# New Directions in Programming Environments: Extensible Software

Günther Sawitzki
StatLab Heidelberg
Im Neuenheimer Feld 294
D 69120 Heidelberg

## Abstract

If we want software that can be adapted to our needs on the long run, extensibility is a main requirement. For a long time, extensibility has been in conflict with stability and/or efficiency. This situation has changed with recent software technologies. The tools provided by software technology however must be complemented by a design which exploits their facilities for extensibility. We illustrate this using Voyager, a portable data analysis system based on Oberon.

## Introduction

Extensibility is a critical requirement if we want software that can be adapted to our needs on the long run. The extensions required may be simple adaptations (like customisation of the user interface) or complex (like the integration of a new statistical method). Software technology can provide prerequisites for extensibility. The main contribution however has to come from system design, and the critical step is to separate abstract concepts and principles from implementation.

In the extremes, there are simple solutions to achieve extensibility. In a completely open system, we have complete extensibility. But we are prone to lose system stability. In the other extreme, in a closed system, stability may be guaranteed. But we can extend a closed system only by wrapping it up in a script or macro environment, and we will lose efficiency. We must find a balance between extensibility, efficiency and stability.

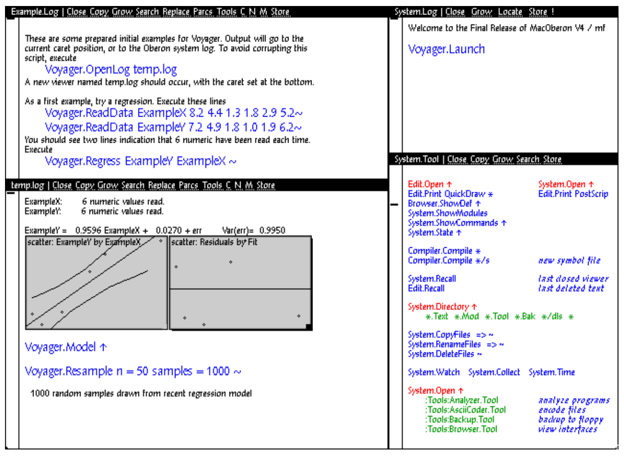
Driven by the need to define the directions of our own computing environment for the years to come, we started a project to explore how recent developments in software technology can be made fruitful in statistical computing. As part of this project we developed Voyager, a portable and extensible system. We will use Voyager as an example. Voyager is based on Oberon. Oberon is both a programming language and an operating system, developed at ETH Zürich. Building upon Oberon, Voyager provides the basis for interactive data analysis.

## 1. Voyager

A general introduction to Voyager is given in (Sawitzki, 1996). Some of the material needs to be repeated here, to give a first impression of Voyager. We will focus on aspects of extensibility. As an example, we look at possibilities to adapt the user interface.

Simple tuning of interface parameters does not need extensibility. Quite often however, we do not even want a statistical system to have an interface of its own, but we want the results imbedded in a report. We want the statistical output imbedded as an element in a text, and we would like to use our preferred editor and keep its interface. This may be a problem in a non-extensible system. In Voyager, output defines text components which can be imbedded in an environment of the user's choice. Figure 1a shows Voyager, used with the standard Oberon V4 text editor Edit.

Commands to Voyager are entered and executed from the editor envionment. Output is returned as text or as graphical text elements and placed in a text document. The connection to Voyager is not lost, even after the output is imbedded. So you can use brushing in one graphical text element and get the corresponding points highlighted in linked displays, or you can select single points and get their identification. If


Example.Log | Close Copy Grow Search Replace Parcs Tools C N M Store
temp.log | Close Copy Grow Search Replace Parcs Tools C N M Store
scatter: ExampleY by ExampleX
scatter: Residuals by Fit
System.Log | Close Grow Locate Store !
System.Tool | Close Copy Grow Search Store

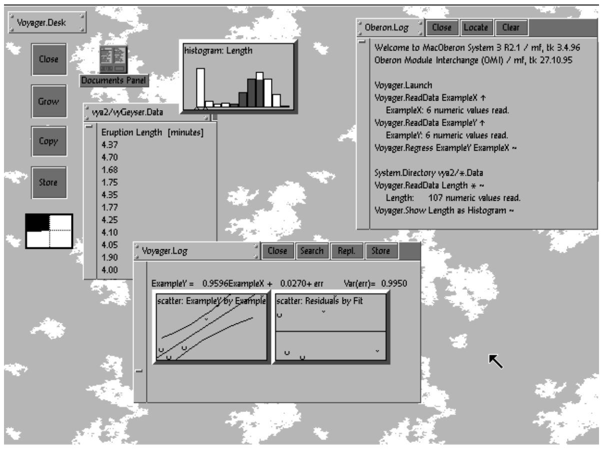
```
These are some prepared initial examples for Voyager. Output will go to the
current caret position, or to the Oberon system log. To avoid corrupting this
script, execute
      Voyager.OpenLog temp.log
A new viewer named temp.log should occur, with the caret set at the bottom.

As a first example, try a regression. Execute these lines
      Voyager.ReadData ExampleX 8.2 4.4 1.3 1.8 2.9 5.2~
      Voyager.ReadData ExampleY 7.2 4.9 1.8 1.0 1.9 6.2~
You should see two lines indication that 6 numeric have been read each time.
Execute
      Voyager.Regress ExampleY ExampleX ~
```

```
ExampleX:      6 numeric values read.
ExampleY:      6 numeric values read.

ExampleY =   0.9596 ExampleX +   0.0270 + err       Var(err)=  0.9950

Voyager.Model ↑

Voyager.Resample n = 50 samples = 1000 ~

 1000 random samples drawn from recent regression model
```

```
Welcome to the Final Release of MacOberon V4 / mf

Voyager.Launch
```

```
Edit.Open ↑                        System.Open ↑
Edit.Print QuickDraw *             Edit.Print PostScrip
Browser.ShowDef ↑
System.ShowModules
System.ShowCommands ↑
System.State ↑

Compiler.Compile *
Compiler.Compile */s               new symbol file

System.Recall                      last closed viewer
Edit.Recall                        last deleted text

System.Directory ↑
   *.Text  *.Mod  *.Tool  *.Bak  */dls  *

System.CopyFiles  => ~
System.RenameFiles  => ~
System.DeleteFiles ~

System.Watch   System.Collect   System.Time

System.Open ↑
   :Tools:Analyzer.Tool            analyze programs
   :Tools:AsciiCoder.Tool          encode files
   :Tools:Backup.Tool              backup to floppy
   :Tools:Browser.Tool             view interfaces
```

*Figure 1a Voyager integrated in an editor. Voyager output appears as integrated text elements.*



```
Welcome to MacOberon System 3 R2.1 / mf, tk 3.4.96
Oberon Module Interchange (OMI) / mf, tk 27.10.95

Voyager.Launch
Voyager.ReadData ExampleX ↑
   ExampleX: 6 numeric values read.
Voyager.ReadData ExampleY ↑
   ExampleY: 6 numeric values read.
Voyager.Regress ExampleY ExampleX ~

System.Directory vya2/*.Data
Voyager.ReadData Length * ~
   Length:     107 numeric values read.
Voyager.Show Length as Histogram ~
```

```
vya2/vyGeyser.Data
Eruption Length  [minutes]
4.37
4.70
1.68
1.75
4.35
1.77
4.25
4.10
4.05
1.90
4.00
```

```
ExampleY =   0.9596ExampleX +   0.0270+ err       Var(err)=  0.9950
```

*Figure 1b Voyager integrated in a desktop. Voyager output appears as integrated text elements, or as desktop components.*

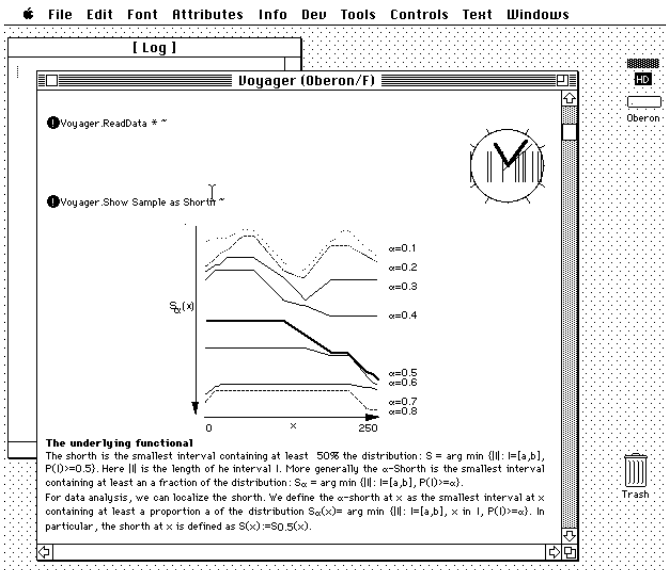
however our preference is towards a desktop model with windows and buttons, we can keep the desktop interface as in Figure 1b. Figure 1b shows Voyager, used with the Gadgets desktop environment under Oberon System 3.

Figure 1a and 1b both show Voyager in an Oberon environment. Figure 2a goes a step further. At first look, the document looks like a text document, following the convention of the host operating system. The host operating system shown is MacOS. In a Windows environment, the document will have the typical form of a Windows document. The only exception to the general look of the host system are some control elements imbedded in the text, which can be used to activate commands. The graph in this figure however is "live". It updates if the underlying data are changed, and new data can be analysed by just dropping them onto the graph as usual in an editor with drag-and-drop facilities. The “clock” in the upper right corner is a random number generator, generating a new sample periodically. If you drop it onto the plot, you get a “life” simulation.

Figure 2b shows a variant of Voyager on a Newton PDA. While the previous examples used exactly the same underlying program, the Newton variant of Voyager needs changes for using NewtOS instead of Oberon as a base system.

Of course the statistics in Voyager does not depend on the user interface. It follows the well-known methods, and displays provide the usual facilities which should by now be familiar in all statistical systems, including linked plots, brushing and identification of data points, and free rotation of 3D-data.

The polymorphism is the result of a basic design feature of Voyager and of Oberon: a strict separation of concerns. The user interface is strictly separated from the computa-

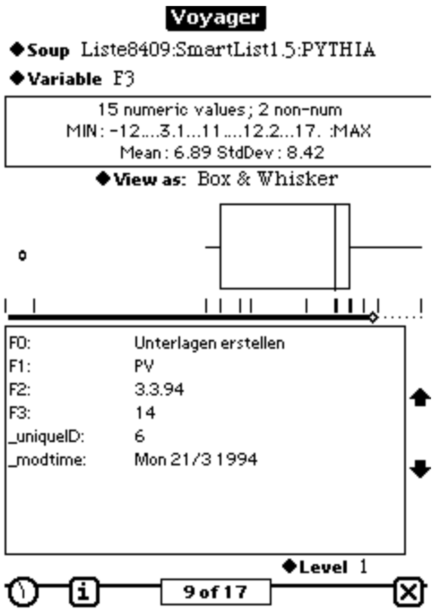


```
Voyager.ReadData * ~

Voyager.Show Sample as Shorth ~
```

**The underlying functional**
The shorth is the smallest interval containing at least 50% the distribution: S = arg min {|I|: I=[a,b], P(I)>=0.5}. Here |I| is the length of he interval I. More generally the α-Shorth is the smallest interval containing at least an a fraction of the distribution: $S_\alpha$ = arg min {|I|: I=[a,b], P(I)>=α}.
For data analysis, we can localize the shorth. We define the α-shorth at x as the smallest interval at x containing at least a proportion a of the distribution $S_\alpha(x)$= arg min {|I|: I=[a,b], x in I, P(I)>=α}. In particular, the shorth at x is defined as S(x):=$S_{0.5}$(x).

*Figure 2a Voyager as a document component. If you drop a data set onto this graph, it will plot this data set. If you link it to a random number generator, it will show a running simulation.*

tional kernel. This allows to use the same computing and data management, while showing quite different visual representations as required by the environment. The interface is handled by the general Oberon system. Voyager contributes components, which extend Oberon. For a general information about Voyager, see (Sawitzki, 1996). We will concen-



◆**Soup** Liste8409:SmartList1.5:PYTHIA
◆**Variable** F3

15 numeric values; 2 non-num
MIN: -12....3.1...11....12.2...17. :MAX
Mean: 6.89 StdDev: 8.42

◆**View as:** Box & Whisker

| | |
|---|---|
| F0: | Unterlagen erstellen |
| F1: | PV |
| F2: | 3.3.94 |
| F3: | 14 |
| _uniqueID: | 6 |
| _modtime: | Mon 21/3 1994 |

◆**Level** 1

*Figure 2b Voyager as an imbedded system on a Newton PDA.*

trate on some of the more technical aspects here.

## 2. Separation of Concerns

Voyager is a modular system. All modules have well-defined functions and communicate only by well-defined input and output. Each module can be freely replaced by alternate implementations, possibly extending the functionality.

A key feature of Voyager is a strict separation of concerns. For example, deriving statistical information is one issue, rendering a presentation and bringing it to the user is something else. In Voyager, the user interface is strictly separated from the proper statistical kernel. Separation of concerns is a principle that requires considerable system analysis before it can be put to practical work. We will give some examples later on.

In some cases, there are guiding models. One of these is the model-view-controller separation used in Smalltalk for interactive applications. The model component has to maintain consistency of and to provide access to the data to be presented. The viewer component is responsible for rendering and presenting the data. Several, possibly different viewers can exist on the same model, and consistency is guaranteed since only the model controls the data. The controller is responsible for handling the user interactions. It will co-operate with the viewer and possibly and may delegate the actual execution to external operations.

While the separation is easy in principle, it needs work in practical applications. Consider a simple plot, like a histogram. If you think of a histogram plot as a view and the raw data as a model, you are trapped. This separation is too simple to allow the most obvious extensions, like choice of a bin width and an offset point. And it cannot even handle the simple task to show a histogram, given the bin counts.

Separation of concerns and modular programming are keys to extensibility. Yet there may be a run time efficiency trade off. For example, as far as “classical” statistics is concerned, separating the user interface from computing does not impose any problem. In classical statistics, you have a data set and a test or an estimator. You apply the test or estimator to the data set and present the result. This scheme worked well in any batch oriented environment, and is simple to implement.

In data analysis, the situation is changed drastically. You start asking “what if”-questions, and these questions are defined by reference to the data at hand. You want to give this reference in an interactive way and of course you hope for an immediate feed back. For interactive data analysis, a close feed-back loop between user action and system response is necessary. If the user interface is separated from the computational kernel, this may be a problem. You need an efficient communication mechanism to allow for interactive data analysis.

Similar considerations apply to the data input side. If data input and data management are intrinsic immutable components of the data analysis system, extending the input facilities may be severely restricted. Access to data should be considered a communication process, not a controlled service. As a consequence, data access may be error prone or not guaranteed. Data may come in over distributed systems that may need to access data using instable channels. Or your analysis may run on an imbedded system with limited bandwidth. We should take these kinds of computing environments into account. Data consistency becomes a problem, and we must have precautions to handle it.

## 3. The Oberon Environment

For the implementation of Voyager, we have chosen the Oberon system as programming and run-time environment. We give a short overview here. For more information, see (Wirth & Gutknecht, 1989) or (Sawitzki, 1996)

Oberon is both a programming language and an operating system, developed at ETH Zürich. It is a light-weight system that can be implemented on top of (or besides) existing operating systems without much overhead. It is available as a co-operative emulation on top of most of the common operating systems (UNIX, MS-DOS, MS-DOS&Windows, LINUX, MacOS, etc.) and for most of the common hardware architectures (Intel x86, Motorola 68xxx, PowerPC, MIPS Risc, SPARC etc.). Oberon solves the portability problem for us: instead of porting our software, we use a portable system as a base.

The Oberon language is a strongly typed language in the tradition of Algol and Pascal. It supports object oriented programming. Oberon programs are modular with type safety guaranteed across module boundaries.

Type safety across module boundaries is supported by the Oberon operating system even at run time. Modules are linked dynamically and stay resident until unloaded explicitly. Garbage collection is supplied.

The quality of services provided for module loading may depend on the implementation. All implementations support a semantic version control upon load time. Attempts to load inconsistent modules are detected and trapped by the system. The mechanisms applied vary from coarse version controls to fine fingerprint systems that allows loading of modules that are inconsistent or out of date, but still are satisfactory for the application at hand. “Version” here refers to the version defined by the module declaration part– its signature, if you like. Oberon does not try to rely on programmer provided version information, like textual version stamps or .h files. If you consider extensibility for a long time schedule, of course fine fingerprints may save you from considerable amount of requests to rebuild the system.

More recent Oberon systems have means to keep code in a system independent way. Instead of generating the final run time code, an intermediate code is produced which is post-processed at load time and casted for the loading environment. The same compiled code can thus be used on different CPUs without recompilation. In contrast to traditional byte code interpreters that follow a fixed scheme, the coding scheme adapts to the code at hand. As a side effect, the code is compressed. Taking into account the access times from disk or other media, in most cases the portable code turns out more efficient than classical native code (Franz, 1994).

Resident modules and type safety across module boundaries are the basis for efficient and reliable communication with extensions. As long as the base system and the extension reside on the same CPU, we have a system with shared memory and in many cases communication boils down to passing a memory reference, while the system guarantees type safety for the access.

In contrast to weakly typed systems, we trade in some flexibility in a system where all types must be fixed at compile time. The potential bonus is speed and safety, since critical type checks are already performed at compile time. Since Oberon (as all object oriented systems) supports polymorphism, part of this flexibility can be regained using appropriate type hierarchies.

Graphical display and a basic window system are part of the Oberon operating system. Like PLAN 9, Oberon tries to avoid modes. Any text, or item displayed by the Oberon system may be used as an input - there is no separate command line or command menu. Any exported parameterless procedure is a command and can be executed in any textual context. The context can be retrieved by querying the global system state. So textual parameters can be accessed and evaluated without procedural parameters.

A large part of the flexibility and extensibility demonstrated above is due to the general properties of Oberon. Voyager adds specific modules to the general architecture of Oberon. Since modulers are resident, this can be done even at run time. Since state information may be exported, efficient communication with extensions is possible, while overall stability is supported. Of course an appropriate design is needed to make use of these possibilities. We will turn to some of the more technical aspects now.

## 4. Case Study: Graphics

We have seen various displays from Voyager  in Fig. 1 and Fig. 2. By convention, all displays in Voyager support the usual interaction, like brushing, inspection, etc., and corresponding displays are linked. In this section, we will have a short look behind the scenes.

Graphical display and interaction facilities are not uniform throughout platforms. To have a portable and extensible system, we need a convenient base definition that can be used in any environment. For this presentation, we assume that we have a unified graphic model supporting at least the graphic primitives like line and circle drawing. In practice, this is not the case (but we do have unifying glue modules for all major systems, so we can ignore this problem).

Voyager graphics is based upon ports. As usual, a port has information about its display space in the current display system, and an access channel to actually perform displays. Moreover, a port has a handler – the facility to respond to messages and interactions. User interaction is not unique throughout platforms. You may need translations. On some systems, you may have a three button mouse for input, on other system a one button mouse and a keyboard. On a PDA, you may have a pen and nothing else, while in still other systems you only have voice input.

### 4.1 Graphical Input

The Voyager graphic base defines events and lateral information, but leaves the implementation to be specified. Capturing low level events and translating them to the events understood by Voyager is left to glue modules. Since Voyager is under development, the list of events will be outdated at the time you read this article. This is the list of events supported at the time this article is written:

| | |
|---|---|
| `unknown` | `Just unknown` |
| `empty` | `No event` |
| `loc` | `Get location` |
| `ident` | `Identify` |
| `select` | `Add to selection` |
| `remove` | `Remove` |
| `copy` | `Copy data` |
| `copyAttr` | `Copy attributes` |
| `open` | `Open a new plot` |

From the selection of events we support, you can see that besides the usual interaction like brushing, identification etc. we have three activities we consider worth a central role: a remove action (the target of which must be defined) and two copy actions: an action to copy data itself, and an action to copy attributes only. An example for the latter is the copy action for formats: if you have two corresponding plots, some scales may be difficult to compare. You want to see them on a common scale. Using the Voyager model, you select one of the displays and copy over its attributes to the other. How you actually do the copy, i.e. which keys to press, may depend on your implementation. The system generically is able to support the concept of attributes, and to identify a “copy attributes” message. User interactions are translated into abstract events, and these are handled in a generic way.

An event message communicates additional information. At present, it records the position of the (a) mouse pointer at the time the event occurred, the time, and the status of optional control keys.

```
Event = RECORD
    gesture: INTEGER;
    x, y: INTEGER;
    time: LONGINT;
    keys: SET
END;
```

## 4.2 Graphical Output

On the output side, a problem to solve is to allow for output with several components aligned to a common reference system. One solution would be to place all components into a common reference plane, and to optimise their placement. This is not a trivial problem. In Voyager, we have chosen a different access. Since most display elements come in groups (like data points, ticks and scales, legends) we delegated the placement task to these groups. What these groups need is at least an information about a common (real world) reference system. This information is kept by the port. Each group is responsible to find its appropriate layout. The graphical display of all groups is added, like putting transparent layers on top of each other.

The appropriate abstraction to support this is an abstract type “layer”. Layers are attached to ports. Ports are the owner of real world co-ordinate systems. If a co-ordinate system is changed (zoomed, shifted…), all layers can react in a consistent way. A layer always covers the complete port. Only the port has information about the display environment. If the port is moved or resized, all the layers go with it.

On top of the abstract layer data type, there are data types for 2D and 3D plots that form the basis for all concrete extensions.

User interaction needs some caution. The first step to handle a user interaction is to find a port that might be the intended target. If the action refers to the content of the port, any one of the Layers may be the intended target, and a conflict may occur. We have chosen a bidding system. First, a message is passed through all layers informing them about the incoming user request. Each layer may return with a flag indicating that it could handle the request, and a weight. The responsive layer with the highest weight wins and gets the user request to handle it (of course possibly by delegating it). The message sequence can be controlled by the user, which allows some adaptation. This model is still an experimental model, and it is not clear whether it can handle all the requirements of forthcoming extensions. So far, it has performed to full satisfaction.

We do lose some fine tuning capabilities if we allow full control over the complete port area for each layer. So far the experience is that user control can compensate for this. But of course it would be nice to have an automatic placement system taking away the chores of fine tuning.

The graphics base is a componenet which rarely will be extended. However it has to be flexible enough to cover a wide spectrum of environments. It needs adaptation if you move to a new platform. After that, it is applied rather than extended. Extensions start with derived entities, like layers. The base catches host system dependencies, so derived layers are not affected.

## 5. Voyager Object Hierarchy

In contrast to the graphical base, the basis of the object hierarchy is often extended directly. The first definitions have to be most versatile to allow all kinds of extensions. On the other hand, it has to encapsulate all management and house keeping information needed for a stable system.

Voyager works with objects. A Voyager object can be anything that is worth its own identity, like data, access information, displays. Voyager objects are abstract data types. As objects, they can encapsulate data and methods, and of course the semantics of the data part is defined in terms of the methods. Objects are linked in a list. A simplified picture of the Voyager object structure is given in Fig. 3.

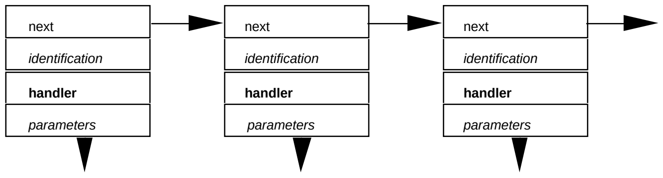


*Figure 3: Simplified Structure of a Voyager Object Chain. Objects are linked in a list. All Objects have a handler responding to incoming messages*

This picture will look familiar to FORTH programmers, and LISP programmers will find an easy way to translate it into a familiar picture for them.

## 5.1 Message Passing

Following the usual Oberon conventions, the handler field does not hold a reference to fixed code operating on the internal parameters, but a reference to a message handling procedure. The handler procedure takes two arguments, a message, and a reference to an object to operate on, usually the containing object. It tries to identify the message. If that is identified as a message the object can handle, the handler proceeds and responds to the message. If it cannot handle the message, by convention the message handler stays quiet. This is a common strategy in Oberon programs, and discussed in (Marais, 1996). Action requests are passed by messages. Ignoring unidentified messages allows adding message types at a later stage. New objects that understand messages of an extension can react to them, while older components simply ignore them and pass them on.

This strategy is possible because messages can be polymorphic and allow type inspection. It relies on a language feature of Oberon. In Oberon, you can define abstract data types, like records, and you can extend records to give derived data structures. We use this language feature extensively. At the base, we define

```
TYPE Message = RECORD
END;
```

So to begin with, a message is just an empty record. One extension is

```
PrintMessage = RECORD
(Message)
END;
```

This is still an empty data structure. But in a strongly typed environment, the type of a PrintMessage is clearly distinct from any other message type, and a handler that has to handle an incoming message, Msg say, can query the message type using a language construct

```
IF Msg IS PrintMessage THEN …  ELSE …  END;
```

No additional information is necessary. The type of the incoming message may be enough to decide upon the action to take, and no more overhead occurs. Of course additional information may be appropriate, and it is possible to add data fields to a derived data structure. It is not required if it is not necessary.

Only the very basics are defined for general Voyager objects, and as much as possible is left for extensions. The design challenge is to find a basis that is light enough to allow flexible extensions without imposing to much overhead, yet being solid enough to provide effective support for an interactive system. For a generic object, we have defined message types

| | |
|---|---|
| `LoadMessage` | `internalise (read from a file)` |
| `StoreMessage` | `externalise (write to a file)` |
| `ReadMessage` | `read from display environment` |
| `WriteMessage` | `write to display environment` |
| `PrintMessage` | `print it` |
| `DumpMessage` | `write extended information` |
| `UpdateMessage` | `adjust to changed state` |
| `CopyMessage` | `copy object` |
| `CloneMessage` | `copy object, including all referenced objects` |

Data, for example are represented by an abstract data type based on objects that understands additional messages for all arithmetic operations. Extensions introduce new object types, based on generic Voyager objects, and message types as required.

## 5.2 Names and Identification

Fig. 3 is simplified. Identification and parameters have more structure and meaning. Let us consider identification first. We may see identifications in various rôles. Identification may mean a user reference to some object. This may be context dependent and need not be unique. For example, the term “Residual” may be used as a generic term to refer to the residual of the most recent estimation. Identification may as well mean the unique identification of a data item or a variable, irrespective of how it is named by a user. Or it may mean the identification of a specific state of a data item or a variable. This may be relevant in a situation where you have on-line data that may change with time, or if you work in a distributed environment, where consistency over different sources is not guaranteed but needs to be checked. It would be fatal to assume a simplistic model for identification, like some name string.

The identification must be flexible enough to anticipate extensions. On the other hand, it must be defined at a very low level due to its central rôle. The solution taken in Voyager is to separate naming from the basic object management. The identification part has several components.

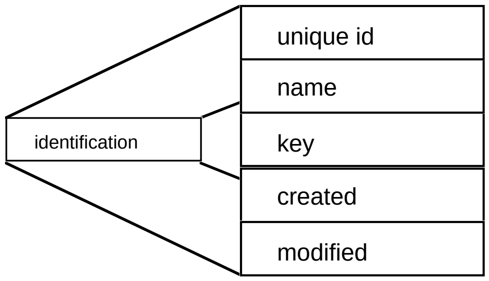


*Figure 4: Object Identification*

A “unique id” field holds a system defined identification. This identification is used for internal purposes only. In particular, it can be used for externalisation and internalisation of objects, i.e. to support persistent objects. In an environment where you store complete memory images to have a persistent state, this would not be necessary. You could use (relative) pointer addresses to identify objects. If you do not insist on complete memory images, you need a possibility to identify objects that is independent of addresses, and the “unique id” satisfies these purposes.

The “name” field holds an identification on user level, usually some string. It is vain to define the name structure ultimately. There will be always some application that needs a more complex alphabet, or a more complex name structure then you anticipated. So instead of fixing the name data type, this is an abstract data type that needs to be extended by some specific implementation. As a consequence name management and lookup are not a part of the Voyager kernel, but a service provided by a separate module. You can replace this module by a module of your choice to adjust the naming system to your needs. Only the abstract rules are fixed in the Voyager kernel. The naming system must be able to use a “name” to look up one or more objects and return the unique identification of these objects. It should be able to handle the arbitration in case of conflicts. The name stored with an object is only a cached value of some recently used references, used for reporting purposes.

In a similar way, the “key” field is defined in an abstract way, and interpretations are delegated to secondary components. The key is used to ease the aliasing problem. Imagine you have accessed some data. You have cached the data, and made some extensive calculations based on your personal copy of the data. Now imagine the data originated from some external source. When reporting your results, you should verify that your data are still valid. The key is the abstract instrument to conclude your transaction. You have to call a system service to verify that your key is still valid. Again, the details are left to secondary components. The Voyager kernel supports an abstract data type to hold a key, and defines the appropriate actions.

Creation and modification time and date are additional fields. They should not be used to verify validity of data - spoofing clocks is too easy, and keeping clocks in synchrony is too difficult. But in well-behaved environments, time and date stamps may be a helpful tool to monitor an analysis.

## 5.3 Parameters

Most of the parameter structure needs to be left to extensions. Only some general administrative structure can be defined in a generic way. The most important feature we have identified is support for interactive features. Although the dependency may form a complex graph, the central need is to notify all direct “dependants” if the state of some object changes. To support this, each object can hold a list of its “users”. If the state of an object changes, all dependants need to be notified. There are two system services needed to support this service. You need a subscription service that registers one object as a user of a different object, and you need an update service that marks one object as updated and informs all users. As is to be expected, message passing is used in Voyager to implement these services.

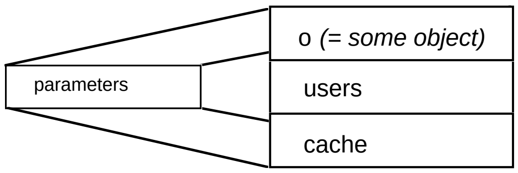


*Figure 5: Generic Parameter Field*

The cache field is a generic field supplied for optimisation. Any calculation can use it to store intermediate results that may speed up later accesses. The cache field is cleared if the object is modified, using the system update service.

The “o” field is a trick to overcome certain design decisions of Oberon. Oberon is designed to be a lean language, and all features that did not have efficient implementations or obvious use have been excluded. Unfortunately, as a consequence, list structures and iterators are not supported as primitive elements in Oberon. Adding a free object slot, the “o” field, allows to implement generic queues and iterators. To avoid overhead, the basic Voyager types are “items”, not objects. Items are like objects. They have a handler, can reference to another object, and can be queued. Yet they do not have an identification, or administrative overhead. Voyager objects are implemented as descendants of items. All types that can benefit support from the generic services of Voyager are extensions of Voyager objects.

## 6. Case Study: Random Numbers

We have seen the extensibility, or adaptability of the user interface. Although this is most spectacular seen from the outside, from a system point of view this is only superficial. We have seen how the basic system defines abstractions designed for extensibility. The need to design components for extensibility or exchangeability is not restricted to the abstract base levels. It occurs again on all levels of the system. As an example, we will study how to implement random number generators in a re-usable extensible way.

Generation of (pseudo) random numbers is at the base of many statistical methods, like bootstrap and other resampling methods. And of course random number generation is the heart of statistical simulation. An elementary implementation would be to provide a function (Random, say) and a way to initialise the random number generator (a procedure SetSeed, say) which sets a state variable (Seed, say) of the random number generator to a specific state. The extension to be anticipated here is that you want to replace some default random number generator with a generator of your choice.

Thinking of common models like linear congruential generators, a first implementation would be to supply a global seed variable and a transition function

```
VAR Seed: LONGINT;
PROCEDURE Random(): LONGINT;
```

where Random runs the random number generator one step and returns the new seed. What is the problem with this setup as far as extensibility is concerned? Obviously, you cannot replace the Random number generator with a generator of your choice. In an environment where functions are first class citizens, the solution is to introduce a variable of procedure type to hold the actual random number generator. The second problem is that seed and transition function are intimately related. If you replace the transition function, the old seed looses its meaning. It may even contain a value that is out of the scope of the new random number generator. The way out is to encapsulate random number generator and seed in an object structure leading to

```
TYPE   Generator = RECORD
    Seed: LONGINT;
    Random: PROCEDURE(): LONGINT
END;
VAR    CurrentGenerator: Generator;
```

For convenience, you may always provide an access function which shields the user from the details and just returns the next value of the current random number generator, whatever that may be, like

```
PROCEDURE Random(): LONGINT;
BEGIN RETURN CurrentGenerator.Random()END;
PROCEDURE SetSeed(VAR seed: LONGINT);
BEGIN RETURN CurrentGenerator.Seed:=seed END;
```

An open question is how we should handle random number generators with differing base type. So far, we used one base type, long integer numbers, in these examples. You may want to install a random number generator that has a period that is beyond what can be represented as a long integer number. You may even have a random number generator that has state information that cannot be expressed in a numerical variable at all. So you must allow for type extension for the seed as well. The way out is to introduce the seed as an abstract data type. You then define the required actions as operating on variables of this abstract type. By defining the type as an extension of the generic object type, we can use all the services of the basic Voyager system.

```
TYPE   Seed = RECORD END;
    Generator = RECORD (Object)
    GetSeed: PROCEDURE(VAR seed: Seed);
    SetSeed: PROCEDURE(VAR seed: Seed)
    NextRandom: PROCEDURE();
    GetRandom: PROCEDURE():LONGINT;
    GetRandomX: PROCEDURE():LONGREAL
END;
```

In this data structure, we have added one more element. We have separated the transition function NextRandom from the read-out function, which we supply in two forms: give a long integer number defined by the current state of the generator, and give a long real number. If you allow for now ranges, there is no guaranteed common rule how to derive standard cased like uniform random numbers over [0,MAX(LONGINT)[ or [0,1] from the seed. So instead of relying on user supplied transformations, we require the random number generator to supply the common forms. We have re moved all information about the implementation of the seed from the abstract definition. If we allow for extensibility, there is no way to allow direct access to the state information. Any specific implementation now has to define its appropriate extension to the seed data structure, and its specific set and get functions. (Of course, as a common case, the long integer variants are pre defined.)

## 7. Summary and Conclusions

We have seen various design details of Oberon. We will close with an example where these pieces are put together and used in action. This example is taken from a recent project of M. Diller, a system for simulation of locally stationary time series, built upon Voyager.

The simulation combines a series of Voyager objects. Voyager does not know about the internals of this simulation. But it knows about their dependency structure and can produce a dependency graph. Moreover, it can identify the class of each of these objects. By a minor extension, the nodes of the graph can be made active buttons and used to call a graphical editor, say, if the object is some input function, or a line plot for real valued output. These displays are pre-defined components supported by Voyager, The application programming can concentrate on the original application tasks.

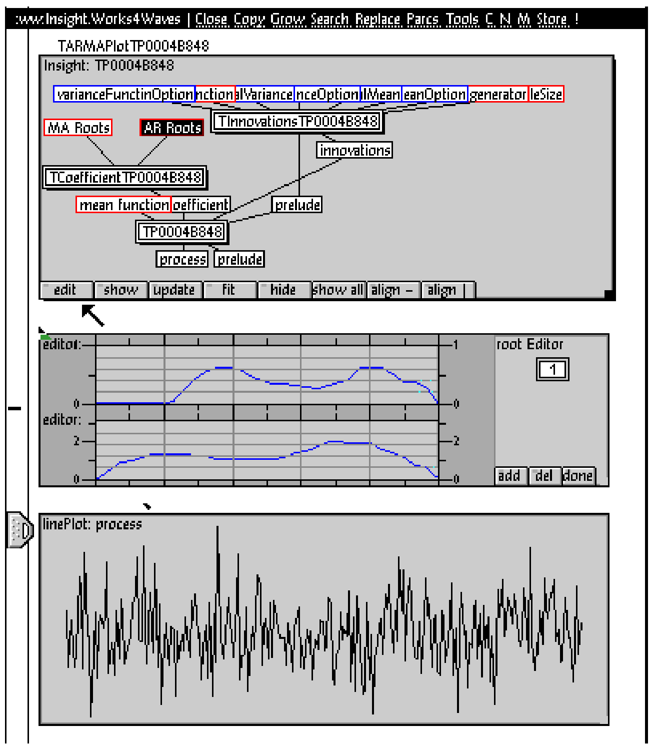


*Figure 6: Sample Application. Voyager used to simulate locally stationary time series. A dependency graph is generated automatically. This graph at the top can be used to evoke additional displays, as the function editor in the middle used for interactive control of phase and amplitude of the ARMA roots, or the output display at the bottom.*

In time past, we have uses program libraries to develop our programs. In times present, we are used to systems and packages. We have gained considerable conveniences, but we have lost the flexibility and control. With component technologies we can have both, a convenient system where we want it, and the flexibility once provided by libraries.

## References

Franz, F. (1994) *Code-Generation On-the-Fly: A Key for Portable Software*. Dissertation. Institut für Computersysteme, ETH Zürich. 1994.
see <http://www.inf.ethz.ch/publications/diss.html>

Marais, J. (1996), “Extensible Software Systems in Oberon,” To appear in *Journal of Computational and Graphical Statistics* Vol. 5 No. 3, 1996.

Sawitzki, G. (1996), “Extensible Statistical Software: On a Voyage to Oberon”. To appear in *Journal of Computational and Graphical Statistics* Vol. 5 No. 3, 1996.

Wirth, N.; Gutknecht, J. (1989), “The Oberon System,” *Software–Practice and Experience*, 19(9), 857–893.

Figure 2 is taken from (Sawitzki, 1996).

For more information see
<http://www.statlab.uni-heidelberg.de>

To appear in: Proceedings of the 28th Symposium on the Interface, Sydney 1996