction and programming of *graphical user-interfaces* (GUI). Looked at it structurally, *Gadgets* consists of a *Gadgets tool*, an *Inspector tool* and a *library of service modules* (see Figure 1).

Figures 4 and 5 show Oberon *desktops* that are laid out with specially styled visual objects called *gadgets*. Gadgets come in great variety, ranging from simple elements like buttons, checkboxes, sliders, text fields, lists, icons etc. to more complex entities like pictures, line graphics, control panels, texts and entire desktops. In addition, there exist non-visual model gadgets like *Boolean, Integer, Real* etc.. Note that some of the gadgets feature a title bar with an integrated name plate and some buttons. They are called *documents* and are considered as autonomous entities that can be stored under their name and reloaded in an arbitrary context. The desktop itself is a document as well, which demonstrates that documents may (recursively) contain other documents as elements.

The *Gadgets tool* is used to create and compose gadgets interactively. As shown in Figure 5, it is itself a gadget (again a document) that contains two lists and some buttons. The lists expose an extensive collection of predefined visual and model gadgets respectively. In addition, the *Gadgets* tool provides other useful support for the interactive construction, such as automatic alignment in regularly laid-out panels and view-model connections with built-in consistency. For example, a text field and a slider could be connected to one and the same

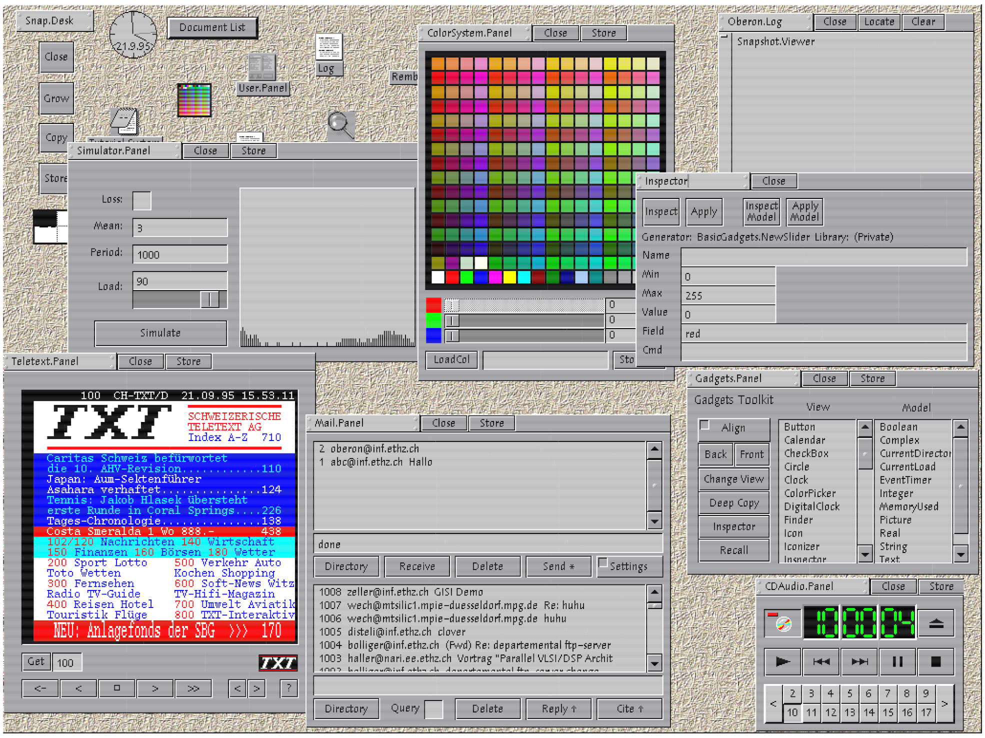


Figure 5: Gadgets Layout on a Desktop

Real type model, or three sliders *red, green, blue* could be connected to a *Color* model.

The *Inspector tool* is also shown in Figure 5. This is a very versatile instrument that can be applied to any gadget (visual or model) for an inspection of its identity, attributes and properties. When applied to a specific gadget, the tool immediately adjusts its shape, so to represent an attribute form for this gadget. Note that attribute forms are again documents, this time created by program, however.

In some cases (such as in the case of adaptive attribute forms just discussed), the interactive method for the construction of gadgets is inapplicable or at least inappropriate, and a programmed approach would be preferable. In the current state of the *Gadgets* package, construction by programming is possible but is not particularly convenient. A much better solution consisting of a suitable layout description language and a corresponding interpreter is planned for the future.

One of the most powerful gadget attributes is the *command attribute*. It is used to connect an Oberon command of the form M.P to a previously neutral gadget, for example to a push-button or a list. Once connected to the gadget, the command is executed implicitly with every user command-action, for example with pushing the button or clicking the mouse at a list element.

Of course, only the most primitive commands need no parameters at all.

Typically, the result of a command at least depends on an entry in a text field or in a list, on the state of a checkbox or on the position of a slider. For such simple cases, a built-in interpreter is provided that is able to retrieve dynamically the value of a specified attribute from a specified gadget (by name).

However, there are more complicated cases. Take an electronic phonebook that is represented by a form containing text fields for name, address, trade and phone number and a button for starting a search action. Obviously, different primary search keys lead to different types of query. The panel therefore needs some built-in heuristics to find out the desired query from the constellation of filled-out fields in the form. For example, if a phone-number is specified, a phone-number query should result, independent of the other entries or else, if a name is specified a name-address query should result etc..

In cases like the electronic phonebook, we cannot get by without any programming at all. A new command is needed that must later be bound to the search button. This command must implement the desired heuristics and in particular, it must be able to identify the different fields in the form and to get their contents. We emphasize that this kind of programming is conventional (i.e. procedural) and well supported by the *Gadgets* module library. Typically, *Gadgets* library modules provide service procedures that hide the entire message handling.

Although the arsenal of predefined gadgets is remarkably extensive, it can by no means satisfy all possible desires. For example, gadgets like the bar diagram in the queue simulator or the color map in the color tool in Figure 5 don't exist a priori. Consequently, there must be a way to program customized gadgets. Remembering that gadgets are finally persistent objects in the sense of the previous section, we already know their program structure in principle. However complicated a gadget program may be in detail, its core is always a message handler that implements the basic or extended system message protocol. It is for this reason that a skeleton implementation serves well as a generic template. Additional programming assistance is provided by the *Gadgets* module library in the form of standard message handlers for both visual and model gadgets.

The complexity of self-programmed visual gadgets is quite essentially determined by their structure. *Non-atomic* gadgets (also called *container gadgets*) like control panels are an order of magnitude more complex than *atomic* gadgets like buttons, lists and bar diagrams. This is not surprising, because container gadgets must be able to manage component objects of any arbitrary type. Their message handlers must properly implement message propagation to constituents and, in addition, must be prepared for feedback requests by constituents (for example, if a constituent is requested to expand).

We summarize this section with a classification of activities in connection with gadget construction. Essentially, we can identify the following activities: (a) Composing gadgets from existing components, (b) combining functionality with existing constructions and (c) programming new components. Note that activity (a) includes two very different methods: Interactive composition and programmed composition. Further note that activity (b) has two different faces. Depending on the point of view, it can either mean adding functionality to some existing visual construction or creating a graphical user interface for an existing application. Finally, remember that (c) comprises programming on two different levels of complexity: Programming of container gadgets vs. programming of atomic gadgets.

Overview

Activity	**Method**	**Supporting Tools**
Composing gadgets	Interactive editing Layout description	Gadgets tool Inspector tool Interpreter (in planning stage)
Combining functionality & gadgets	Interactive editing Conventional programming	Inspector tool Gadget modules
Programming gadgets	OO Programming Implementation of basic message protocol (two levels of complexity)	Gadget modules Templates in source code

6 Conclusion

The design of a comprehensive and complete object-oriented environment has been explained, discussed and justified. The design of both the Oberon language and system was guided by the two somewhat controversal principles "unification of concepts" and "separation of concerns". As language and system designers we have obeyed our own recommendation and have made extensive use of reuse. We consider the result as a worthy member of the Pascal-Modula family whose lean and minimal characteristic is widely acknowledged. In a next step, we plan to integrate concurrency into the system in the form of active objects, thereby strictly preserving its original spirit.

Acknowledgement

I am greatly indebted to my collaborators, in particular to Hannes Marais (co-designer and implementor of Gadgets), Ralph Sommerer (co-designer implementor of object embedding in text) and Andreas Disteli (author of the DOS Oberon implementation, including a preliminary version of active objects). Without their enthusiasm and professional expertise, this project could not have been started, let alone successfully completed. My thanks also go to the (anonymous) referee for valuable suggestions.

References

[1] N. Wirth and J. Gutknecht. *Project Oberon.* Addison-Wesley 1992.

[2] H. Mössenböck and N. Wirth. *The Programming Language Oberon-2.* Structured Programming, 12(4): 179-195, 1991.

[3] A. Goldberg, D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley 1983.

[4] J. Gutknecht. *The Smart Document or How to Integrate Global Services.* GISI Jahrestagung 1995, Proceedings.

List of Figures