An object oriented application for corporate networks design

Erik Chrisment
France Telecom Cnet
06921 Sophia Antipolis Cedex - FRANCE
chrismen@sophia.cnet.fr
(+33) 92 94 53 10

1.0 Introduction

For several years, the CNET has developped many tools to design networks, especially corporate networks. Many of them are based on very efficient algorithms, some of them are intended to help the network designer to identify the input data. Others tools compute the cost of a network solution.

Recently, many of these tools were developped independently, without any friendly nor common user-interface. Because of these differences, few people were able to use successively several of them. So, this situation led the CNET to launch a new project whose target was to offer network designers an application containing most of the best home network design tools, with a high-level integration and a friendly graphic user interface (GUI). This application (ORIENT) must be open, which means that it must be possible without much work to integrate new design tools.

This paper presents ORIENT which is based on a dynamic language with an object oriented layer : the LISP dialect ILOG-TALK [TALK 95] and its associative object system POWER-CLASSES [POWER-CLASSES 95]. This application is quite a novelty because it combines, thanks to the TALK/C++ binder, a dynamic language with a static one : indeed, the GUI is made with a C++ classes library and some critical tools are also written with C++.

Section 2.0 explains why we chose a dynamic language for our application.

Section 3.0 presents the application internal architecture. Each part of ORIENT has been made as independently as possible, and there is especially a strong gap between the applicative world and the graphical one ; a specific tool manages the dialog between these worlds, it is presented in Section 4.0 .

Then, Section 5.0 concludes and presents our future work in dynamic objects.

2.0 Why dynamic objects

In this section, we try to explain why we chose an object oriented approach with a dynamic language as LISP.

The major reasons are :

- Modelling : A powerful object language is more appropriate to model a domain close to the reality. With Power-Classes, we used single and multiple inheritance, relations between classes, demons on slots. As in CLOS [Steele90], the Meta Object Protocol allows to customize the object oriented language, we used it for the allocation of some classes.
- Simplicity and flexibility : One of the advantages of LISP lies in the fact that development is fast, because we do not have to worry about pointers, memory allocation,… LISP is well adapted to incremental development. Moreover, generic functions that can be overloaded with several methods provide flexibility and simplicity. It is a proper language for integrating new tools.
- LISP interpreter : It is essential in 2 ways.
  At development time because it is easy to test new functions.
  At runtime ; indeed, the ORIENT user can use the interpreter in a specific panel : with some basic knowledge on LISP and the ORIENT API, an “expert” user may write functions which can connect up basic Orient operations to solve specific problems. Moreover, thanks to the interpreter, we can rapidly help a user who met a bug : we often just have to send few lines of LISP code to solve the problem.
- portability : we needed a portable technology to supply ORIENT on different platforms.

Then, we chose the LISP dialect TALK because it matches these needs and it has another important characteristic : opening. TALK provides a high integration between C and TALK programs : we can call TALK functions in C programs and *vice versa*. Moreover, TALK provides a TALK/C++ binding which allows us to automatically access any C++ libraries.

3.0 Architecture of the application

This section begins with a short description of the functionnalities of ORIENT and then presents the internal architecture of the application.

3.1 Overview

ORIENT is composed of 2 major kinds of tools, applicative tools and user-interface tools.

Many of the **applicative tools** come from combinatorial optimization activities, others has been developped especially for ORIENT. These tools are :

- Graph handling algorithms : shortest path, flow routing, minimum cut, minimum spanning tree…
- Network algorithms : these tools have to solve combinatorial optimization problems to find the best or the optimal solution among a finite or infinite number of alternative solutions. They are specialized for networks and take into account traffic, company geographic description, parameters describing service quality… A good solution is a topology which can carry traffic with a minimal cost ; some of these tools are heuristics : for example, we can cite simulated annealing ;

- Costs functions : from database files describing the costs of the services which are sold by the most important carriers in the world, the cost functions compute the client cost of a solution. The user can compare the prices and choose the carrier which will propose the cheaper solution ;
- Performace evaluation functions : these functions may be used during the optimization phase or on a given network.

User-interface tools : The user interface is composed of panels. Among these panels, there are editors — especially, there are a grapher which allows network edition, a spreadsheet which allows traffic matrices and objects attributes edition — and all the windows from which the applicative tools can be run.

So, we can understand that ORIENT is composed of an important number of heterogeneous tools. Moreover, in the future, new tools will be added, and old ones could be removed. It was very important for the architecture of ORIENT to be open. We will now describe it in more details.

3.2 Internal architecture

We first describe here the architecture of ORIENT. In a second part, we will introduce the organization of the applicative part.

The main characteristic of ORIENT architecture is modularity of the system components. We avoided mixing different kinds of codes. The API of each functionnality is published and most of the functionnalities can be used outside ORIENT.

Any applicative object has no knowledge as to what its graphic representations are. Therefore, there is a strong separation between the applicative part, which is the 'actual' application , and the graphic one. The mediation between these 2 parts is made by a dialog manager (DM) — DM is introduced in the next section.

For graphical tools, we chose a C++ graphic library which handles 2D vectorial drawing, and widgets building. These graphical tools are integrated in ORIENT using the TALK/C++ binding and DM.

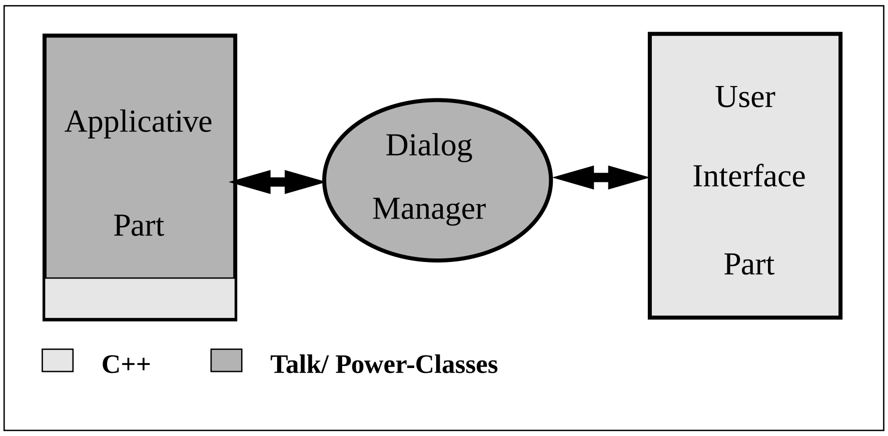

FIGURE 1. Global architecture

The figure 1 shows the 3 main parts of ORIENT : the applicative part can be used without its user interface through its API, and the user interface part can be change into a new version or used in another application. The dialog manager part manages the mediation.

The applicative part (figure 2) is, of course, very important. We did not try to find a generic model of telecommunication networks that would be optimally adapted to every tools : specific treatments need specific structures. So the domain has been modelled close to the reality with POWER-CLASSES and a powerful software layer, composed of TALK generic functions, has been developped to handle graphs. This layer, we called GREO, is the core of ORIENT. GREO has many interesting features : the graph elements can belong to several graphs, a graph element can contain subgraphs, a graph can have partial graphs…

Because such tools can run for a long time, some of them are written in C or C++ with proper internal structures or classes. Then, an interface software layer is set between the tool and ORIENT to convert tool structures into ORIENT objects and *vice versa*. Others tools are written in TALK as a set of generic functions and directly work on ORIENT objects.

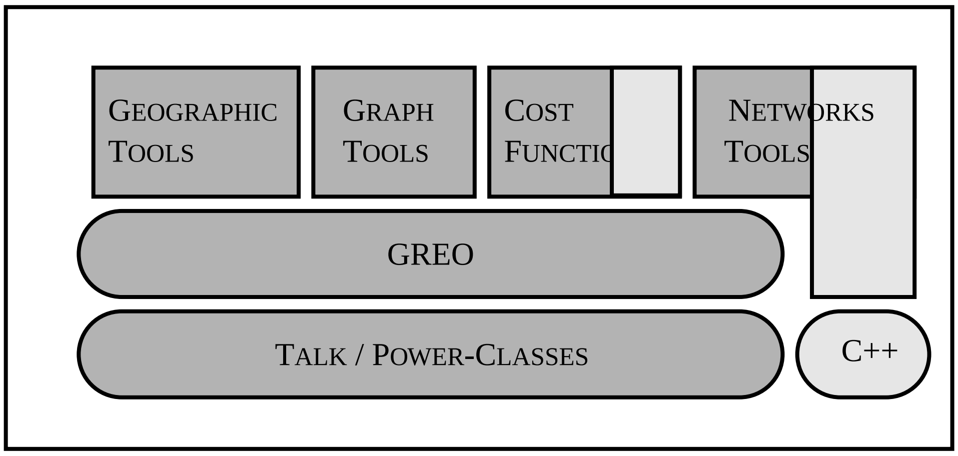

FIGURE 2. Applicative part architecture

4.0 The dialog manager

We do not demonstrate here the interest of a strong separation between the application and its GUI. Everyone knows that it is a necessary, but not a sufficient, condition to get an easy maintenable application. MVC [Krasner88] is the most well-known concept which demonstrated the advantages of this approach. Moreover, it allows us to provide the ORIENT applicative part as a library which can be further linked to any other appli-cation.

In this section, we present the software layer DM, that we created to handle the dialog between the application and the GUI. The GUI must be consistent with the application and this mediation tool is responsible for the initial construction and the permanent updating of the display. The code that handles applicative data is isolated from the code that handle graphics input and output : working on the application itself, we never worried about the GUI.

DM has been developped with TALK and POWER-CLASSES. Callbacks are written in TALK and set in the widgets from the DM part. Using the TALK/C++ binding, all the C++ widgets can be handled from TALK and all their public member functions are available.

The DM part is composed of **views** that are organized in a tree. These views manage the consistency between applicative objects and their graphical representations. They are defined with a **legend**. The legend is the way to update graphical objects when some events modify applicative objects, it is defined with 3 generic functions for creating, refreshing or removing graphical objects.

In a more detailed view, the architecture of ORIENT can be seen as follow :

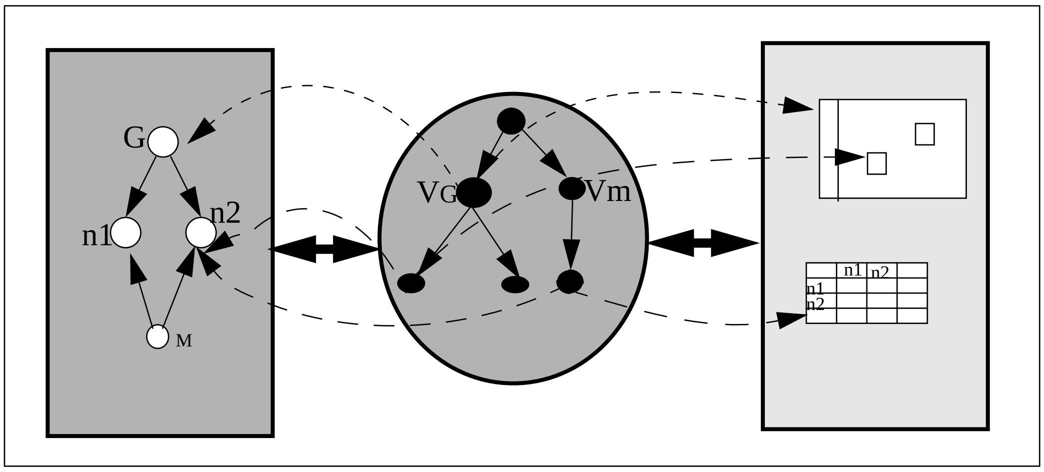

FIGURE 3. Exemples of lreations between applicative and user-interface objects

The applicative part contains a graph (G) , composed of 2 nodes, and a traffic matrix (M) composed with the same nodes. G is graphically represented in a graph editor and M in a spreadsheet. A dialog manager view (Vg) references G and its associated editor. The 2 nodes are referenced by 2 subviews of Vg which know their display objects in the graph editor. For M, the view Vm and 2 subviews point on the matrice editor.

An applicative object may have several graphical representations. Note that it is not mandatory to instanciate so many views : the views are instances of a class that can be derived.

DM supplies several services. They are :

- a notification mecanism associated to a subscription service ;
- an incremental updating of the display ;
- a delayed updating of the display.

In the next sub–sections, we present these services.

4.1 Notification

This service is composed of :

1. a notification service which allows objects to send signals ;
2. a subscription service to the notification mechanism.

Besides the declaration of a class, we specify the slots that can notify signals and the type of these signals. These slots will be able to send a signal at modification time. This signal is emitted by a *after-write* demon.

The specification is done outside the class declaration, in an other module. It is inherited. When needed, this service can be desactivated at any moment ; we will see, further in the paper, that it is very important for ORIENT.

Within DM, an instance of such a class subscribes to this service when it is graphically represented. Until that moment, no signal is emitted.

4.2 Incremental update

When application objects are modified, they send a signal to DM if they are graphiccaly represented. This signal is buffered and it will be treated by the dialog manager engine.

A signal is a small object which contains information of the applicative modification : the object that has been modified, its slot, its new value. The engine analyses this information and dispatch the signal to each view of the modified applicative object. Then, thanks to their legend, the views can update the associated graphic objects.

So, the state of the user interface part depends on the applicative part. When the user activates a widget callback, applicative functions are fired. They will change applicative objects. It is only when the engine receives messages from these objects that the display is updated. A callback ends with the activation of the engine.

4.3 Delayed resynchronization

But, in our application, there are many tools, especially algorithms which make combinatorial optimizations. These algorithms can run for a long time and they can test many solutions that will have an important impact on applicative objects. In that case, we do not want an incremental updating of the display representation. It is unuseful (except for debugging) and it can take a long time.

With DM, it is possible to stop for a moment the sending of signals. Then the *after-write* demon does nothing, only a function call is lost. The applicative part is modified by the algorithm and the resynchronization process will update the user interface at the end of the algorithm.

This delayed resynchronization process applies to a view, but it can be run on the entire application if the view is the root of the tree. It computes the differences between the applicative objects, returned by an overloaded generic function, and the associated graphical ones. It first removes obsolete graphical objects, then it adds new ones and it refreshes old graphical objects that are kept. It is a set of generic functions that can be overloaded for specific situations. It is also used to restore the graphical representation of an applicative object whose editor has been closed.

5.0 Conclusion and future works

The architecture of our application has been defined according to the following concepts : genericity, modularity and maintenability. The last one can be see as a consequence of the 2 previous.

- Genericity because our components are designed to be reused as much as possible.
- Modularity because it is an important feature to have a good maintenability level.

DM has been developped in CNET for our own needs and it is, at present, finalized and packaged by the software editor ILOG, that will distribute it.

For the moment, applicative and user-interface parts are set together with DM in one process. In the next version, we want to keep these parts in different process using sockets. The mediation part will remain in the same process as the applicative part and any graphical function call will be send through the socket, as a printable LISP form, to the user interface part. This LISP form will be so small that it will not need any high throughput link between the 2 hosts.

An other objective is to supply ORIENT on different platforms. At the moment, it runs on Sun Sparc Station, we will compile it on WINDOWSNT and on WINDOWS95.

6.0 Bibliography

[Krasner88] : G. E. Krasner and S. T. Pope. A coobook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *JOOP*, pages 26-49, August 88.

[Power-Classes95] : ILOG POWER CLASSES, Reference Manual, version 1.3. Prototype was developped by the CNET with the Meta Object Protocol TELOS of ILOG-TALK.

[Steele90] : G. L. Steele JR. Common Lisp, the language. Second Edition.

[Talk95] : ILOG TALK, Reference Manual, version 3.13.