Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Collaboration to the COLOS project. The use of an Object-Oriented environment, RMG, for the creation of interactive simulation applications

de Paul de Barchifontaine, Dominique

Award date:
1991

Awarding institution:
University of Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 27. Sep. 2024

Facultés Universitaires Notre-Dame de la Paix
Namur

Institut d'Informatique

Collaboration to the COLOS project. The use of an Object-Oriented environment, RMG, for the creation of interactive simulation applications.

Thesis presented by Dominique de Paul de Barchifontaine
in order to obtain the degree of Licencié et Maître en Informatique

Promoter - Professor Philippe van Bastelaer

Academic year 1990-1991

May we be permitted to thank our promoter, Professor van Bastelaer, for his good advices all along this work.

We would like also to thank Mrs Véronique Nachtergaele with whom we realized the 'OSI on X-25' application and who was of great help in any moments. We thank Mister Dominique Corbugy and Mister Joël Denis for our great collaboration within the Namur's COLOS team

Our whish is to thank also Doctor Hermann Härtel, Mister Uwe Heimburger, Mister Detlev Wegener and each person working at the Institut für die Pädagogik der Naturwissenschaften of the University of Kiel (West Germany), who helped us achieve our work there, during five months.

Finally, we thank each member of the COLOS project and each person who took part directly or indirectly in the achievement of this work.

Abstract

In this work, we first present the basic principles of Object-Oriented languages; this concerns a general presentation and a presentation of the Objective-C language. This language is the base of the RMG (Real-time Measurement Graphics) environment presented next, which is the Object-Oriented environment used in the COLOS (COnceptual Learning Of Science) project to build interactive simulation applications. We give a small guide to this environment which enables a programmer to start developing applications inside it. We introduce also the different projects which are born from the Namur University for the COLOS project and concerning telecommunications using ISO (International Standard Organization) OSI (Open Systems Interconnection) standard. Finally, we present one of the applications created in Namur by its COLOS team: the 'OSI on X-25' application.

Résumé

Dans ce travail, nous présentons tout d'abord les principes de base des langages orienté-objets; ceci concerne une présentation générale et une présentation du langage Objective-C. Ce langage est la base de l'environnement RMG (Real-time Measurement Graphics) qui est présenté ensuite et qui est l'environnement orienté-objets utilisé dans le projet COLOS (COnceptual Learning Of Science) afin de construire des applications interactives de simulation. Nous donnons un petit guide de cet environnement qui permet à un programmeur de commencer à y développer des applications. Nous introduisons aussi les différents projets nés à l'Université de Namur pour le projet COLOS et qui concernent les télécommunications utilisant le standard OSI (Open Systems Interconnection) de l'ISO (International Standard Organization). Finalement, nous présentons une des applications créées à Namur par son équipe COLOS: l'application 'OSI on X-25'.

Table of contents

Acknowledgments

Abstract

Table of contents

Chapter 1: Introduction 1

FIRST PART: Object-Oriented languages and RMG 2

Chapter 2: What is Object-Oriented programming ? 2
- 2.1. Principles of Object-Oriented languages 2
  - 2.1.1. Objects, data, methods and classes 3
  - 2.1.2. Encapsulation, inheritance and messages 4
- 2.2. Programming with Object-Oriented languages 9
  - 2.2.1. Object-Oriented programs 9
  - 2.2.2. Finding the right objects 13
- 2.3. Differences between tradition and 'Object-Orientedness' 15
- 2.4. Summary 17

Chapter 3: A specific approach: Objective-C 19
- 3.1. Main topics of the language 19
  - 3.1.1. Introduction 20
  - 3.1.2. The syntax 21
  - 3.1.3. The main program 26
  - 3.1.4 Syntax summary for a class definition 27
- 3.2. How it works 28
  - 3.2.1. Compiling 29
  - 3.2.2. Instances 30
  - 3.2.3. Messages 31
  - 3.2.4. Inheritance 31
  - 3.2.5. Methods 32
  - 3.2.6. Self and Super 33
- 3.3. Illustration through an example of Objective-C program 33
- 3.4. Summary 38

Chapter 4: Looking at RMG 40
- 4.1. The COLOS project and RMG 40
  - 4.1.1. RMG as a Graphical User Interface 40
  - 4.1.2. RMG as an application development platform 42
- 4.2. First steps in RMG 43
- 4.3. Getting accustomed to RMG with two applications 47
  - 4.3.1. IconEdit 48

4.3.2. MoleView ... 51
4.4. Summary ... 53

SECOND PART: A little guide to RMG ... 55

Chapter 5: Interesting bibliography ... 55

Chapter 6: How to create a new application in RMG ? ... 58
6.1. What is an RMG application ? ... 58
6.2. What to include in a new class ... 60
6.3. Programming without RMG tools ... 65
6.3.1. Editing ... 65
6.3.2. Makefiles, mainClasses and compilation ... 65
6.3.3. Including the application in the RMG environment ... 68
6.4. Programming with RMG tools ... 69
6.4.1. Editing ... 69
6.4.2. Compilation ... 72
6.4.3. Browsers ... 73
6.5. Summary ... 74

Chapter 7: First useful classes ... 76
7.1. RMGView ... 76
7.1.1. Instance variables ... 76
7.1.2. Factory methods ... 77
7.1.3. Instance methods ... 81
7.2. Envir ... 82
7.2.1. Instance variables ... 82
7.2.2. Factory Methods ... 82
7.2.3. Instance methods ... 83
7.2.4. C functions ... 83
7.3. RMGString ... 83
7.3.1. Instance variables ... 84
7.3.2. Factory methods ... 84
7.3.3. Instance methods ... 84
7.4. RMGIcon ... 87
7.4.1. Instance variables ... 87
7.4.2. Factory methods ... 87
7.4.3. Instance methods ... 88
7.5. Summary ... 89

Chapter 8: New notions about programming in RMG ... 90
8.1. The actions ... 90
8.1.1. C functions ... 90
8.1.2. Methods ... 94
8.1.3. RMG actions ... 95
8.2. The menus ... 96
8.2.1. The specification of a menu ... 97

8.2.2. Which class can be used ? 101
8.2.3. Actions in a menu 103
8.3. Additional features 108
8.3.1. The mouse 108
8.3.2. Iconizing an application 109
8.3.3. The active collection 110
8.4. Summary 113

Chapter 9: The Video application 115
9.1. General description of the application 115
9.1.1. Introduction 115
9.1.2. The hardware 115
9.1.3. The application's functionalities 116
9.2. The application on the screen 118
9.3. The application in deep 120
9.3.1. Instance variables 120
9.3.2. Factory methods of the 'Video' class 121
9.3.2. Factory methods of the 'Video' class 122
9.3.4. Instance methods 122
9.3.5. The Actions 125
9.3.6. The mouse 126
9.3.7. The menu 127
9.3.8. The active collection 128
9.4. The classes used 129
9.4.1. Fixtur17 129
9.4.2. ModStrI 130
9.4.3. RMGLine 131
9.4.4. Displaying personal icons 132
9.5. Problems encountered 133
9.5.1. Creating and setting a device file 133
9.5.2. Problems while quitting the application 134
9.5.3. Critique 135
9.6. Summary 136

THIRD PART: Illustrating telecommunication principles under RMG 138

Chapter 10: Representing computer telecommunication under RMG 138
10.1. OSI basics 138
10.2. The different steps in representing the OSI model under RMG 140
10.3. Summary 141

Chapter 11: The OSI on X-25 application 142
11.1. The principles of a scenario 142
11.2. The 'OSI on X-25' scenario 143
11.2.1. Goal of the application 143
11.2.2. Screen composition 144
11.2.3. Designing the scenario's evolution 145

11.3. The implementation 153
11.3.1. User-application interaction 153
11.3.2. The application's classes 155
11.4. The 'Osi1' class 156
11.4.1. Instance variables 156
11.4.2. Factory methods 157
11.4.3. Instance methods 157
11.4.4. The actions 159
11.4.5. The classes used 160
11.5. The 'Osi1MTree' class 162
11.6. The 'OsiStack' class 162
11.6.1. Instance variables 163
11.6.2. Factory methods 164
11.6.3. Instance methods 164
11.6.4. The classes used 169
11.7. The 'Layer' class 169
11.7.1. Instance variables 169
11.7.2. Factory method 170
11.7.3. Instance methods 170
11.8. The 'Interface' class 171
11.9. The 'Pipe' class 171
11.10 Critique 172
11.11. Summary 173

Chapter 12: Conclusion 175

Bibliography 177

Appendices

Appendix 1 : Video application listing
Appendix 2 : OSI on X-25 application listing
Appendix 3 : Example of mainClass.m file
Appendix 4 : Example of individual makefile
Appendix 5 : Example of the environment's makefile
Appendix 6 : Screen copy of the Video and OSI on X-25 applications
Appendix 7 : RMG directory structure
Appendix 8 : Example of A.menu file

Chapter 1: Introduction

This thesis is concerned with the collaboration to a European project: the COLOS[1] project. Within this project and with the use of an Object-Oriented environment -RMG (Real-time Measurement Graphics)-, several programmers in different European universities try to realize some interactive simulation applications to be utilized as a help to Computer Assisted Learning -CAL- in universities.

The RMG environment which is used within the COLOS project was developed by the laboratories of the Hewlett-Packard company which sponsors the project. This environment is only available to educational institutions and lacks greatly of works or books explaining its use and its programming technique. This lack of references explains the fact that it is an environment difficult to learn and to master. This is why in this work we try to provide the future RMG programmers with a basic guide of RMG programming.

The university of Namur is part of the COLOS project and is mainly interested with the building of applications concerning telecommunications. We realized one of these applications which is presented in this thesis: the 'OSI on X-25' application. It concerns telecommunications using the ISO (International Standard Organization) OSI (Open Systems Interconnection) standard and concerns particularly the opening and closing of a connection between two machines.

This thesis is divided into three parts.

The first part concerns a presentation of Object-Oriented languages and RMG. We see what is hidden behind Object-Oriented programming (chapter 2) and give a specific example, the Objective-C language (chapter 3). In chapter 4, we look for the first time at the RMG environment and give a first description of it.

In a second part, we give a little guide to RMG programming. In chapter 5, we give a small bibliography useful for the one who wants to start programming in RMG. We see which is the process to be followed to create an application in RMG (chapter 6) and we give an introduction to the first useful RMG classes (chapter 7). This is followed by a presentation of new notions concerning the programming in RMG (chapter 9) and by a small illustration of what can be done with the environment: the Video application (chapter 9).

The third part concerns the illustration of telecommunication principles under RMG. We present the projects of the university of Namur concerning the COLOS project, that is projects concerning the representation of computer telecommunication in RMG (chapter 10). This is followed by the presentation of one of these projects which takes the shape of the 'OSI on X-25' application (chapter 11). We end by a conclusion and a critique of the COLOS project (chapter 12).

---

[1] : COnceptual Learning Of Science

FIRST PART: Object-Oriented languages and RMG

Chapter 2: What is Object-Oriented programming ?

Object-Orientation, nowadays fashion in the computer world. Everybody is using it, everybody has a different meaning for it, as says Brad J.Cox in [COX, 87] (p.29): "*It is hard to imagine two languages more different than Smalltalk-80 and Ada, yet both are sometimes called object-oriented languages. Others think of objects as primarily a way of expressing concurrency, and yet others as a way of organizing complex facts into hierarchies.*"

Nothing will be said about the implementation of Object-Oriented languages, mainly because it differs from language to language and we want to stay in a fairly general presentation. We try also to stay fairly objective and concise, swimming in a sea of opposite definitions and views on the subject[1]. We do not pretend to give here an exhaustive description of 'Object-Orientation'; our goal is only to introduce a few notions in order to facilitate the reader's comprehension of the next chapters.

So now, what is hidden behind these two words: Object-Oriented ? We will start by reviewing the principles of Object-Oriented programming (2.1) introducing the different basic concepts. Then we will try to describe the type of programming that has to be done with Object-Oriented languages (2.2). In a third point we will see which are the main differences between traditional programming and 'Object-Orientedness' (2.3).

2.1. Principles of Object-Oriented languages

Object-Oriented languages is "*a way to organize resources very much like the way we organize objects in everyday life.*" ([STEPSTONE, 88], p.1.1.). It is difficult to find a better general definition of an Object-Oriented language without entering details. That is why we will not give another one, but will just proceed on the subject.

The principles of Object-Oriented languages can be separated in two. A first point describes what an object is, with reference to internal data, methods and classes. The second point concerns the concepts of encapsulation, inheritance and messages.

[1] : Though we want to stay in a general presentation of Object-Oriented languages, the reader will notice that the syntax used in the examples and some other parts of this chapter are 'impregnated' of the Objective-C language which is the language presented in the next chapter. This is done so that the reader does not feel to desoriented while migrating to this next chapter

2.1.1. *Objects, data, methods and classes*

We see in this point what an object is, with reference to internal data, methods and classes.

The main starting point of Object-Oriented languages is that the part of the world that has to be represented can be seen as a set of individual objects. Indeed, if one looks at a square, it is composed of four equal edges and four right angles and each of these can be taken as individual objects. This is a very simple and basic view of the matter, but we think that it is a nice way to introduce the subject; so we will say that an object represents a certain part of the analyzed world. We can make here a distinction, for clarity, between external objects which are the objects of the real world and internal objects which are a representation of the reality in the computer's memory.

In real life each object has a certain number of properties or parameters. For example: a point has X and Y coordinates; a line has a certain length, etc; thus we could see, for the time being, internal objects as records with a certain number of fields. An object can also be changed, by modifying one or many of its parameters or properties. A point can be erased and put elsewhere; a line can be stretched. So in our system the objects must contain the parameters that characterize themselves and we must also have a way to change these parameters. We just introduced one of the great characteristics of Object-Oriented languages: one object contains BOTH the data and the 'procedures' to modify or access this data.

If we look at this in a more concrete way, an object[2] is a part of a system that contains some internal data, the static side, and some procedures, the dynamic side. The static side will be represented by a certain number of variables contained in the object itself; they will take different values depending on the evolution of the object during the system's execution; so they represent the STATE of the object. The dynamic side will be represented in the object by a certain number of procedures, called **methods**, whose goal is to internally manage the set of variables (print their value, change their value ...); so they characterize the actions that can be performed by the objects to change their state or to compute some values deduced from their state.

But object is a very abstract term. In fact, the Object-Oriented languages supply a way to create what is called a **class**; that is the code that represents the definition of an object, it is a "*technical term that will be applied in Object-Oriented languages to describe such sets of data structures characterized by common properties*" ([MEYER, 88], p.52). Thus a class is the description of a family of objects having the same characteristics and behavior. So now we will see a class as the definition of reality (precedently called object) in which the

---

[2] : Internal object, as described above

variables are called **instance variables**[3] and the set of methods, the class' **protocol**. An object will be an **instance** of a class, that is a structure occupying memory space during system execution. In this "*the basic idea is just that the class describes the structure of its instances, and the objects themselves contain the variable data*" ([STEPSTONE, 88], p.2.3.). To illustrate this a little bit further we can take the example from [COX, 87] (p.65): "*Betsy is a cow means also that Betsy is an instance of class cow*" "*Betsy is a cow but Betsy is also Betsy, the individual*" "*The instance, Betsy, is a tangible flesh and blood creature, but a class is an abstraction. Betsy, the instance, can moo, but cow, the class, can not.*"

As an example we can take an object 'Point', which has as internal data: 'x_coord', 'y_coord' (both reals), and has as method: 'location' -which returns the location of the point in a two dimensional space. **(Example 2.1)**

Now we can go a little further and say that a difference can be made between two types of methods :

- Factory Methods : methods used to create new instances of a class; methods "*which are the behaviors of the class*" ([STEPSTONE, 88], p.2-4).
- Instance Methods : methods used to consult or modify the state of the instances of a class; methods "*which are behaviors of an instance of a class*" ([STEPSTONE, 88], p.2-4).

2.1.2. *Encapsulation, inheritance and messages*

This point concerns the concepts of encapsulation, inheritance and messages.

Encapsulation is the right word to represent what we saw in the former point; it means that the consumer can **only** see and use the services[4] of an object that are available through an interface (see Figure 2.1, from [MASINI, 89]); the entire set of data and methods described in a class are only available to the supplier of the class. It is one of the most important points of Object-Oriented languages, **information hiding**. It follows that the object -the instance of the class- decides itself how to respond to a demand; the consumer asks and the object provides, all the implementation details are hidden. As well, all the internal data contained in the instance variables and the methods not accessible by the consumer are called the **private part**, mainly because the object ONLY has access to this data; the rest is called the **public part**[5]. All this allows the builder of the object to present "*cleanly specified interfaces around the services they provide*" "*How an object implements its actions and how its internal data is arranged, is encapsulated*

[3] : We will only talk here about instance variables to characterize the type of data one can find in one's classes; we can find a lot of other different 'types' of variables but this depends mainly on the language analyzed, and they are not common to all of them

[4] : We will see later that these services are mainly the use of certain methods

[5] : The methods accessible to the consumer through the interface

inside a procedural shell that mediates all access to the object" ([COX, 87], p.52).

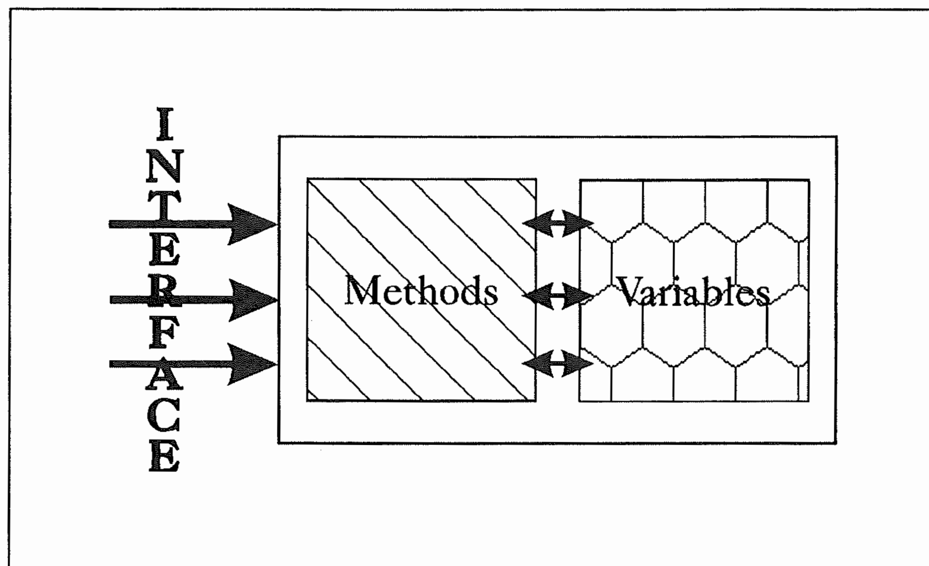


Figure 2.1 **The consumer can only see and use the services available through the interface.**

To illustrate this point, we can deepen example 2.1. A point is characterized by its coordinates, we will represent them by two variables: 'X_coord' and 'Y_coord'. This internal data is not available directly to the consumer, as shown in Figure 2.2 (based on [COX, 87], p.52).

The consumer can access these variables and thus their value through two methods: 'location' -that returns the value of 'X_coord' and 'Y_coord'- and 'changePoint:' -that changes the values of 'X_coord' and 'Y_coord' to the values given by the consumer. **(Example 2.2)**

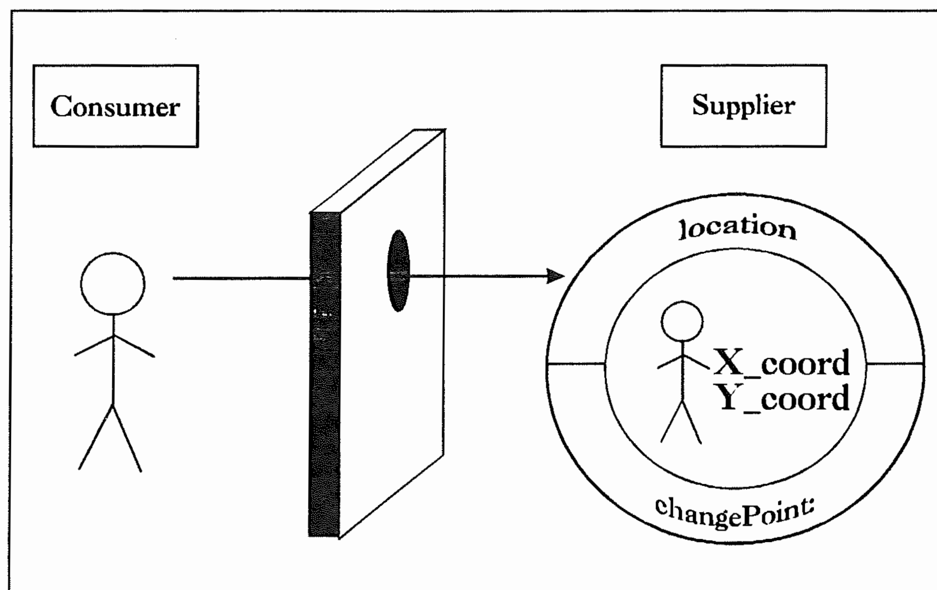


Figure 2.2 **Consumer can access variables only through methods available to him.**

Now that we have seen what encapsulation means, we can ask ourselves if a class on its own is really worth-while and sufficient? Of course it is not, otherwise one class would be representing a whole system, without differentiation between individual objects and the whole concept of Object-Orientation would be useless. That is why in an Object-Oriented programming environment one is able to create many classes representing a lot of different things, but also, and this is the important point here, one can create a **hierarchy** of classes. For example a square is a particular type of polygon, we can thus create two classes, a first one quite general named 'Polygon' -it is a shape with a certain number of sides-, a second, more precise, named 'Square' -a polygon with four sides, all of the same length, and four right angles. This second one will be called **subclass** of Polygon, the **superclass**. But Polygon could also be a subclass of another class 'Shape', and so on ... (see Figure 2.3)

It does not make a lot of sense to talk about a hierarchy of classes, of subclasses and superclasses, without talking about **inheritance**; in Object-Oriented languages one goes with the other. Inheritance is a RELATION of interdependence established between some classes; it means that within a specified hierarchy, a subclass A of a class B inherits all the features of B, the instance variables and the methods, this principle being TRANSITIVE. So a class will be an extension, a specialization or a combination of others; we will just mention here that some languages enable simple inheritance -one class inherits directly the characteristics of a single other class- as well as multiple inheritance -one class inherits directly the characteristics of several other classes. Inheritance provides the ability to create classes that will automatically model themselves on other classes, just by specifying their difference(s) from the existing ones. Says [COX, 87] (p.69): "*Without inheritance, every class would be a free-standing Unit, each developed from the ground up.*" "*Inheritance makes it possible to define new software in the same way we introduce any concept to a newcomer, by comparing it with something that is already familiar.*"

We can illustrate this by starting again with our polygon and square. Let us say we will build a class 'Polygon' which will contain one instance variable: 'numberOfSides' -containing the number of sides of the polygon-, and one method: 'howmanySides' -that returns the number of sides of the polygon.
Let us also build another class named 'Square' with two instance variables: 'lengthOfSide' -containing the length of one side- and 'stringSquare' -containing the value: "This is a Square"-; and a method: 'changeLength:' -that changes the length of one side. We will also say that Square is a subclass of Polygon and so inherits of all the characteristics of Polygon. This means that an object Square -an instance of class Square- will also have a variable 'numberOfSides' and will be able to consult the value stored into this variable through the method 'howmanySides'.(see Figure 2.3) **(Example 2.3)**

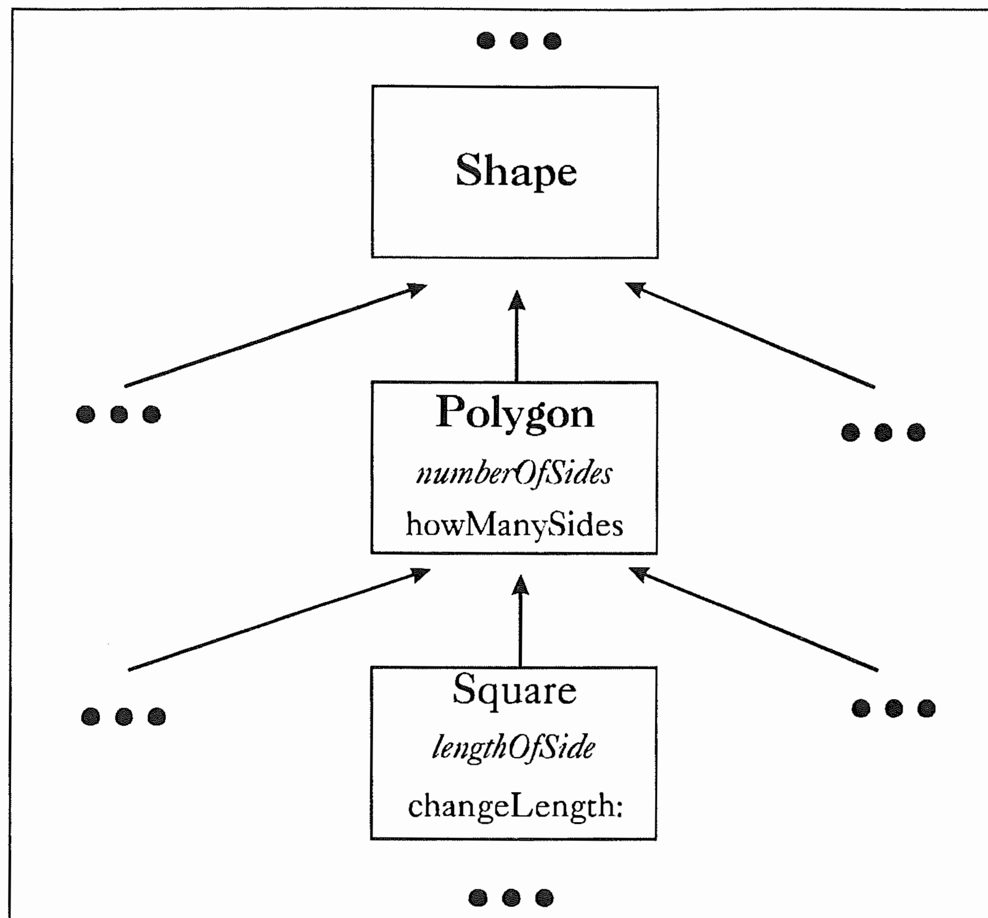


Figure 2.3 inheritance graph for example 2.3

So inheritance is a nice way to avoid duplication in code writing and to nicely take advantage of already existing code. We can still say two things on inheritance; the first is that a kind of redefinition is available which means that there is a possibility to redefine 'features' owned primarily by one of the superclasses. For example, in our class Square we could redefine the method 'howmanySides', so that it returns the number of sides AND print the value of the instance variable 'stringSquare'. This will be done by first 'calling'[6] the method 'howmanySides' of the class Polygon and then printing the content of the instance variable 'stringSquare'. As said in [PUGH, 90] (p.18): "*A specialized class inherits all the attributes and operations of the more general class and may:*

- *have additional operations*
- *have some modified operations*
- *override existing operations*
- *have additional data attributes.*"

The second thing we can stress about inheritance is renaming; by this we mean that methods of superclasses can be renamed in the subclasses, one of the purposes being clarity.

[6] : This 'calling' will be done by message-passing, which is explained in the next point

There is at least one last concept that so far we failed to talk about, that is 'messaging' or **message-passing**. Behind this stands the key to several questions: how do objects communicate between themselves? How does a consumer have access to services through an object's interface? (see Figure 2.2) and many more. To explain this we must first bear in our minds that an object controls the access to its state. After this we can state that a message is in fact a request for an object to 'execute' one of its methods; it specifies what kind of operation has to be carried out but does not say how to perform it, this is indeed hidden inside the object itself.

A **message-expression**, usually called message, can be divided in three different parts :

- the name of the object one wants to access (in other words the receiver of the message);
- the name of the **selector** (or name of the requested method in the object);
- a number of arguments, if needed.

One thing must be said about the name of the object. If the message has to be sent to a class -so in this case we are talking of sending a message that concerns a factory method- the name of the object will be the name of the class; this name is indeed unique, each class being uniquely present in the system. This leads to the conclusion that a class is itself an object, unique in the system. For example if one wants to send a message to class Square, the message will be :

Square <selector> <arguments>

In the case of a message being sent to an instance, one has to know the identifier of this instance. This is possible due to the fact that when an instance is created the system returns the address of this instance to the user which has asked the creation[7]. The user has just to store it in a variable and from then on is able to access the instance. We can illustrate this by extending example 2.3. If we want to know the number of sides of an instance of class Square, we will issue to this instance the message (without particular attention paid to the syntax) :

aSquare howmanySides[8]

where 'aSquare' is a variable containing the address of the instance of the class 'Square'.
This is indeed correct by inheritance and it will return the value 4. If we want to change the length of the sides of this square to 20, we will issue the message :

aSquare changeLength: 20

[7] : An instance is created using a factory method called an initialization method, this will be explained in detail later on

[8] : 'aSquare' is written following a naming convention, which specifies that each word except the first one composing a variable name will take a starting capital letter

Supposing that the argument must be a numeral. **(Example 2.4)**

By what we just said, we are now able to give a simple definition to our interface of figure 2.1 :

An interface is a set of messages to which an object is able to respond by selecting and executing one of its methods.

When an object receives a message[9], it determines itself how to carry on the operation; it responds to the message by first choosing the method that implements the selector specified in the message, executes this method, and then returns control to the caller; even in some cases, an object can accept OR reject one's message, depending on the fact that it recognizes it or not. To explain this a little bit more, let us say that each class structure -that is in a run-time system- has a table listing the names of each method defined in this class and an address of a function that implements the behavior. When an instance receives a message, the messaging 'system' finds the class by following a link from the instance of this class to the class itself[10] and searches the table to find a correspondent to the selector contained in the message. If this fails, the search continues through the tables of each class composing the inheritance graph until it finds something or nothing at all, in this last case the message-passing fails and an error occurs. So we can see that objects have, through message-passing, a very special kind of autonomy, but also that the interactions between objects are carefully controlled; all this furnishes a very powerful tool in the whole.

2.2. Programming with Object-Oriented languages

Now that we have more informations on Object-Oriented languages and what they imply, we are going to look briefly at the programming itself. This will require two points; the first one will discuss the questions of what is really an Object-Oriented program, how it is composed ...; the second will mainly concern the problem of finding the right objects for the right applications.

2.2.1. *Object-Oriented programs*

We are now asking ourselves how it is possible to create a tangible and running program with all the raw material we presented in point 2.1. An important thing to say is that in an Object-Oriented language, creating a program is first analyzing the real world to divide it into objects which will be represented in the system. Then after this one must create the program. This program will in fact be based

[9] : Here we are talking about a combination of a selector and zero or more arguments

[10] : This is possible due to the fact that the instance of a class contains a pointer to the class itself

on a certain number of classes, which when instantiated will interact with each other by sending messages, this to execute a certain number of things depending on the current situation. We could describe programming in such an environment, in three different steps[11] :

1) Creating classes
2) Creating instances of classes
3) Specifying sequences of message exchanges among objects

After designing formally the different new objects one will use in a system, one must create a class for each of them (step one). Creating a class is going through multiple phases, which will be described later, but mainly we can introduce the subject by saying that a class is composed of:

- a name, which will identify it among the other classes of the system;
- a declaration part which specifies the superclass(es) of the current class; so a definition of a part of the inheritance tree;
- a declaration part for the new instance variables that are not inherited from the superclass(es);
- a 'method part' which will declare and define all the new methods of the class.

Here we will not enter the implementation and syntax details because they all depend highly on the type of language used; but as an example, we could extend Example 2.4 by giving a 'definition' of the class Square, following the structure given above :

```
Class Square;
Superclass Polygon;

Instance Variables Definition Part:
    {
    Integer lengthOfSide;
    }

Methods Definition Part:
    {
    changeLength: (integer)aLength
            {
            lengthOfSide = aLength;
            }
    }
```

Each method name is defined in the Methods Definition Part, here 'changeLength: (integer)aLength', where 'aLength' is a formal parameter which will give the new value to be put in the instance variable 'lengthOfSide' defined in the Instances

[11] : Note that steps 2 and 3 are not specially sequential

Definition Part. This method name definition is followed by the code itself, here 'lengthOfSide = aLength;'. **(Example 2.5)**

The second step is creating instances of classes. So how can we create these instances? We can divide this problem in two, the first part concerns the matter of creating an instance of a particular class, while the second part deals with the problem of knowing how instances can use other objects.

Concerning the first part, the general principle is that each class must contain an **initialization method**[12]. The effects of this method is that a 'live' representation of the class -an instance of the class- is created in memory with default parameters. Also it returns the address of the new instance which will be stored in a variable; this will indeed enable the user to send the instance messages. This method can be particular for each class[13] but in any case refers through inheritance to the initialization method of the root class of the hierarchy tree. So when one tells the system[14] to run a class, it will send a message -which contains the selector of this initialization method- to this class.

The elements involved so far are a class present in the system, a dedicated method in this class, and a possible message sent to this class to execute this method. Again here, we will not go further due to the fact that implementations of this mechanism, of memory managements, of the exact 'location' of the classes... are different from language to language.

So from now on we have in memory an instance of the class we want to 'run'. But, second part, what does this instance do? The first thing is it can 'play around with itself', so executing its methods in a certain order, according to the instance itself, without anybody asking anything, so far this is not very useful. The second thing, it can 'play' with other objects by creating them and coping with them in a certain defined way: sending them messages to reach certain informations or to change their state, etc. Thus it underlies that an existing object can act dynamically on the system by sending classes their initialization message; this will change the state of the whole system by incorporating new instances of classes among the already existing ones.

To explain this a little bit more, we can extend our example 2.5 by saying that our class Square, to be 'accepted' in our system must contain a new instance variable called 'aSquare' -which will contain the address of the instance- and a method called 'new', defined in the Methods Definition Part:

```
...
Instance Variables Definition Part
   {
   Pointer aSquare;
   Integer lengthOfSide;
   }
```

[12] : This method is a factory method

[13] : This means that it can do particular things depending on the class' purpose

[14] : By selecting an option in a menu, selecting a name in a list ...

```
Methods Definition Part
   {
   new
          {
          aSquare = Polygon new: 4;[15]
          }
   ...
   }
...
```

This method 'new' when executed[16] :

- creates an instance of Square in memory by allocating a certain amount of memory to store the data of the instance; this will be done by invoking the new method of its superclass -'Polygon new: 4', where 4 is the number of sides of the new polygon-;
- can ask the execution of the method 'changeLength:' with a parameter to effectively change the state of the object:

```
new
   {
   aSquare[17] = Polygon new: 4;
   aSquare changeLength: 10;
   }
```

- can send a message 'save: lengthOfSide' to an already existing object in the system -called 'aFile'-; this will result in the object aFile storing the value of 'lengthOfSide' on a disk[18] :

```
new
   {
   aSquare = Polygon new: 4;
   aSquare changeLength: 10;
   aFile save: lengthOfSide;
   }
```

- can do any other thing necessary at the instantiation of the class, like variable initialization for example.

It is thus possible to create an instance of class Square from scratch by sending it an initialization message that will have as effect the execution of the class' Square 'new' method. This is what is illustrated in figure 2.4 with the black 'new arrow' which goes from the system to the class Square and which has as effect the creation in memory of an instance of Square. **(Example 2.6)**

---

[15] : Note that this syntax was chosen for clarity and is not to be found in every language

[16] : After sending a message of the type 'Square new' to the class Square

[17] : aSquare is here a variable that will contain the address of the newly created object

[18] : To make things easier, we take the liberty of assuming that the object identified by 'aFile' already exists in the system

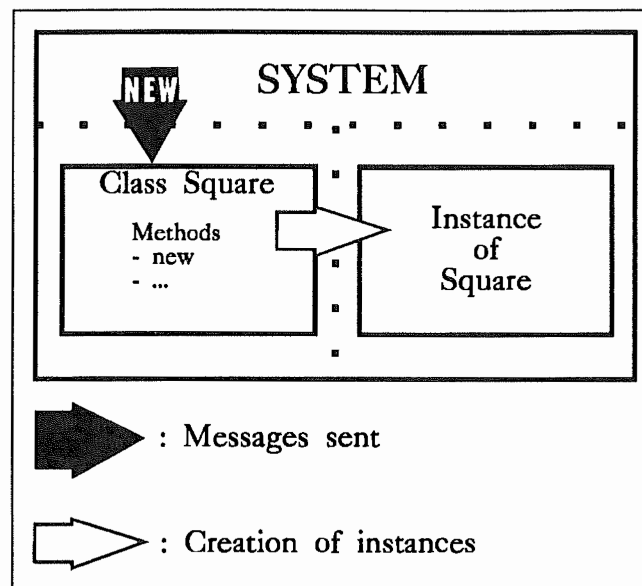


Figure 2.4 Creating instances of classes

The third and last step is about message exchanges. How can one specify a sequence of messages to be executed? One solution is to program this sequence in a method to be executed later. So when the method is called, each message, one after the other, is sent to the right object; this is what we call the static way. The second solution is the dynamic way. When an instance of a class exists in a system, the user[19] can send it any 'legal' message[20] through a special editor, browser, list of methods name ...[21]

2.2.2. *Finding the right objects*

To the problem of finding the right objects to describe a defined problem or to describe accurately a part of the real world, we will not give direct solutions or 'magic tricks', but just a few suggestions among which the reader will have to choose. But before directly considering the point of finding right objects for right applications, we must again ask ourselves what programming is all about? We can reply by naming three different purposes for which we use software:

- obtaining certain answers about questions over the real world;
- interacting with the world;
- creating new world representations.

In this prospect, the software must be based in any case on an analysis of this world in terms of interesting points relevant to the application. In this optic,

[19] : By user, we mean any person that has to use instances from a terminal

[20] : Message accepted by the instance

[21] : Note that in certain languages, the user has to build a main program in order to use his classes

understanding **Object-Oriented modeling** is quite simple: *"the world being modeled is made of objects" "and it is appropriate to organize the model around computer representations of these objects"* ([MEYER,1], p.51).

Usually, in this type of programming, one does not ask oneself a lot of design questions, a programmer just picks up the external objects of the real world. So says [MEYER,1] (p.51): *"The software will simply reflect these external objects" "... just use as your first software objects, representations of the obvious external objects"*. This is the external objects optic.

A second optic, yet less creative, is the existing classes view. In this case, the programmer tries to find what he looks for in the existing libraries of classes. If one or many of the classes do not meet exactly his needs, adaptation is possible. This adaptation can be done by directly changing the original class, but unless there is a big mistake lying in the code, it is preferable not to do it; let us think of the interesting mechanism we dispose of, inheritance, and let us use it to add or change some methods or variables by defining a new subclass. This enables the class to be undisturbed, thus preserving its clients against 'misfunctioning'.

Every programmer should document his classes sufficiently to explain their existence in a system so that every other person could understand why a class was created, in regard of the problem. This can lead another programmer to apply the same techniques to his own problem. Unfortunately classes are not always well documented, thus another optic is the evaluation one. In the first place, we can analyze a bunch of classes and the world they represent, and try to discover the underlying 'philosophy'. When it is found, we can just try to apply it to our case.

We can analyze existing classes or the new founded ones, on a theoretical design point of view : 'Are the interactions between two classes not too high?'; 'Is the application domain of one class not too large?'; 'Is the messaging not too complicated?'; etc... If replies can be found to these questions, a great pace can be made towards a better design of the considered applications. *"The rule 'criticize and improve existing designs' is not itself a solution to the design problem. But good Object-Oriented design, as good design in any discipline, must be taught in part by apprenticeship and experience."* ([MEYER,1], P.327)

Usually, something should only be formalized into a class if it describes, in the real world, some objects characterized by interesting operations, typical particularities. But everyone must bear in mind that in some cases external objects or facts must be represented or must not be, depending on the type of application and the meaning that they yield in the environment.

The beginner must particularly beware of two nasty habits:

- designing unneeded classes
- designing classes that are not classes.

The first habit depends mainly on the experience of the programmer and the type and complexity of the applications; the solution is getting more and more used to this type of programming, by generating code and testing it on a design and functional point of view. Designing classes that are not is often the case

when someone builds a class around a routine, a procedure, without a solid object's base under it. As says [MEYER,1] (p.328): "*In contrast with a routine, a class should not DO something but offer a number of services (features) on objects of a certain type.*" Thus each class has to correspond to "*a meaningful **data abstraction***" ([MEYER,1], p.328) to be valid.

We just saw that finding the right objects for the right applications is not so easy, there is no magic trick. The only solution is habit, training and practice.

2.3. Differences between tradition and 'Object-Orientedness'

Now that we know a little bit better what Object-Oriented languages are all about, what makes the difference between them and a traditional language? Why is it that a lot of people in the programming world are venerating this new kind of programming?

We will try here to underline a few of the main topics that make Object-Oriented languages or techniques different and in some cases more advantageous[22]. To introduce the subject, we will start by looking at language implementation details; let us first recall that in many cases, in the traditional type of programming, variables may be local to a procedure, that procedures pass arguments of various types - strings, numerals...-, etc; in Object-Oriented programming the building block is the object, an autonomous piece of information which contains some local data and local ways to modify it. Further on, these blocks do not just interact with each other by passing arguments -like in structured programming- but the local methods enable the mechanism of messaging between objects. As says [TELLO, 89] (p.11): "*objects resemble smaller computers within the host computer, each with its own data and code areas*"; or [DUFF, 90]: "*Object-Oriented technology improves software systems because it facilitates better factoring of functionality and related data than do traditional structured-programming techniques.*".
Through the few ideas just suggested, we can deduce two important statements :

- Object-Oriented programming is different because of its MESSAGING MECHANISM, which enables the communication via messages, referring to methods reachable through an interface unique to each class;
- Object-Oriented programming is different because of its 'DESIGN PHILOSOPHY', an object is designed around its data and incorporates fully the methods to modify it; this is the concept of encapsulation or data abstraction, it means that one does not have to know the implementation of the object to ask it something; it all alone decides how it will execute the demand, depending on its physical implementation. It is very important to realize this at a design stage, because all the 'thinking' will be made on a data point of view -identify the objects/ identify inter-relationships between

[22] : Note that Object-Oriented languages have also disadvantages, but they do not enter the scope of this part

them/ ...- this has to be opposed to a function point of view -identify the functions/ the inter-relationships between them/ ... .

An other concept that makes the difference is INHERITANCE. This makes the design of programs much more modular and distributed than the one usually incorporated in traditional programming. Indeed the programmer disposes of a complete set of existing classes -distributed in various libraries-, a complete rewriting of the code is thus useless; it is quite enough to write classes for the new concepts present in the problem and inheritance will do the rest. So the fact is that the represented part of the world is MODULARIZED -structured- into objects EASILY MODIFIABLE without too much interaction with other objects, due mainly to inheritance, redefinition, etc. So we are talking of EXTENSIBILITY here.

An other advantage is COMPREHENSIBILITY. Through the modularization of systems, the encapsulation, the hidden implementation details, etc, code is more easily read, understood. In the same way GENERALITY is yet an other point, indeed the different components to be built have to be very general to enable a future reusability. This is not the case in some 'usual' programming languages. Thus the overall concept developed here, is the one of reusability -being defined by its behavior, the object is easily reachable through its interface-, which is one key concept of Object-Oriented programming and which is not present at this stage in the more classic languages and types of programming. As says [DUFF, 90]: "*OOP* - Object-Oriented programming- *improves code reuse by using less complex, loosely coupled, highly cohesive components.*". You do not bother on how it is done, you just bother on what you want to do.

We can still go further and state that 'Object-Orientedness' provides the ability to handle complexity in a more transparent manner. Indeed an object view of the world seems to be easier to understand and to formalize than a structured and procedural one, it gives the programmer more liberty and more possibilities to work on big projects. On the other hand using an Object-Oriented 'technique' does not mean better design, one can do awful coding and designing in such languages, so in the whole it is something that takes time to acquire but that makes life a lot easier when mastered.

On a more technical point of view, we will stress three more points:

- one can have as many instances of a class present in a system at the same time as memory will allow, without them interfering with each other. This is surely not a prerogative of these types of languages, but it is a very important factor.
- inheritance enables to deal with a greater and greater specialization of functions, just by adding the new features, the rest is inherited. Thus we face LIMITED REDUNDANCY in coding.
- a uniform interface can be provided over the widest possible range of object types. For example, a method 'divide' can be implemented for types of

object like Integer and Real, so the same name can be found but the code and the behavior are different.

We could not possibly go through all the differences between Object-Oriented languages and the others, so we tried to underline the major ones. But to conclude this point, let us say that Object-Oriented programming is not in the whole a revolution, but an evolution. 'Object-Orientedness' is not 'a panacea', it has benefits but they are not for free. To exploit it at the maximum, needs a significant organizational support on a design point of view but also for the apprenticeship of this type of programming. It is a whole philosophy to adopt and to master, and this is even more important than language or application design problems. That is why the few advantages and differences brought in the light above are vital to understand.

2.4. Summary

We tried in this chapter to produce a simple and general overview of what is an Object-Oriented language. We started with a general presentation of the main and most important principles of 'Object-Orientedness' :

- objects, data, methods and classes
- encapsulation, inheritance, messages.

By analyzing different ways to find the 'right objects', we exposed what a program was in these kind of systems and in a very simple and basic way how to build one. The steps we followed were :

- creating classes
- creating instances of classes
- specifying sequences of message exchanges among objects
- analyzing briefly Object-Oriented modeling.

We asked ourselves, after that, what made these languages really different from the traditional ones. We underlined a few differences :

- the messaging mechanism
- the design philosophy
- the concept of inheritance
- the generality of the components
- the handling of complexity.

To strengthen the preceding point, we stressed a few advantages of Object-Oriented languages, among which the main ones were :

- the data abstraction

- the extensibility
- the reusability.

Chapter 3: A specific approach: Objective-C

We will give here an example and illustration of Object-Oriented language, by analyzing Objective-C[1]. The goal of this chapter is not to give a complete and thorough description of Objective-C but to give the reader sufficient bases to understand the following chapters.

Objective-C is a C based Object-Oriented language designed by Brad J. Cox and distributed by the Stepstone corporation (Productivity Products International Inc.).

We will proceed in three steps; the first of which will present the main topics of the language (3.1), the second will show briefly how it works (3.2) and the last one will give an example of an Objective-C program (3.3) to illustrate what will have been said in the two preceding points. We assume from now on that the reader is accustomed to the C language and therefore we will not explain the C code used in this work.

3.1. Main topics of the language

Objective-C implements the style of Object-Oriented programming used in Smalltalk-80 as a set of extensions to the C language. Thus it is a good example of how 'Object-Orientation' can be introduced in other more conventional languages.

Objective-C is a tool to write applications involving Object-Orientation and all the features of C itself.

It brings two things to C:

- a new data type
- a new operation.

The new data type is the object[2] and the new operation is the message expression; apart from that, Objective-C works just like a usual C compiler. As says [COX, 87]: "*Objective-C is a hybrid language that combines the object-oriented features of Smalltalk-80 with the C language.*" "*Since it is hybrid, it allows the programmer to use object-oriented tools when they suit the task at hand, yet all of C remains convenient for when hand-tools are sufficient.*"

One important thing to see is that Objective-C is simply a 'supplementary' layer of C, which enables construction of classes and objects, without touching to the C mechanisms. For example, variable types are not integrated in the language, an integer variable is not an instance of a class 'Integer', on the contrary of Smalltalk-80.

---

[1] : Note that everything said here applies to Objective-C version 3.3

[2] : This new data type is in the language the type '**id**'. A variable of type id is in fact a pointer which is NULL or represents the address of an instance of a class

We start now with a short introduction on the language. This is followed by a description of the Objective-C syntax and a description of the main program that has to be built. Finally we give a small syntax summary.

3.1.1. *Introduction*

This part will describe in short the main characteristics of the language.

A class, in Objective-C, is the description of informations related to a group of similar objects; this description is done in terms of methods and instance variables and contained in a file called **Class Description File**. Each class, in this language, has only a role of description; at run-time and after their compilation, each one will be represented in the system by a single particular object called a **factory object**, that will help the system building the instances.

The second thing to be said concerns the new type added by Objective-C , id, which is in fact a pointer to a data structure representing the object in memory. Into each class description file, Objective-C generates a 'typedef' statement which defines it in terms of another well-known C type: pointers to structures. Each instance of a class used in a program will be represented by a variable of this type, which will contain the instance's address in memory. An instance identifier -variable of type id representing an object- enables manipulation of this object in a message expression, which is the only legal operation on an id.

In Objective-C, the instance variables represent the data -called **private data-** composing the **private part** of an instance of a class; this means that it is only accessible by the instance itself -principle of encapsulation- and is different from instance to instance; to oppose to the **shared part**, which is the part common to all instances of a class -this concerns the methods. But as says [COX, 87] (p.53) concerning the private part: "*it is also true that it is always possible to bypass the Objective-C machinery to access an object's private information directly.*". The methods are usually represented, in Objective-C, by a kind of C function, able to contain usual C coding as well as 'message sendings' and as C functions, these methods are able to return a value. This will be described later in further detail.

Inheritance appears in this language as a simple inheritance, in short one class can only have one parent. It has to be noted that the root of Objective-C hierarchy tree[3] is a class called '**Object**'.

When one has created his own classes, one needs to build a main program in Objective-C that will enable the different classes to be instantiated and the different instances to interact with each other.

---

[3] : Called also hierarchy graph

3.1.2. *The syntax*

The syntax of an Objective-C program is slightly different from a normal C program. But before starting its description we will give a number of conventions useful for the comprehension of the continuation. A first one concerns class names; if they are composed of several words put in one, each of these words will have its first letter in capital (example: MyNewClass); the same is applicable for each method name[4], except for the first word composing this name (example: myNewClassMethod). A second convention concerns the name of a class source file[5]; it will be composed of the name of the class followed by '.m' (example: MyNewClass.m). Concerning method's return types and values, if a method has anything useful to return, it should return it, otherwise it should return the address of the receiver.

A class is defined in a Class Description File and is composed of several parts :

- the definition of its name and the name of its superclass;
- the definition of the instance variables;
- the definition of the factory and instance methods;

To that, one must add possibly the traditional C code with definition of functions, of local or global variables, etc.

The description of a class begins with the definition of its name, following the symbol '=' and followed by a colon. Immediately after comes the superclass name, from which the new object will inherit everything; this is followed by a list of names called the Message groups. In the case of Objective-C, the result is :

= <class name>: <superclass name> (Message Groups)

where '**Message Groups**' is a list of file names. Each of these files contains informations concerning the return types of methods. Objective-C provides three message groups which concern the methods implemented in the classes provided with the language :

- '**Primitive**', which concerns the methods implemented in primitive classes;
- '**Collection**', which concerns the methods implemented in all the collection classes;
- '**Geometry**', which concerns the methods implemented in the graphical classes.[6]

---

[4] : Note that we use the name 'selector' in the case of a message and the name 'method name' in the case of a method definition

[5] : Also called Class Description file

[6] : For further informations on these classes, the reader will consult [STEPSTONE, 88]

As the user writes code, he has to use his own message group; he only specifies it and the system will create it if it is non-existent. It is better to always specify the message group 'Primitive', as one probably always uses messages from the built-in Objective-C library. 'Message Groups' is of type:

<User Message Group name> *[*, <Other Message Group>*]**[7]

For example, if we define a class 'MyObject', we will have:

= MyObject: Object (MyGroup, Primitive)

where 'MyObject' is the name of the new class and 'Object' its superclass.

(Example 3.1)

Note that all the principles exposed here and below will be further illustrated in point 3.3. Note also before going further that C comments are allowed, as the Objective-C ones, introduced by //.

Just afterwards comes the definition of the private data or instance variables:

```
{
    <type> <variable name> [, <variable name>]*;
    <type> <variable name> [, <variable name>]*;
                ...
}
```

For example:

```
{
    int aNumber, anOtherNumber, c, d;
    id  anObjectAddress;
}
```

(Example 3.2)

As said earlier, in Objective-C, a method is a sort of C function in which can be combined some C programming and Object-Oriented messaging. As said in the previous chapter, there are two types of methods: factory methods and instance methods. The first ones are methods describing the behavior of the class and designed to create an instance of this class, they are prefixed by a '+'; the second ones are methods used to consult or modify the state of an instance of a class and are prefixed by a '-' in Objective-C. Methods can have zero, one or many arguments. Note that it is compulsory to specify the type of each argument

[7] : From now on, * means that the preceding item can be repeated; *[]* means that the items between the square brackets are optional, the reader must not confound the latter with the message-sending syntax defined later

just before its name -between parenthesis-, like in C, except if the argument is of type 'id'.

A method name can be composed of a unique keyword :

- <keyword> [8]

For example :

- myMethod (**Example 3.3**)

A method name can have also one keyword with one argument, with the type of the argument preceding its name :

- <keyword>: (<type>) <argument>[9]

Examples of method names as defined above, are :

- numberOfSides: (int)aNumber
or
- objectPointer: anAddress

In the first case, the argument passed to the method is of type integer; in the second case, the argument is of type id. (**Example 3.4**)
It is very important not to leave blank spaces between a keyword and the colon in this keyword, if any. Indeed, this colon is entirely part of the keyword and changing its place will have as result the system interpreting the method name completely differently. For example 'myMethod:' is entirely different of 'myMethod :'. On the other hand one can insert as many blank spaces as wanted between a keyword and its argument.

A method name can also have any necessary combination of keywords and arguments :

- <keyword>: <argument>: <argument>
or
- <keyword>: <argument> <keyword>: <argument>
etc, ...

For example :

- coordinate: (int)x: (int)y
or

---

[8] : The starting minus could be a plus, depending on the method desired; this will be true for each syntax illustration

[9] : The type of the argument is not compulsory if it is of type 'id'

```
- coordinateX: (int)x coordinateY: (int)y
etc, ...
```

where in the latter the method name is 'coordinateX: coordinateY:'.

(Example 3.5)

The methods are defined like C functions; if the type of the returned value is different from the type id, it must be specified before the method's name, this can be applied to each type of method name definition given above. A method definition is thus :

```
- [(<return type>)] <method name>
  {
  Method's Body
  }
```

where *'Method's Body'* is C code with possibly Objective-C messaging. To illustrate this, we can extend the last example of example 3.5, specifying that the two arguments passed have to be stored in two local variables, 'varX' and 'varY', and among other things their sum as to be returned. This gives a very simple example :

```
- (int)coordinateX: (int)x coordinateY: (int)y
  {
    int varX, varY, sum; /* line 1 */

    varX = x;            /* line 2 */
    varY = y;            /* line 3 */
    sum = varX + varY;   /* line 4 */
    ...
    return sum;          /* line 5*/
  }
```

The C variables local to the method are defined at line 1. The arguments are stored in the local variables, at lines 2 and 3. The sum of the two variables is 'done' at line 4. This sum is returned at line 5.

Now that we saw a little bit further the syntax of a method definition, we can say that method names composed of a single keyword and no arguments are usually used to identify methods which are designed only to return the value of specific instance variables or to perform an operation on instance variables without changing them to new external values[10]. **(Example 3.6)**

The main differences with a C function are that the name of the method need not be unique across several classes, that methods are not called by name but indirectly by messaging, that methods must address an additional space which is the private data inside the object.

[10] : By new external values, we mean a value given from the outside of the instance by the user of the method, by mean of an argument

To send a message, one just specifies the receiver of the message and then the selector with the right arguments :

```
[<receiver> <selector>];
```
[11]

In the case of a class, '<receiver>' is replaced by the class name which identifies the class uniquely in the system. In the case of an instance, '<receiver>' is replaced by the instance address.
For example :

```
aSum = [oneMyObject coordinateX: 5
                    coordinateY: 10];
```

where 'oneMyObject' is a variable of type id containing the address of an instance[12] of the class 'MyObject' to which the message must be sent; where 'coordinateX: 5 coordinateY: 10' is the selector.
In this case, the value returned by this method 'coordinateX: coordinateY:' after execution, is stored in a variable 'aSum' that can be a local C variable, a global C variable or an instance variable. (**Example 3.7**)

The following is also valid in message-passing :

```
[[<receiver> <selector1>] <selector2>];
or
[[[<receiver><selector1>] <selector2>]<selector3>];
etc, ...
```

In this case, the return value of the method identified by 'selector1' must imperatively be of type id and must be the address of the instance to which one wants to send the message composed of this address and 'selector2'. It is in fact equal to :

```
rec = [<receiver> <selector1>];
```

followed by :

```
[rec <selector2>];
```

where 'rec' is an id variable which receives from the execution of the method identified by 'selector1', the address of an instance to which the message '[rec <selector2>]' is sent afterwards.

---

[11] : Note that in this case and for every messages, the [] are integrally part of the syntax and are thus obligatory, as for the items between them. Note also that the semicolon is obligatory in Objective-C, following the same rules as in C

[12] : We will see later that this address is the value returned at the creation of the instance of the class 'MyObject', by the initialization method.

So this syntax is valid only in the case of the 'selector(n-1)' method return value being an id, so that 'selector(n)' can be sent to the instance identified by this id.
For example :

```
[[oneMyObject myMethod]
              coordinateX: 5 coordinateY: 10];
```

where the method identified by 'myMethod' performs a set of actions on the instance identified by 'oneMyObject' AND returns a value of type id which identifies an instance to which will be sent the selector 'coordinateX: 5 coordinateY: 10'. **(Example 3.8)**

One last thing we want to say about the syntax is that each class definition will end with the sign: '=:'.

3.1.3. *The main program*

When the user has defined several classes, he has to build a **main program** to make possible instances of these classes interact with each other. The overall aspect of this program is very much the one of a traditional C program, except for particular details. Note that a simple example will also be given in point 3.3.

A program starts with an equal sign followed by the definition of the message groups that have to be used in the program :

= (Message Groups)

The message groups that have to be specified are the ones used in the different classes that will be used in the program.
For example :

= (MyGroup, Primitive) **(Example 3.9)**

Then comes the definition of the body of the main program itself, which is done exactly in the same manner as for a C program. As said in [STEPSTONE, 88] (p. 3.10): "*...the only difference between a C program and an Objective-C program is that messages are sent in Objective-C programs.*". One remark that has to be made is that the methods must be made available so that the system knows which classes are used. This is made with a C 'extern' declaration, using the id type, as follows :

extern id <name of class> *[*<other class name>*]**;

For example :

extern id MyObject; **(Example 3.10)**

The last things that must be enclosed after the definition of the body of the main program are two clauses. The first one is '**@classes**()' and the second one is '**@messages**()'. These clauses are necessary to the compiler so that it can combine the classes and the messages used, to build tables so that it can reference them. One just has to add these two lines after the program :

```
@classes (List of used classes);
@messages (List of message groups);
```

where 'List of used classes' is a list of all the class names used, separated by commas and enclosed in parenthesis; 'List of message groups' is a list of all the message groups used, also separated by commas and enclosed in parenthesis.
For example :

```
@classes(MyObject);
@messages(MyGroup, Primitive);   (Example 3.11)
```

Note that the file containing these two clauses and thus the main program must be compiled AFTER all the classes used have been compiled.

3.1.4 *Syntax summary for a class definition*

```
= <class name>: <superclass name> ( Message Groups)
{
  <type> <variable name> [, <variable name>]*;
  <type> <variable name> [, <variable name>]*;
}

+ [(return type)] <method name>
 {
 Method's Body
 }
...

- [(return type)] <method name>
 {
 Method's Body
 }

=:
```

where 'Message Groups' is :

```
<User Message Group name> [, <Other Message
                                      Group>]*
```

'<method name>' is :

<keyword>
or
<keyword>: (<type>) <argument>[13]
or
- <keyword>: <argument>: <argument>
or
- <keyword>: <argument> <keyword>: <argument>
or any necessary combination of keyword and arguments.

And '*Method's Body*' is C code with eventually Objective-C messaging.

Objective-C messaging takes the shape of :

[<receiver> <selector>];[14]
or
[[<receiver> <selector1>] <selector2>];
or
[[[<receiver><selector1>] <selector2>]<selector3>];
etc.

We will just recall here that a main program has the same structure as a C program except for the definition of message groups at the beginning :

= (Message Groups)

except for the declaration of the classes used, inside the program :

extern id <name of class> *[*<other class' name>*]*;**

except for the possible Objective-C messages present in the code and for the clauses ending the file of the program :

@classes (List of used classes);
@messages (List of message groups);

3.2. How it works

We will try to see in this part, how the language works without entering the details.

---

[13] : The type of the argument is not compulsory if it is of type 'id'

[14] : Note that in this case, the [] are integrally part of the syntax and are thus obligatory, as for the items between them.

This part takes in concern the compilation and the creation of instances. It is followed by an explanation concerning messages, inheritance and methods. Finally we introduce the notions of 'self' and 'super'.

3.2.1. *Compiling*

The compilation of an Objective-C program or a class description file by the Objective-C compiler is done in several stages. First we must say that when compiling, Objective-C uses a script to control the compilation and the linking: **objcc**[15].

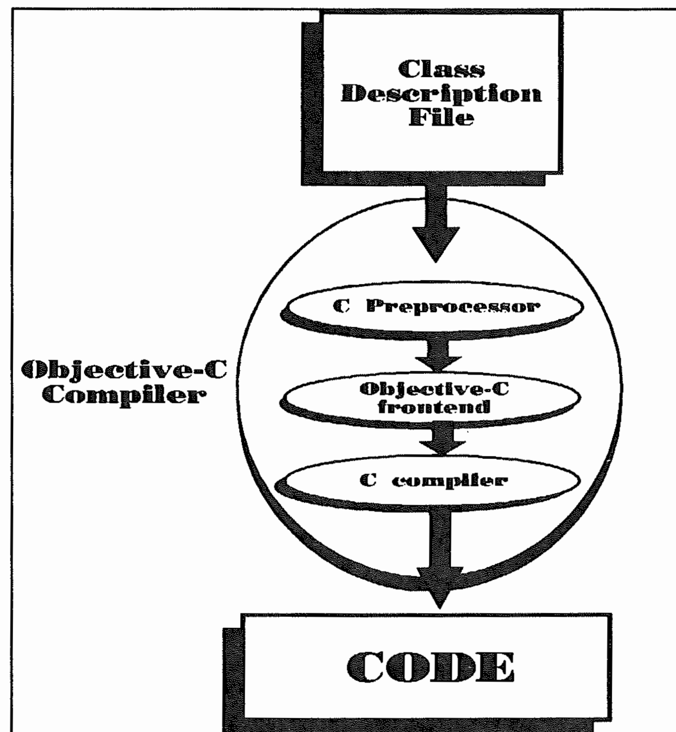


Figure 3.1 The Objective-C compiler composition

A first stage concerns a C Preprocessor which translates statements like '#include' or '#define' into C source; concretely, objcc transforms the '.m' files into '.c' files with the help of the Objective-C compiler.

The second stage concerns an Objective-C frontend, which checks the accuracy of the Object-Oriented code. Then the third stage is the compilation of the generated '.c' files into assembler '.o' files by the C compiler into assembler source code[16] during a third stage. (see figure 3.1, inspired of [COX, 87]).

---

[15] : This script works very much like the C script: **cc**. The reader will refer to [STEPSTONE, 88] to have all the existing compiling options for objcc

[16] : Note that after this, the assembler code is compiled into binary by an assembler

3.2.2. *Instances*

An instance of a class is created from a factory object. This factory object is the result of the compilation of a class; it is its representation in the system. The user must recall here that a factory object is a real object which purpose is to provide a way to create instances of a class. We will only say here concerning compilation of classes that at compile time, two tables are created, one for the instance methods and another for the factory methods. These tables contain a pointer to the superclass -so that the inheritance tree can be followed- but also all the pointers to the C functions representing the methods.

At an instance creation request, the factory object knows the amount of bytes to allocate in memory. This memory block will contain a part from the data structure and will constitute the private part of the instance; it will also contain a pointer to the 'owning' factory object containing the methods -both instance and factory methods. This method part contained by the factory object will be called the shared part of the instance, because shared by all the instances of one same class (see figure 3.2). Note that a factory object, as it is an object like every other one, has also a private and a shared part composed essentially of respectively the instance and factory methods of the class.

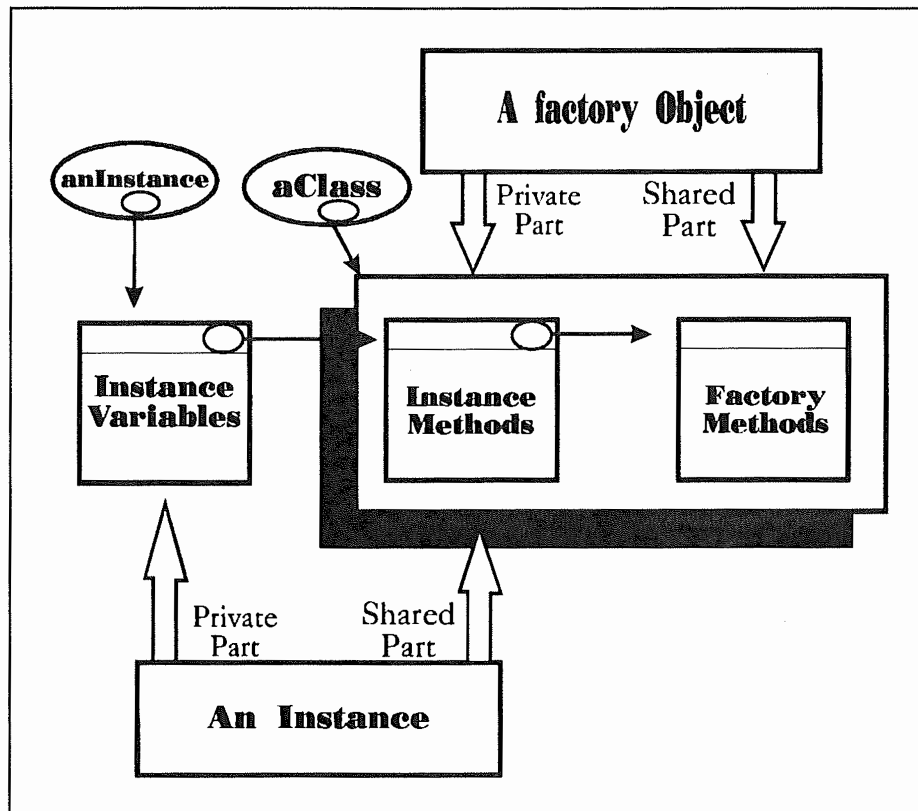


Figure 3.2 Link between instances and factory objects

But we could ask ourselves how can objects be created in the system when nothing already exists? First we saw that when the whole system is running, all

the classes exist under the shape of factory objects. Their purpose is to provide a way to create instances of their class, they are automatically present and available at run-time, because built by the compiler and the linker. So all the material is ready to build instances, one just has to send to the factory objects the right messages. Secondly, an instance of a class is built by sending the class a message referring to a factory method; the result will be the allocation of memory, the initialization of this memory and the 'return' of the address of the newly created instance. This address is stored by the 'caller' in an id variable; only at this stage does a new instance of the class exist. This method allocates memory for the new object's private part ONLY, but also initializes the link between this part and the already existing shared part (see figure 3.2).

This initialization message contains a selector identifying a factory method which is implemented by the class one wants to instantiate or which is inherited by the class one wants to instantiate. In any case the method identified by this selector refers to the 'new' method owned by the root factory object -'Object'- and inherited by any factory object.

3.2.3. *Messages*

Once a user has an object's identifier, he can send this object messages. The principle is similar to calling a procedure, except that here dynamic binding is involved. When a message is specified in a program, it is in fact translated by the Objective-C preprocessor into a call to a function :

_msg() or **_msgSuper()**[17]

in which the first parameter is the receiver of the message and the rest is the message selector. In general, the class receiving the message or the class linked to the instance receiving the message contributes to furnish a table that the routine searches to determine how the object implements the selector. The table in question is composed of the selector of every message that the object knows how to perform and a pointer to the procedure -or function- that implements this message.

3.2.4. *Inheritance*

Inheritance has multiple aspects, in the sense that one can inherit instance variables, instance methods or factory methods.

As a subclass inherits everything of its superclass the variables present in the private part of an object are the duplication of the variables in the private part of its superclass, plus the newly defined ones. At least every class will inherit the only instance variable contained in the root class[18] -variable 'isa'-, which links every object to its shared part[19].

---

[17] : The difference between these two functions is explained later

[18] : The class 'Object', as it is the root of the inheritance tree

[19] : In other words, which contains the address of its shared part

Concerning the inheritance of instance methods, the shared part is not represented by one sole block containing the local and inherited information, instead the inherited information is attached to the eventually new one by linking the shared parts together. Each shared part has a table composed of names and implementation for each method. The messager function[20] only, has access to these links and disposes of two or more arguments, the receiver's identifier and the message selector. This function takes the receiver's identifier to find the private part and follow the 'isa' variable to the table of methods and implementations. If the selector is recognized in this table, the implementation identifies the C function to be executed. If this fails, the search is continued across the superclass shared part table by following the specified link to the next inherited shared part, and so on until it finds something or reports an error because ending the inheritance chain without success.

The case is the same for the inheritance of factory methods. All this seems complicated until one realizes that each object has a private and a shared part and that this last one is a chain of inherited subparts.

3.2.5. *Methods*

Looking at methods, we can see that the compiler builds a factory object by generating its two parts -shared and private- from information provided by the programmer, essentially through the method definitions. Each of these definitions contain several informations :

- whether it is an instance or a factory method
- the return type of the method;
- the selector -or method name-, composed possibly of argument names with their types;
- the method's body.

Note that inside any method, one is able to access a variable 'self' that identifies the object that performs the method. This is detailed later.

We talked earlier about a factory object knowing the amount of memory to be reserved for an instance. In fact when a class is compiled, a variable is created in the private part of the factory with as value, the amount of memory required to hold the class' instance variables. Then, when a factory method is called in a particular class, it stores in the first word of the new block of memory reserved for the new instance the address of the concerned factory object and then returns the address of this block to the caller as the id of the new object. So a new instance has been created -through the C 'alloc' function-, initialized to default values and is thus fully operational as a new, fully functional, instance of a class.

[20] : _msg()

3.2.6. *Self and Super*

Before going further and illustrating what we have said until now, we must introduce two new notions: **self** and **super**.

Methods are called by sending a message to a receiver. The messaging routine chooses the proper implementation for the message and invokes the C function with a formal parameter, 'self', that contains the address of the object receiving the message and enables the system to remember to which object the message was being sent.

Instance variables within an object are described as a C structure declaration[21]. The methods can access these variables by Objective-C looking in the method body for the instance variable names and replacing them by a structure member reference. So the variable 'self' is implicitly used each time an instance variable is accessed in the code. In short, it is a predefined object identifier designating 'a priori' the receiver of the message and enabling access to the private data of this object in the body of a method. As says [FAZARINC, 89] (p.32): "*The variable self has been reserved to always contain the address of the object receiving the message.*"

Sometimes it is necessary to access directly a method in the superclass, for example when a method is inherited but overridden in the current class. Unlike self, 'super' is not a variable, it is really, as says [COX, 87] (p. 83), "*a syntactic device*". Its effect is to 'short-circuit' the messaging process, by calling the function '_msgSuper()' in place of '_msg()'. What changes is the first argument of the function; it is no longer the receiver of the message, but the address of the shared structure in which the selector search should begin. So the pseudo-variable super is defined to cope with the situation where the message search begins in the object's superclass 'method table' and not in its own.

3.3. Illustration through an example of Objective-C program

We tried in the two preceding points to give a general review of Objective-C in terms of concepts and principles found in nearly every Object-Oriented languages. To illustrate these points, we are going to develop a very simple example of class definition based on example 2.6 and with a few added 'items'.

Let us create a class Polygon:

```
=Polygon: Object (MyGroup, Primitive) /* line 1 */
```

---

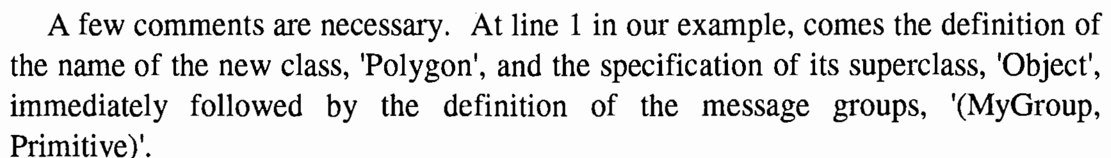
[21] : struct _PRIVATE

```
/****************** Instance variables definition Part ***************/
 {
 int numberOfSides;          /* line 2 */
 }

/***************** Factory methods definition Part ****************/
+new: (int)sides             /* line 3 */
 {
 self = [self new];          /* line 4 */
 numberOfSides = sides;      /* line 5 */
 return self;                /* line 6 */
 }

/***************** Instance methods definition Part ****************/
-(int)howManySides           /* line 7 */
 {
 return numberOfSides;       /* line 8 */
 }
=:                           /* line 9 */
/****************** End of class Polygon definition ***************/
```

And another class Square, subclass of Polygon:

```
= Square: Polygon (MyGroup, Primitive) /* line 10 */

/****************** Instance variables definition Part ***************/
 {
 int lengthOfSide;                /* line 11 a*/
 char stringSquare;                /* line 11 b */

 }

/***************** Factory methods definition Part ****************/
+new: (int)aLength                /* line 12 */
 {
 self = [self new: 4];            /* line 13 */
 lengthOfSide = aLength;          /* line 14 a */
 strcpy(stringSquare, "This is a Square"); /* line 14 b */
 return self;
 }

/***************** Instance methods definition Part ****************/
- changeLength: (int)aLength      /* line 15 */
 {
 lengthOfSide = aLength;          /* line 16 */
```

```
    return self;
    }

  - (int)howManySides                    /* line 17 */
    {
     int sides;                          /* line 18 */

     sides = [super howManySides];   /* line 19 */
     printf (" %s and it has %d sides !!!", stringSquare, sides); /* line 20 */
     return sides;
    }
  =:
/*****************End of definition of class Square*****************/
```



A few comments are necessary. At line 1 in our example, comes the definition of the name of the new class, 'Polygon', and the specification of its superclass, 'Object', immediately followed by the definition of the message groups, '(MyGroup, Primitive)'.

At line 2 one finds the instance variable definition part, with 'numberOfSides' defined as an integer. Note that as standard C can be used, the programmer can define global C variables, define some constants, define some local variables inside methods and so on, just as in a normal C program.

At line 3 starts the definition part of the factory methods. Note that no order is imposed to define factory or instance methods, but by convention one will first define all the factory methods and then the instance methods. So line 3 defines a factory method -'new:'- for this class, with an integer argument, 'sides', which represents the desired number of sides of the new instance of Polygon. Line 4 is where the instance is really created. In '[self new]', 'self' designates the Polygon factory object, in other words the receiver of the message and 'new' designates the method invoked, inherited from 'Object'; the result of this message will be the return of the address of the new instance created. This address will be stored in 'self', but could indeed be stored in another variable. This is a principle adopted by many Objective-C programmers so that the 'self' variable references directly the instance of a class.

Line 5 initializes Polygon's only instance variable with the value of 'sides'. Line 6 specifies the returned value of this method, which is 'self' as this method has nothing important to return. This will enable imbrication of messages, as said in point 3.1.2.

With line 7 starts the instance methods definition part and the definition of method 'howManySides'. This method has no argument and is only designed to return the value of an instance variable, 'numberOfSides' which is an integer (see line 8); this is why the returned variable type is specified before the method name. Line 9 means that the class definition is finished.

Line 10 starts the definition of another class with the definition of its name -'Square'- and its superclass which will be here the class Polygon. This is followed by

the definition of the message groups. Line 11 a and line 11 b define the two instance variables of class Square: 'lengthOfSide' and 'stringSquare'.

At line 12 starts the definition part of the factory methods. It defines a factory method for this class, with an integer argument, 'aLength', which represents the desired side length of the new instance of Square.

Line 13 is where the instance is created. In '[self new: 4]', 'self' designates the Square factory object, 'new:' designates the factory method inherited from Polygon and 4 is the desired number of sides of the polygon; the result of this message will be the return of the address of the newly created instance. This address will be stored in self.

Line 14 a initializes Square's instance variable -'lengthOfSide'- with the value of 'aLength' and line 14 b copies into Square's second instance variable -'stringSquare'-, the value "This is a Square".

Line 15 starts the instance methods definition part by defining the method 'changeLength' which has one integer argument -'aLength'. This argument represents the desired side length to be stored into lengthOfSide. Line 16 stores the argument's value into the instance variable.

Line 17 starts the redefinition of Polygon's method 'howManySides'. Line 18 defines a local C variable named 'sides' which is an integer. Line 19 stores into the local variable 'sides', the value returned by the method 'howManySides' in the superclass. '[super howManySides]' causes the message to be directly sent to the superclass. Line 20 prints a string on the screen composed of Square's instance variable 'stringSquare' and of the local variable 'sides'.

Now that we have created our two classes, let us build a main program to test them :

```
= (MyGroup, Primitive)                    /* line 1 */

/******************Main program to test Polygon and Square***************/

main()                                    /* line 2 */
 {
 extern id Polygon, Square;               /* line 3 */
 id aPolygon, aSquare;                    /* line 4 */
 int numSquSides;                         /* line 5 */

 aPolygon = [Polygon new: 5];             /* line 6 */
 aSquare = [Square new: 10];              /* line 7 */

 printf ("This is a Polygon with %d sides \n\r", [aPolygon howManySides]);
                                          /* line 8 */
 numSquSides = [aSquare howManySides]; /* line 9 */
 }
/******************End of the program******************************/
@classes (Polygon, Square)                /* line 10 */
```

```
@messages (MyGroup,Primitive)          /* line 11 */
/*******************End of the definition of the test program***************/
```

Line 1 declares the message groups to be used in the program, of course one must take in consideration the message groups of the different classes used in the program. Here the message groups are 'MyGroup' and 'Primitive'.

Line 2 starts the main program as any C program. Line 3 is the line where the factory objects are made available; in our case we use the classes Polygon and Square.

Line 4 declares two local variables that will be used to point at instances, respectively of Polygon and Square. These two variables are 'aPolygon' and 'aSquare'. Line 5 is also the declaration of a local variable -an integer-, 'numSquSides', that will contain the number of sides of the instance of class Square pointed at by aSquare.

Line 6 contains a message which creates an instance of class Polygon; it will also have as effect, if we look at the method 'new:' in Polygon, the 'storage' of the value 5, passed as parameter, into the instance variable of Polygon: 'numberOfSides' and the return of the address of the newly created instance. This address will be stored in the variable 'aPolygon'. Note that the receiver is explicitly specified -'Polygon'-, because we are not currently in a class -we are in a program- and thus the variable 'self' can not be used, the system not knowing at which factory object this variable would be pointing.

Line 7 creates an instance of class Square. By calling the method 'new:' of class Square, this will create a Polygon of 4 sides, store the value 10, passed as parameter, into class Square's instance variable 'lengthOfSide' and copy the string 'This is a Square' into class Square's other instance variable 'stringSquare'.

At line 8, one can see that in the message '[aPolygon howManySides]', aPolygon is the receiver of the message, so we are sending the message to the instance of Polygon created above and one can see also that the selector is 'howManySides', causing this method to be executed. The overall result of line 8 is the printing on the screen of the string 'This is a Polygon with %d sides', where '%d' will be replaced by the value returned by the message '[aPolygon howManySides]'.

At line 9, one can see that in the message '[aSquare howManySides]', aSquare is the receiver of the message, so we are sending the message to the instance of Square created above and one can see also that the selector is 'howManySides', causing this method to be executed. This execution will cause the message '[super howManySides]' to be sent to aSquare's superclass, the result of this message-passing to be stored into the local variable 'sides' and the string "%s and it has %d sides !!!" to be printed, where '%s' will be replaced by aSquare's instance variable 'stringSquare' and %d by the value of 'sides'. The overall result of line 9 will be the 'storage' of the result of the message-passing -'[aSquare howManySides]'-, which is the number of sides of a square, into the variable 'numSquSides'.

Line 10 and 11 are required by the Objective-C compiler to be able to combine all the classes and messages used in a program so that it can build accurate tables to reference them. Line 10 defines the classes used in the program: Polygon and

Square. Line 11 defines the message groups used in the program: MyGroup and Primitive. (**Example 3.12**)

We can see through this example how a class definition is really structured and how all the principles of Object-Oriented languages apply to Objective-C :

- ENCAPSULATION : to access the instance variable 'numberOfSides' -in Polygon-, one must use method 'howManySides'.
- INHERITANCE : class 'Square' inherits of the instance variable 'numberOfSides' of Polygon and the method 'howManySides' which is however overwritten in this class. Class 'Polygon' inherits of factory method 'new' from its superclass 'Object'.
- MESSAGING CAPABILITY : messages can be found in the two classes, as '[self new]' in method 'new:' of class Polygon or as '[super howManySides]' in method 'homManySides' of class Square, etc.

This example is very simple but provides a good view of a class structure, of the utilization that can be made of classes and finally a good illustration of what has been said about Objective-C in the preceding points.

3.4. Summary

In this chapter, we reviewed quickly an Object-Oriented language: Objective-C. With the presentation of the main topics we saw that it was an extension of the C language, Smalltalk-80 based and bringing to C two things :

- a new data type, id
- a new operation, messaging.

We saw also that in Objective-C, the classes are merely the description of informations related to a group of similar objects and that they are represented at run-time by what is called a factory object. This object is unique in the system, is also composed of a private and a shared part and its goal is to produce instances.

Concerning the instances, they are composed of a private part, the data specific to each instance, and a shared part composed mainly of the methods shared by each instance -methods inherited or newly built.

We described the main points of the syntax and tried to explain the functioning of this language :

- COMPILATION is carried on through three different steps.
- INSTANCES are built 'using' factory objects. These objects are built at compile time and are always present in the system. They keep the necessary informations to be able to create the instances.
- MESSAGES are implemented using two C functions, _msg() and _msgSuper().

- SIMPLE INHERITANCE concerns instance variables by duplication of the superclass variables in the object's private part; it concerns also instance methods and factory methods, by linking the object's shared part to the shared part of its superclass.
- the 'new' METHOD reserves a certain amount of memory -known from a variable kept in the factory object's private part- and places the address of the factory object at the beginning of the block, before returning the address of this block.

After that we introduced two new notions :

- self : an object identifier designating 'a priori' the receiver of the message and enabling access to the private data of this object in the body of a method.
- super : a pseudo-variable defined to cope with a situation where the message search begins in the object's superclass 'method table' and not in its own.

To finish we gave a short example to illustrate what had been said.

Chapter 4: Looking at RMG

Our goal in this chapter is to look at an Object-Oriented environment: RMG (Real time Measurement Graphics). Designed by Hewlett-Packard laboratories, RMG is used in a European project called COLOS (COnceptual Learning Of Science), whose goal is to build simulation tools for Computer Assisted Learning (CAL).

In the first place we see what the COLOS project and RMG imply (4.1), stressing the fact that RMG can be seen as a Graphical User Interface. Then we try to lead the user in his first steps in the RMG environment (4.2). The last point describes more in depth two RMG applications (4.3)[1].

4.1. The COLOS project and RMG

We are going to describe briefly the ins and outs of the COLOS project and RMG.

The COLOS (COnceptual Learning Of Science) project is a European project initiated and sponsored by the Hewlett-Packard company. It gathers approximately ten European universities whose goal is to create graphical, interactive simulation applications in the RMG (Real time Measurement Graphics) Object-Oriented environment. These applications are designed to be used mainly in universities as pedagogical tools for teachers and students. The applications concern fields as different as electricity, integrated circuits design, computer communications, etc.

The software base of the project is the RMG environment. It was written in Objective-C by Charles Young[2] and currently runs on a Hewlett-Packard workstation -9000/360 series- with HPUX (UNIX) as operating system. This environment is designed to be highly graphical and highly interactive, offering to the user a great suppleness of utilization.

RMG can be seen as a Graphical User Interface, as we see in a first point. But it can also be seen as an application development platform, as we see in a second point.

4.1.1. *RMG as a Graphical User Interface*

RMG can be seen by some users merely as a **Graphical User Interface** -GUI- designed to run applications. GUI usually implies for the 'common user' :

- using a **mouse**;

---

[1] : The reader will notice from now on that very few authors are quoted. This is due to the near complete lack of books and works on the RMG environment. The following work done on the RMG environment is the result of our working experience in this environment and is thus 'non-exhaustive'

[2] : Charles Young, Hewlett-Packard research laboratories, Palo Alto, California, USA.

- coping with various **windows**[3];
- clicking[4] with the mouse on **icons** representing programs or files;
- coping with roll-down or pop-up **menus**;

all this on the display screen.

RMG fully enables all this[5]. It is a mouse-driven environment; so once inside, the user is able to control it using a two buttons mouse, represented on the screen by an icon[6] -usually a hand or an arrow.
The actions that can be performed with the mouse are among others :

- clicking on objects present on the screen -icons, boxes, etc-;
- dragging[7] the mouse -to move an object on the screen for example.

Windows are also present in RMG. Each application that runs in RMG runs inside an independent window, enabling the user to run various applications at the same time, in different windows.
Icons are omnipresent in RMG. They usually represent two different things :

- an application's window that the user does not want fully present on the screen at the current moment;
- an action (making some text scroll up or down in a text editor, enlarging an application's window, etc) that can be performed if the user clicks on this icon.

These icons take the shape of little squares with a drawing or a string inside them.
Menus are something that the user can also find in RMG. The principal reason to this is that, as we explain in a following point, menus are one of the communication means that enables the user to interact with the environment and its applications. For example, when the user enters the environment and wants to start a particular application, he has to select an option in a menu.
Note that it could be a very interesting thing to compare RMG to other GUIs like 'X Window' for example[8] and could be the subject of a future work.

In regard of what was just said now, we can state that the four aspects reviewed above and concerning a GUI are to be found in each RMG application.

---

[3] : By 'window', we mean a rectangle displayed on a part of the screen representing and containing a running application

[4] : By clicking we mean depressing one of the mouse buttons, while the icon representing the mouse is positioned on an object present on the screen

[5] : The reader will see in a following point how to use RMG as a Graphical User Interface only

[6] : We call this particular icon the '**RMG cursor**'

[7] : By dragging we mean depressing one of the mouse buttons and holding it down while moving the mouse

[8] : This is not done in this work as it is outside its scope and because of our lack of knowledge concerning the X Window environment

We talk later about developing an application in RMG and we see that a certain number of classes shipped with the environment enable the programmer to create these graphical screen objects -icons, menus, etc- and enable the programmer to include in his applications every mouse functionality, allowing them to be highly 'mouse-interactive'.

Another important parameter in a GUI is the graphical speed. It brings many advantages :

- it enables the user not to wait too long for the graphical information;
- it enables the creation of applications using more complicated graphics;
- it enables the creation of applications using graphical animation;
- it enables the creation of applications which 'redrawing time' is faster than the user perception time.

The particularity of RMG on this point is that it is able to redraw the whole screen and to manage the graphical memory at a very high speed[9]; this enables the use of complicated graphics in the applications and in the general graphical presentation of the environment. This is of course a big advantage for the creation of COLOS simulation applications using animation.

To conclude this point, we just want to say that RMG is reinforced in its role of GUI by the gist of the COLOS project itself, which is the creation of simulation applications to be used in the RMG environment possibly by non-programmers. This is supported by the presence of many graphical applications designed by the Hewlett-Packard laboratories and the members of the COLOS project. Indeed, one can find various tools like a text editor or a graphical editor. But one can mainly find various simulation programs as 'Particle Physics' or 'Elstat' which simulate particles interaction or electrical fields interaction, 'ICDesign' which simulates the elaboration of an integrated circuit design, etc.

4.1.2. *RMG as an application development platform*[10]

The second aspect of RMG is the **application development platform** aspect.

As RMG is an Object-Oriented environment and was created with the Objective-C language, the development of applications follows an Object-Oriented 'philosophy'[11]. Nevertheless, we can say that the creation of an application in the RMG environment can be summarized into three steps :

- **creation** of the classes that composes the application;

---

[9] : Due mainly to the management of the graphical memory by assembler routines

[10] : Note that everything about the creation of an application is fully detailed in a following chapter

[11] : Except for some little changes, we will see later that it uses Objective-C's syntax

- **compilation** of these classes;
- **incorporation** of the application in the environment.

For the moment we do not have to know how an application and its classes are created, it is explained later. We just have to know that RMG enables the user to create applications outside the environment or inside the environment using some of its applications. If the applications are developed inside the environment, various tools are offered to the programmer. Among these tools, the basic ones are editors -both text and graphical editors- and a certain number of browsers. These browsers are very useful tools that enable the user to consult a class' code, to obtain the address of an instance or to obtain the address and value of all its instance variables, etc. One also finds :

- a menu tool which enables the user to easily create an application's menu;
- a prototyping tool which enables the user to create applications by graphically defining scenarios under the shape of flow charts. The various objects in the flow charts are controlled by attaching algorithms or equations to each scenario;
- a program interpreter that enables the user to utilize RMG classes without compiling Objective-C code.

RMG also provides the user with a certain number of classes. First the user disposes of all the classes present in the Objective-C libraries. Among these Objective-C classes, one can find classes designed to cope with arrays, graphical objects, etc. Secondly the user disposes of all the classes present in RMG. These classes are mainly designed to create graphical objects such as squares, boxes, windows but also strings, menus, etc.

4.2. First steps in RMG

In this part we are going to try and lead the user in his first steps in the RMG environment.

One thing one needs to recall before starting RMG is that nearly everything is mouse-driven. This means that 90% of the things done by the user are mouse directed, the 10% left being mainly some text typing, imperatively done from the keyboard.

'**env.out**' is the name of the executable file to be typed by the user at the Unix prompt in order to start the RMG environment. This file is located in the '**RMG/ENVIR**' directory. After typing 'env.out', hitting the 'Return' key and waiting a few seconds the screen goes blank and the user is able to see the RMG cursor[12] by moving the mouse up. Says [FAZARINC, 89] : "*The appearance of the hand-cursor indicates that you have a working RMG environment literally at your fingertips.*"

[12] : The cursor takes the shape of a hand

From now on the user is inside the RMG environment, everything depends on his actions, on his requests. The user is not prompted to reply any questions, no window is created without him asking for it. In other words the user has got the initiative.

At this stage the user is not very far; no application is running; nothing appears on the screen except the RMG cursor. By depressing the **right-hand side button** of the mouse the user sees a **pop-up menu** appear on the screen -at the current location of the RMG cursor-, this is the environment's main menu. As we said earlier, menus are one of the main means of action in the RMG environment and each application has its own particular one[13]. While holding down the right button in order to make the menu appear and pushing the mouse up and down, one can see the RMG cursor and the highlights of the menu items[14] following the motions of the mouse. Now, when one wants to select something from a menu[15], one depresses the right-hand side button to make the menu appear, moves the mouse until the desired menu item is highlighted and releases the button; the action associated with the menu item is performed. Note that if the number of menu items exceeds the length of the menu, non-displayed menu items can be reached by moving the highlights of the menu items towards the top or the bottom of the menu, the hidden menu items appear with the menu scrolling up or down. Note also that to access the menu of a particular application, the user has to position the RMG cursor into the desired application and then proceed as depicted above. This introduces one of the conventions of RMG, the right-hand side button of the mouse, when depressed, makes pop-up menus appear. It is valid for each RMG application. Note that concerning the RMG cursor, its shape can vary depending on its position on the screen; in some cases it becomes a simple arrow, a cross, etc.

The environment's main menu (see figure 4.1) is the menu from which each application can be started, from which various parameters of the environment can be changed, etc.

For the moment, the two most interesting menu items in this menu are :

- the '!! **QUIT** !!' item;
- the '*NEW* >' item[16].

The first one is the item one chooses when one wants to quit the environment[17]. The second one hides a submenu, listing all the applications executable in the environment. If one wants to run a particular application, one just has to go and see

---

13 : If some applications don't have any menu this is only a design decision of the programmer

14 : We call here a menu item, a line appearing in a menu and representing a possible choice for the user. A highlight is a menu item appearing in reverse video or in an other color

15 : Select an item in a menu to execute an action, change some parameters, etc (further details on menus will be given in a following part)

16 : Note that when the '>' sign appears in a menu item, it means that a sub menu is hidden by this menu item and is accessible by dragging the mouse rightward (the sub menu appears beside the current one)

17 : Normally this menu item is always present in a menu. If this menu item is selected by the user inside an application's menu, the resulting effect is the 'closing' of the application's window

in this submenu and select -like explained above- the desired menu item representing the name of the application; this application soon appears on the screen. The menu is a very important concept in RMG because of the user using it very often to 'communicate' with the system.

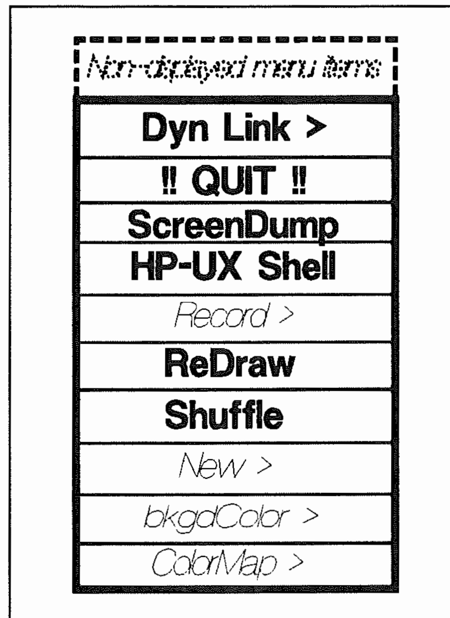


Figure 4.1 Pop-up menu of the environment

We talked above of an application 'that appears on the screen', without explaining anything on how it appears. We just want to say that if the user selects an application to be run, in the main menu of the environment, a window appears on the screen with the application running inside it. So each application can be run as many times as wanted. Different applications can be run at the same time, each of them taking place on the screen in different windows completely independent from one another. The user is able to interact with one or the other just by positioning the RMG cursor inside the desired window and depressing the left-hand side button of the mouse.

Other features are often available in applications menu, among which the major ones are :

- a '**Size**' item, which enables the user to stretch or reduce the application's window with the mouse;
- a '**Iconize**' item, which enables the user to reduce the application's window to an icon[18];
- a '*color>*' item which enables the user to change the background color and the text color of the window.

[18] : These two items are very often listed in a sub-menu accessible from the item '*OTHER >*' in the application's menu

The other items available in application menus depend highly on the kind of application. Indeed, we see later that a programmer can build his own application's menu and thus include in this menu any menu item useful for the application.

We can say now that as soon as the user runs an application, he is able to work with it. 'Work with it' means that the user is able to execute a certain number of things inside this application by selecting menu items in the application's menu or by coping directly with different objects on the screen, among which one can find :

- buttons to click on,
- scroll bars to drag,
- graphical objects to move,
- etc.

This other way of communicating with the system is done by depressing the **left-hand side button** of the mouse while the RMG cursor is positioned on a screen object; we call this '**clicking**' on a screen object. Here comes a second RMG convention: the left-hand side button of the mouse is used to click on screen objects. For example if the user places the RMG cursor into an application's window, depresses the left-hand side button of the mouse and drags the mouse, the user is able to make the outline of the window follow the motions of the mouse; if the user releases the button, the window is redrawn at the new place where the outline of the window is situated[19]. This convention is also valid when an application's window contains :

- buttons or icons designed to perform actions when clicked on with the mouse (see figure 4.2.b and 4.2.c);
- scroll bars capable of being dragged up and down (making some text scroll for example) (see figure 4.2.c);
- graphical objects that can be moved around in the application's window;
- etc.

Now we just want to say a few words about actions. Indeed, we talked above of actions attached to menu items, to buttons and icons, but what is an action? An action is something performed by the system that modifies the current state of the application or of the environment -we see later that this 'something' can be a method or a C function. For example, when one selects an application to be run, into the main menu of the environment, a window is created with the application running inside it, thus changing the state of the whole environment. Actions can be attached to menu items, this means that when one selects this menu item the action linked to it is performed. They can also be attached to icons, to buttons,

[19] : Note that this is only an example and that the implementation or 'not-implementation' of this type of action in an application is the responsibility of the programmer

etc; this means that when the user clicks[20] on one of these screen objects, the action linked to it is performed.

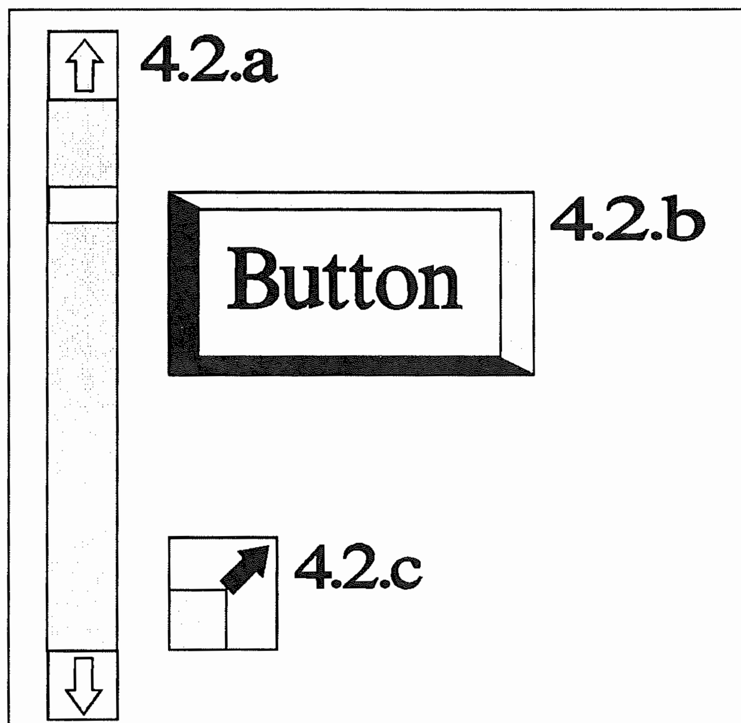


Figure 4.2 Various screen objects

We just saw here a little portion of what the RMG environment and the user are able to do together. We just saw the basic principles that one has to master to use RMG as a Graphical User Interface and get accustomed to it by 'playing' around with the different applications available. The next point analyses two RMG applications.

4.3. Getting accustomed to RMG with two applications

In this part, we analyze two traditional applications one can find and use in the environment. This is done so that the user can get accustomed with the overall handling of RMG applications. The first one is a graphical editor -'**IconEdit**'- and the second one an application called '**MoleView**'.

Note that more applications are available in the environment, we chose the first one because it is usually available in other environments -in the shape of bitmap editors or 'painting softwares'- so we want here to present something that is usually found in GUIs. We chose the second one because it represents a tiny fragment of what is tried to be done in the COLOS project with RMG.

[20] : We recall that from now on 'clicking' on a screen object always mean that the user positions the RMG cursor on the screen object and depresses the Left-hand side button of the mouse

4.3.1. *IconEdit*

IconEdit is a graphical editor which was designed initially to enable users of the environment to create easily icons and images.

This application is started by selecting the menu item '**IconEdit**' in the '*NEW* >'[21] menu item of the main menu. When started, the application appears on the screen in a window, if one looks at this window with attention one can see several things.

The first thing is a large black square in the middle of the window, which is the area where the image can be created[22]. Secondly, in the lower right-hand side corner of the window, one can see the current width and height -in pixels- of the drawing area. Note that the drawing area has a starting width and height of 16 by 16 pixels. If the image is larger than the window, it can be scrolled using the icon situated in the lower right-hand side corner of the window. One can see also the current position of the RMG cursor in the upper left-hand side corner, this position is relative to the origin of the drawing area[23]. The little icon in the upper right-hand side corner is the icon one uses in order to stretch the application's window[24]. Last is the sub-window situated at the bottom of the icon editor, which displays the status of the editor and the current color used in the editor.

Let us now press the right-hand side button of the mouse, with the RMG cursor in the application's window; the main menu of the application appears (see figure 4.3). We are going to detail certain menu items.

We are familiar with the first menu item, '!! **QUIT** !!', that quits the application by closing the application's window.

The user is able to specify individually the width and height of the drawing area, using the option: '*Specify* >'.

One of the menu item is: '*Gain* >'. This menu item enables the user to change the magnification of the drawing area by selecting a value in the submenu it hides. The default magnification at creation is 8; by changing it to 1, one can see the true size of the drawing area.

When working with IconEdit, one could like to have a grid on the drawing area, enabling the user to see the magnified pixel size. This is possible, using the menu item: '*Border* >'. When selecting the value 1 in the submenu, a grid soon appears in the drawing area. This grid disappears when selecting the value 0 in this same submenu.

---

21 : The reader must recall that when the '>' sign appears in a menu item, it means that a sub menu is accessible by dragging the mouse rightward (the sub menu appears beside the current one)

22 : We call this area the drawing area.

23 : This position is measured in pixels and is relative to the X and Y axis

24 : This is done by clicking on the icon and dragging the mouse while holding down the button

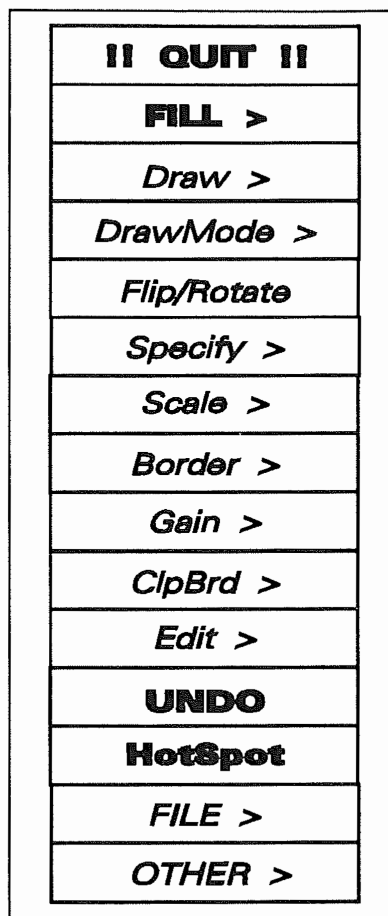


Figure 4.3 Main menu of the IconEdit application

Another important menu item is '*Edit* >'. If one selects the menu item '**ON**' in its submenu, a set of colors appears at the right of the drawing area. These colors are the colors one can choose to work in IconEdit. One selects the desired color by just clicking on it in the set of displayed colors. When a new color is selected, the status subwindow displays the newly selected color and points to it with the string: '**<- current color**'.

Now if one positions the RMG cursor on the drawing area and clicks once, a pixel appears in the selected color. If one clicks and drags the mouse while holding the mouse button depressed, a trace of pixels appears in the selected color, following the motions of the mouse. The user notices that his actions are reproduced in true size within the little window at the left edge of the application's window. This little window can be enlarged by positioning the RMG cursor on its right edge, clicking on it and dragging the mouse while holding the button depressed.

The '*Draw* >' menu item enables the user, through its submenu, to draw a certain number of shapes, such has circles, boxes, etc. When the user selects one of these shapes, the choice made is echoed in the status subwindow and the RMG cursor changes from an arrow to a pen tip. The user has just to position the RMG cursor into the drawing area and click while dragging the mouse to see the desired

shape being created. The user is able to draw the selected shape anywhere in the drawing area, until he clicks once on the right-hand side button of the mouse; by doing it the user sees the RMG cursor revert to normal. Note that the user is able to create any desired shape by drawing lines connected to each other, thereby creating a polygon.

The user is able to store a drawing in a temporary place, called a clipboard. This enables him to start a new drawing while keeping the initial drawing somewhere where he can access it immediately. This is possible going through the sequence[25] of menu items: '*ClpBrd* >', '**Cut To** >', '**whole image**'. When this is done, the user sees a new window appear on the screen containing the saved drawing. The user is also able to cut out a defined portion of a drawing by selecting '**Cut To** >' in the submenu of the '*ClpBrd* >' menu item. At this stage and returning to the drawing area, the RMG cursor is replaced by two cross-hairs whose intersection defines the upper left corner of the area which is cut out to the clipboard. By clicking and dragging the mouse while holding down the button, a rectangular shape is drawn, defining the area to be cut. When the button is released, the defined area appears in the icon clipboard.

If one wants to suppress a drawing, one has just to select the black color and go through the sequence: '*Edit* >', '**ON** >', '*replace* >', '**all with CurColor**'. Note that this operation is valid with every color in the set of colors, except that the drawing is not 'erased' if the selected color is not the same color as the background color of the drawing area.

The sequence '*Edit* >', '**ON** >', '*replace* >', '**picked with CurColor**' enables the user to replace with the selected color, each portion of the drawing colored with a particular color. One has just to go through this sequence and then click on the desired color in the drawing, the change is automatic.

The sequence '*Edit* >', '**ON** >', '*replace* >', '**region with CurColor**' enables the user to change the color of a defined area into the selected color. To do this, one goes through this sequence and goes back to the drawing area, where the RMG cursor is changed to two cross-hairs whose intersection defines the upper left corner of the area which has to be transformed. By clicking and dragging the mouse while holding down the button, a rectangular shape is drawn, defining the area to be changed.

One can also enlarge the surface of the drawing area by going through the sequence: '*Edit* >', '**ON** >', '*enlarge* >' and then select a number representing the coefficient by which the height and width of the drawing area has to be multiplied. The new height and width is displayed in the lower right corner of the application's window.

As one can see, there is a '**Fill** >' menu item present in the main menu. This menu item is chosen to fill a shape or an area with a particular color. It has three possibilities. If the menu item '**Fill** >' is selected, the area pointed at by the RMG

---

[25] : By "going through the sequence", we mean going down in the successive sub-menus, starting from the main menu

cursor is filled, stopping at the current color boundary. If the menu item *'border color >'* is selected in the '**Fill >**' submenu, everything bordered with the specified color is filled, without paying attention to shapes or structures. Finally, if the menu item '**Flood Fill**' is selected in the '**Fill >**' submenu, the shape pointed by the RMG cursor is filled with the selected color. Note that it is always possible to undo the last change made to a drawing by selecting the '**UNDO**' menu item in the main menu.

Now let us go back to the drawing we saved via the clipboard. By going through the sequence: *'ClpBrd >'*, '**Pattern Fill >**', *'border color >'* and selecting a color in the color bar hidden at the right of *'border color >'*, the user is able to fill shapes delimited by the selected border color with the drawing contained in the clipboard. Note that many drawings can be saved one after the other in the icon clipboard, the user can skip from one drawing to another with the six tape-recorder type buttons at the bottom of the icon clipboard window. Note also that the drawing restored with sequence *'ClpBrd >'*, '**Pattern Fill >**', *'border color >'* is the one currently displayed in the icon clipboard.

From the clipboard, one is also able to restore a portion of a drawing cut out earlier. To do this, one must go through the sequence: *'ClpBrd >'*, '**Paste From >**' and one of the four menu items contained in the submenu. For example if one selects '**Replace**' and returns to the drawing area, he sees the RMG cursor replaced by a rectangle and the status window warning that the IconEditor is in '**Paste From Clipboard**' mode. The above mentioned rectangle figures the outline of the former cut out. When clicking, the cut out appears at the location of the rectangle. One is able to do this as many times as wanted and one can stop it by depressing the right-hand side button of the mouse. Recall that the desired cut out must be displayed in the icon clipboard.

We just reviewed the basic possibilities of this very rich application. The user is able to find more of the particularities of IconEdit by 'playing' around with it.

4.3.2. *MoleView*

MoleView is an application designed to view three-dimensional representations of molecules[26]. It can be seen as "*the rotation of three-dimensional objects consisting of spheres of varied sizes and colors around the azimuthal and polar axes at controllable speeds.*" (says [FAZARINC, 89] p.16).

The overall functionalities of this application are that :

- the displayed structures can be modified in terms of positions of the spheres;
- the displayed structures can be zoomed, either in or out;
- the displayed structures can rotate;

[26] : We call structures the 'representations of molecules'

- the structure's center of rotation can be chosen to be any sphere composing it.

This application can be started by selecting the menu item '**MoleView**' in the '*NEW* >' menu item of the environment's main menu. When started, the application appears on the screen in a window; if one looks at this window with attention one can see several things.

The first thing one notices when starting the application is the presence of one structure, constituted of six spheres connected to the seventh central one by 'straight bonds'.

The second thing is the presence of four arrows on the right-hand side of the window. Two of them are placed respectively above and underneath the 'θ' sign and are used to make the structure rotate around the azimuthal axis. The two others are placed respectively above and underneath the 'ϕ' sign and are used to make the structure rotate around the polar axis.

At the top of the application, one finds three labels. When the user clicks on the first one -starting from the left-, '**Zoom In**', the structure currently displayed is enlarged as long as the mouse button is depressed. The second is the status of the application. The third, '**Zoom Out**', enables the user to reduce the size of the currently displayed structure.

At the bottom of the application, one finds two labels -'**Remove**' and '**Restore**'- which are respectively used to make the spheres disappear in order to view the center of the structure and to restore the initial state of the structure -that is with all its spheres. Note that when the user clicks on one of the spheres, its number is displayed in the status window.

One is also able to change the positions of the spheres by positioning the RMG cursor on the desired sphere and by clicking and dragging while holding down the mouse button, the sphere then follows the motions of the mouse with the bonds stretching or shrinking accordingly.

Looking at MoleView's main menu (see figure 4.4), one can see several menu items; '**START**' is one of them. This item makes the displayed object rotate automatically on both axis. The '**STOP**' menu item stops this rotation.

One can find two other menu items, '**Connect**' and '**Disconnect**', that enable the user to change the bonds between the spheres. For example, to remove a bond between two spheres the user has to select the '**Disconnect**' menu item in the main menu and has next to click on the spheres linked by the bond, each in turn; the bond between them will disappear. Note that this suppression is only graphical, this process does not affect the positions of the spheres in the structure. The '**Connect**' menu item works following the same principles, but for the connection of two spheres.

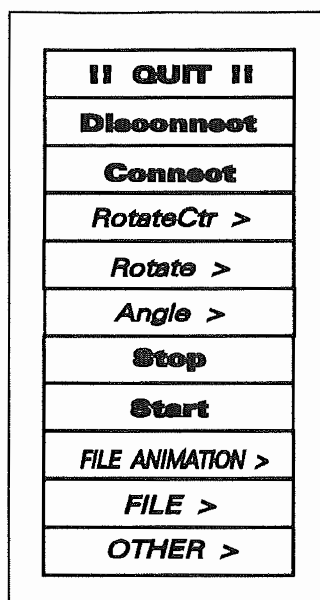


Figure 4.4. Main menu of the MoleView application

The '*RotateCtr* >' menu item enables the user to define the sphere that is the center of rotation of the structure. Two menu items are present in this menu item's submenu: '*Center Mass*' and '*Atom No* >'. The first one, when chosen, specifies that the center of rotation is the center sphere. The second one enables the user to choose, in a third level submenu, the number of the sphere he wants to be the center of rotation.

The '*Rotate* >' menu item enables the user to specify the angular velocity around the two axes. Two menu items are present in this menu item's submenu: '*Azimuthal* >' and '*Polar* >'. Both can be adjusted by the user highlighting the desired menu item and dragging the mouse to the right. A third level submenu appears under the shape of a box containing a number -the angular velocity. This number can be changed by the user pushing the mouse up or down. This is of course done with the user always holding down the right-hand side button. The menu item '*Angle* >' enables the user to adjust the 'θ' and 'ϕ' angles, with the same manipulations.

Note that other structures can be brought into MoleView, using the '*FILE* >' menu item and its submenu.

As for IconEdit, it would be too long to describe each menu item and their possibilities in full. But we reviewed through these two traditional RMG applications, some of the basic manipulations one finds in nearly all RMG applications.

4.4. Summary

Our goal in this chapter was to look at an Object-Oriented environment: RMG (Real time Measurement Graphics).

In the first place, we saw that RMG was an environment used in a European project: the COLOS (COnceptual Learning Of Science) project. This project's goal is to create interactive simulation applications to be used in universities as a pedagogical tool.

The next point showed that one could consider RMG as a GUI (Graphical User Interface), because it implies among other things :

- the use of a mouse;
- the management of various windows;
- the management of icons representing programs;
- the use of menus.

Another point described RMG as an application development platform, because it enables the user to create his applications using or not the various tools and classes shipped with RMG.

Then we gave a little survey on how to make ones first steps in the environment, by explaining how to use RMG's first communication means: the menu. We saw also that the user could control the application through icons, buttons and other screen objects.

We finished by describing more in depth two RMG applications: 'IconEdit' and 'MoleView'.

SECOND PART: A little guide to RMG

Chapter 5: Interesting bibliography

In this chapter, we give the reader a non-exhaustive bibliography that we advise him to consult if he wants to start programming in RMG.

As we saw in the preceding chapters, RMG is an Object-Oriented environment, constructed with Objective-C. Therefore one has to acquire certain notions before starting developing applications in this environment. These notions include :

- knowledge of Object-Oriented mechanisms;
- knowledge of the C language;
- knowledge of the Objective-C language;
- notions of the Unix operating system.

Concerning the knowledge of Object-Oriented mechanisms, we advise the reader Bertrand Meyer's :

> *Object-Oriented Software Construction*;
> Prentice Hall International Series in Computer Science, C.A.R. Hoare series editor;
> Englewood Cliffs;
> 1988.

This provides a good introduction on Object-Orientation, with references to the **EIFFEL** Object-Oriented language.
The second book we advise is Brad J. Cox' :

> *Object Oriented Programming, an Evolutionary Approach*;
> Addison-Wesley Publishing Company;
> Reading (Mass.);
> April 1987.

This book is of great help for the user as Brad J. Cox is the creator of the Objective-C language. It provides a good introduction to Object-Orientation and to Objective-C.
If one wants a more 'language centered' approach, he can read the first part of Adele Goldberd and David Robson's :

> *Smalltalk-80, the language and its implementation*;
> Addison-Wesley Publishing Company;
> Reading (Mass.);
> 1988.

This book provides a good explanation of the Object-Oriented mechanisms with the always present concern of applying them directly to Smalltalk-80.

Concerning the knowledge of the C language, one can find a tremendous lot of books. Among the ones we consulted, we advise :

> KERNIGHAN, Brian W., Dennis M. RITCHIE;
> *The C Programming Language*;
> Prentice-Hall;
> 2nd ed. Englewood Cliffs;
> 1988.
>
> Herbert SCHILDT;
> *Teach Yourself C*;
> Osborne McGraw-Hill;
> Berkeley;
> 1990.
>
> Bruce HUNTER;
> *Introduction à C*;
> Translated from English by Dominique Pitt;
> Sybex;
> Paris;
> 1986.

Concerning the knowledge of Objective-C, we already named Brad J. Cox' book, "Object Oriented Programming, an Evolutionary Approach". We would like also to advise the reading of the Objective-C manual that provides a very useful '*thirty minutes tutorial*' :

> THE STEPSTONE CORPORATION;
> *Objective-C compiler with IC pack 101: Foundation class library. Objective-C 3.3 Reference Manual*;
> Productivity Products International Inc;
> 1988.

Concerning notions of Unix, the reader can go through the documentations of the Unix language provided with the language itself, but this seems to be a very long task indeed. So we advise Steve Bourne's :

> *The Unix System*;
> Addison-Wesley Publishing Company;
> Reading (Mass.);
> 1982;

for a general survey on how to use the Unix operating system.
For a conceptual approach of Unix, we advise James Groff and Paul Weinberg's :

> *Unix, une approche conceptuelle*;

Translated from English by Dimitri Stoquart;
QUE InterEditions;
Paris;
1989.

To conclude, we would like to introduce some works concerning the RMG environment. The first one constitutes a guide for the inexperienced user :

Zvonko FAZARINC;
RMG User's First Aid Kit;
Hewlett-Packard company;
Palo-Alto (Calif.);
1989.

The second one contains all the references to RMG basic classes :

HEWLETT PACKARD COMPANY;
RMG, a Tool Kit For Development of Visualization Courseware;
Reference Manuals N° 1 and 2;
Hewlett-Packard Company;
Palo-Alto (Calif.);
1989.

The two following ones can be of a certain use for a more experienced user. These are two thesis written by two German students who worked within the COLOS project :

Uwe HEIMBURGER;
Entwicklung zweier interaktiver Simulationsanwendungen zu Lehr- und Lernzwecken unter Verwendung einer objektorientierten Graphikumgebung: Benutzerschnittstelle und Realisierung physikalischer Gesetze;
Institut Für Informatik und Praktische Mathematik, Christian-Albrechts-Universität;
Kiel;
Mai 1990.

Detlev WEGENER;
Entwicklung zweier interaktiver Simulationsanwendungen zu Lehr- und Lernzwecken unter Verwendung einer objektorientierten Graphikumgebung: Konzeption und Realisierung der grundlegenden Klassen;
Institut Für Informatik und Praktische Mathematik, Christian-Albrechts-Universität;
Kiel;
Mai 1990.

Chapter 6: How to create a new application in RMG ?

In this chapter, we see how one has to proceed to create a new application in the RMG environment. We first explain shortly what is an RMG application (6.1). Then we talk of what has to be included in a class description and is different of Objective-C (6.2). The next point encompasses the creation of an RMG application without the use of RMG tools (6.3). And the last point takes in concern the creation of an application using RMG tools (6.4).

6.1. What is an RMG application ?

We explain in this part, without entering implementation details, what is an RMG application.

The interior of an RMG application is composed of **one or many classes** written in Objective-C. Let us just say for the moment that these classes can be already existing classes -RMG or Objective-C classes- or classes newly created by the programmer.

The relations existing between classes in an application can be multiple. Among these relations, one usually finds :

- an **inheritance relation**;
- a **utilization relation.**

The first one implies that one of the classes involved in the relation is subclass of the other one. The second one implies that the two classes do not have any direct inheritance relation, but they can instantiate one another and when they are instantiated, they use one another through messaging.

In the set of classes composing an application, the programmer has to choose a '**master-class**'. It is the first class instantiated when the application is started by the user selecting its name in the '*NEW* >' menu item submenu of the environment's main menu. From this class, starts the instantiation of each class directly linked to it by a utilization or inheritance relation in the application and necessary to the progress of the application. We said 'directly linked' because each class instantiated in the case of an application can also instantiate any class necessary to the good progress of the application. Furthermore, this master-class has the responsibility of creating the application's window on the screen. In other words, the master-class plays the role of **manager of the application,** it is the black-box, the main controller of the application. To compare it to Objective-C, it is really, in combination with the environment, the main program we talked about in chapter 3, necessary to enable the 'testing' and the use of Objective-C classes.

Each master-class, in order to play correctly its role in the environment, has to be a subclass of a class called '**Envir**'[1] or of any class below Envir and linked to it in the

---

[1] : This class is detailed in the following chapter

inheritance tree. As says [HP, 89] (Envir, p.1) : "*This is the main environment window. It is the background for all the additional applications that are brought up.*". This 'Envir' class, directly or through inheritance, handles the screen management. It enables the programmer to use some already existing methods that permits him to :

- move a window on the screen;
- enlarge a window;
- etc.

This class also handles, directly or through inheritance, the mouse management. It enables the programmer to use some methods that permits him to :

- test the state of each mouse button;
- know the coordinates of the RMG cursor's location;
- etc.

In short, this class supplies the programmer with methods enabling him to manage the mouse and cope with an application window. Note that it is not compulsory that the other classes eventually composing an application be subclasses of 'Envir'; they can be subclasses of the master-class or of any other class. Note also that the name of the master-class has to be the same as the application's name.

We talked above of the master-class being instantiated. This is done when the user selects the application name from the '*NEW* >' menu item of the environment's main menu. At this moment, the environment sends a message to the master-class of the application. This message contains the name of the master-class and a selector which is always the same: '**newIn:**'. This selector identifies <u>in each master-class</u> the factory method that instantiates this class and thus starts the application. As says [FAZARINC, 89] (p.76) : "*This* -newIn:- *contains the instructions for creation of a new instance of the subclass of Envir.*". We see in a following point which are the other additional items a class must absolutely contain in RMG and which are different from those in an Objective-C class.

We insisted earlier on the fact that the menu is one of the important communication means of the RMG environment. This is why nearly each application has its own main menu. This menu is created once at the instantiation of the master-class, through the instantiation of a menu class which is integrally part of the set of classes composing an application.

This menu class is always subclass of the '**EnvMTree**' class, which is the class that enables the creation of a menu for the environment itself and for an application of the environment. Further details are given about menu classes in a following chapter.

The other classes composing an application can not be described precisely here. They are entirely the programmer's creation and depend highly of the application itself.

6.2. What to include in a new class

We see in this point which are the things one has to include in a new RMG class description. There are few differences between Objective-C and RMG concerning what must be included in a class description; though the overall syntax does not change and one can always refer to chapter 3 for more precisions.

When one starts the definition of an RMG class, one has to specify, like in C, the names of some **header files** necessary to RMG. Recall that a header file is a file containing information about the standard library functions, that they all end with a '.h' extension and that these files are included in a program using the preprocessor directive '#include'. Three header files are compulsory in an RMG class description :

- '**objc.h**';
- '**rmg.h**';
- '**envir.h**'.

The first one can be found in the directory /usr/local/lib/objc3.3, is supplied with the Objective-C compiler and contains some special definitions of data types and variables. The two following ones can be found in the directory RMG/INCLUDE and define special RMG variables, structures, etc[2]. In addition to these three header files, the programmer can include other ones depending on the application. For example, one can include 'math.h' if one uses mathematical functions[3]. So this declaration is of type :

```
#include "objc.h"
#include "rmg.h"
#include "envir.h"
[#include "<other header file name>"]*4
```

The next step involves the declaration of global and static variables, just like in C. Usually, all the global variables are defined as static variables, this enables the suppression of name conflicts between variables or constants included in header files and the other ones declared in the classes description files themselves.

This is followed by the '**@requires**' clause. This clause lists all the classes used in the application, other than the superclass of the programmer's class. It takes the shape :

```
@requires <class name> [,<class name>]*
```

[2] : Note that the user can directly access these files to consult their content

[3] : For further informations on the available C header files, the reader can consult the C user manual

[4] : Where *[]* means that the enclosed items are not compulsory and * means that this item can be repeated

Note that this line is not obligatory as long as the programmer does not make any use of other classes than the superclass of his class.

Then follows the declaration of the **class name**, its **superclass** and the **message groups**, just like in Objective-C. Note again that if the class is a master-class, the superclass is 'Envir' or any class below 'Envir' and attached to it through inheritance. The declaration is of type :

= <class name>: <superclass name> (Message Groups)

Four groups are obligatory in the 'MessageGroups' list :

- '**working**';
- '**Collection**';
- '**Primitive**';
- '**RMGVW**'.

The first three groups are relative to the Objective-C classes -see chapter 3. The last one is relative to RMG base classes such has 'RMGView' for example -see chapter 7. To these message groups must be added the user message group, as in Objective-C.

The programmer lists next the **instance variables** or private data of an instance. One could declare each variable used within a class as instance variables but unfortunately this would occupy to much memory space as the whole set of variables plus the inherited ones would have to be contained in memory for each instance. That is why some variables :

- that are common to all instances but are not considered as private data are declared before the require clause;
- that are considered as local variables can be declared within a method itself, just as local variables can be declared inside C functions.

Now comes the **method definitions**, both factory and instance methods. Their syntax is exactly the same as in Objective-C. But in the case of a master-class, one has to include special methods :

- '**newin:**';
- '**delayInit**';
- '**extraNewIn:**'.

The first one, 'newIn:', is a factory method -thus prefixed by '+'- which goal is to create a new instance of the subclass of 'Envir'. When a call from the environment is addressed to the class in order to instantiate itself -through the 'newIn:' factory method-, the address of the environment is passed as argument to the method in the variable 'anEnvironment'. Note that this variable name is arbitrary as long as it is

being used consistently. The shape of the 'newIn:' method is :

```
+ newIn: anEnvironment
  {
  [self delayInit];                    /*1*/
  self = [self <method name for window creation>];
                                                   /*2*/
  ...
  return self;
  }
```

The first thing done is '[self delayInit]' (line 1). Recall that self, at instantiation of a class contains the address of this class. This is thus a message sent to the class 'asking for execution' of another factory method: 'delayInit'. This method is described below.
The second thing done is the creation of the instance of the class as a window : 'self = [self <method name for window creation>]' (line 2). The method identified by the selector taking the place of '<method name for window creation>' is a method enabling the creation of a window, a method that the class can access through inheritance. We see in a following chapter which are the inherited methods that one can use for this creation. The result of the creation of the instance -an address- is stored, by convention, in the variable self -see chapter 3. In short, one has just to remember for the moment that this line is the place where the instance of the class is really created, in the shape of a window.
The '...' denotes the fact that many other things can be done in this method, among which one can find :

- creations of subwindows;
- initialization of global or local variables;
- definition of actions[5];
- etc.

The second method that must be included in a master-class is 'delayInit'. It is also a factory method, which is called at the beginning of the 'newIn:' method. Its goal is to instantiate the only instance of the menu class, in order to create the application's menu, if it has a menu[6]. Its shape is usually :

```
+ delayInit
  {
  static BOOL beenHere = FALSE;              /*1*/

  if (!beenHere)                            /*2*/
        {
        if (self == <class name>)           /*3*/
```

[5] : Explanation about actions is given in a following chapter

[6] : Indeed it is not indispensable for an application to have a menu as long as all the actions that would have been performed through a menu are present in the application under the shape of buttons or icons

```
                {
                [<menu class> delayInit]; /*4*/
                beenHere = TRUE;          /*5*/
                }
                else
                [<class name> delayInit]; /*6*/
        }
    return self;
    }
```

This method uses a static local variable (line 1) defined as a boolean, to keep track of possible previous visits, as this method has only to be executed once per instantiation of the master-class.
Line 2 checks if a visit has already been made. If this is the case, the method is directly skipped; if not, it checks the 'identity' of the caller (line 3). If it is not the class itself, the message is sent again to the class itself (line 6).
Line 4 is one of the common things one can find in this method, a message sent to a menu class in order to create an instance of it for the application.
Line 5 sets the variable beenHere to TRUE in order to avoid other visits.

The last method is an instance method -prefixed by a '-' -, 'extraNewIn:'. This method is used whenever one wants to do special initializations. It takes the shape :

```
- extraNewIn: anEnvironment
    {
    [self showAll];                       /*1*/
    menuTree = [<menu class> getIt];      /*2*/
    ...
    return self;
    }
```

Line 1 is the message sent to the instance itself, which selector corresponds to a method contained in Envir's superclass: 'RMGView'[7]. This line contributes to make every created window appear on the screen.
Line 2 calls up the menu tree for the instance and stores its address in the variable 'menuTree' which is an instance variable inherited from the class 'Envir'.

Of course any other factory or instance method may be added by the programmer in addition to these three. Note that the class description ends also with the sign '=:'.

Table 6.1 gives a summary of what has to be included in a class description.

[7] : The class RMGView is detailed in chapter 7

Header Files	#include "objc.h" #include "rmg.h" #include "envir.h" [#include "<other header file name>"]*
C variables declaration	*Declaration of global variables*
Declaration of classes used	@requires <class name> [,<class name>]*
Class name declaration	= <class name>: <superclass name> (<personal message group>, RMGVW, Primitive, Collection, [,<other message groups>]*)

Instance variables declaration	{
	Declaration of instance variables
	}

Methods definition	+ newIn: anEnvironment {...} + delayInit {...} *Any other factory method* - extraNewIn: anEnvironment [8] {...} *Any other Instance method*
End of class description	= :

Table 6.1 What has to be included in a RMG class description

---

[8] : The three methods specified are only obligatory in the case of a master-class

6.3. Programming without RMG tools

We see in this part which are the different steps one has to follow to create an RMG application without using the RMG tools described briefly in point 4.1.2.

We have isolated three different steps. The first one concerns the edition of the class description. The second one concerns the compilation and linking of the classes. Finally, the last one concerns the inclusion of the application in the environment's main menu.

6.3.1. *Editing*

The first thing to do when one wants to create a class description file is to 'write' it or edit it, thus using an **editor**.

The only things we have to say is that every editor available under Unix is of course valid, for example 'Vi' or 'Emacs'. The thing to bear in mind is that the class description must be absolutely saved with a '.m' extension.

6.3.2. *Makefiles, mainClasses and compilation*

After creating a class description file, one wants to compile it. In order to facilitate this process, one uses three particular files: two '**makefiles**' and a '**mainClass.m**' file.

The compilation of a class description file involves the invocation of the ObjC compiler in order to obtain a relocatable class definition. Then this file has to be added to a library archive and linked to the other libraries. As says [FAZARINC, 89] (p.80) : "*The process is quite involved and not particularly interesting, but does require additional skills*". So to make life easier, we are going to use two makefiles. These files are sorts of scripts which purpose is to compile class descriptions and archive them in a library. They must also manage the distribution of message declaration files and remove all unnecessary compilation remnants.

This process is divided into two parts. The first part involves the compilation of one's new classes in one's own directory. This involves a makefile contained in one's individual directory. The second part involves the reconstruction of a new environment. This involves a makefile found in the RMG/ENVIR directory and which is described later.

Table 6.2 gives a template for an individual makefile[9].

---

[9] : Note that the reader can find in the appendices a full example of each file discussed in this part

```
COMPILE = objcc3.3

OPT = -q -O -N -nRetain -I../PANEL -I.. -I../../CP -I../../INCLUDE
OPTISL = -I/usr/local/lib/ink

<message group name>.a:<message group name>.a(<class name>.o)\
                        <message group name>.a(<class name>.o)\
          ...
          <message group name>.a(<class name>.o);
          objcc3.3 $(OPT) -c $(?:.o=.m)
          ar ruv <message group name>.a $?;
          cp [CP]_* ../CP
          /bin/rm -f $?
```

Table 6.2 Template of an individual makefile

Explanation of the total makefile presented in table 6.2 is not relevant here. We just want to say that '<message group name>' with the '.a' extension corresponds to the user's message group name. Every class using the message group must be listed following the scheme: '<class group name>.a(<class name>.o)', where '<class name>' is the name of the class. Note that the '\' symbol means that the declaration is continued on the following line. This file can be saved into a file named 'makefile' for example.

To compile its classes, the user has only to issue the command 'make <message group name>.a' at the unix prompt; the compiler shows on the screen which step it is currently going through. If the compilation fails, the user has to watch carefully the screen for the causes of this failure. Indeed, the errors -Objective-C errors or C errors- are usually reported with the number of the line where they occur in the class description file and the type of error. The user just has to go and correct them in the source class description file.

The next step for the user is to notify RMG that new classes are being added and that they must be linked into the system. All the informations concerning this notification can be found in a file called 'mainClass.m' into the RMG/ENVIR directory. We reproduce a portion of the mainClass.m file in order to show the user what he needs to add to it (see table 6.3).

```
...
#define RMG_CLASSES SysIcon, RMGLine, RMGVLine,
    RMGHLINE,\
...
#define COLOR_MAP ClrMapEdit, ClrMTree
...
#define <macro name> List all classes belonging to the message group
                     corresponding to the macro name          /*1*/
...
@classes(
    RMG_CLASSES,
    #ifdef STAGE1
        COLOR_MAP,
    #endif
...
    #ifdef STAGE18
        <macro name>,
    #endif
        PPI_CLASSES
```

Table 6.3 Interesting portion of the mainClass.m file

In table 6.3, the '<macro name>' is a name in capital letters that is usually the same as the message group name. Line 1 corresponds to the definition of a macro; this definition means that from now on this macro name can replace the list of classes placed at its side.

Now that we have changed the 'mainClass.m' file it is time to change the makefile placed in the RMG/ENVIR directory. We present a portion of this file at table 6.4. If one looks through this file before changing anything, one notices that its STAGE -STAGE18 in our case- has been identified with its macro name and message group name with a '.a' extension -see line 1. Two lines are missing and must be added, line 2 and 3.

Now that these different files are modified, we are ready to ask the **rebuild** of a new environment including our new classes and message groups. In order to do just that, the user has to issue the two following commands at the Unix prompt and while in the RMG/ENVIR directory :

make cp
make env.out

```
...
STAGE15 = PTY/pty.a
STAGE18 = <macro name>/<message group name>.a    /*1*/

ALL = $(STAGE1) \
      ...
      $(STAGE15) \
      $(STAGE18)                                 /*2*/

ALLSTAGE = -DSTAGE1 \
           ...
           -DSTAGE15 \
           -DSTAGE18                             /*3*/
```

Table 6.4 Portion of the environment's makefile

The first command touches the mainClass.m file and forces its recompilation. The second one causes a recompilation of modified class descriptions and the relinking of the environment. Note that the entire process appears on the screen with the program 'telling' the user which step it is currently going through.

6.3.3. *Including the application in the RMG environment*

Now that the environment is successfully recompiled including one's new classes, one has still to **include the application** in the RMG environment and particularly in the main menu of the environment.

A list of all the applications appearing in the '*NEW* >' menu item of the environment main menu is maintained in a file called '**A.menu**'. This file is situated in the directory RMG/ENVIR/CONFIG. We present a portion of this file at table 6.5.

```
30 12 10
AnalogClock
BarMeter
...
Voltmeter
PtyApp
/*30 items;
  maximum 12 characters;
  display 10 items on screen */
/* note the names above must correspond to class names */
```

Table 6.5 Portion of the A.menu file

Each name in the file represented at table 6.5 is the name of an application with the restriction that this name must correspond to the name of the master-class, as specified at the last line of the file. When one wants to add an application in the environment's main menu, one has to add the master-class name in this file, preferably in alphabetical order. The user must not forget to update the first number of the first line which denotes the total number of items that appears in the submenu of the '*NEW* >' menu item.

When this is done the user can start the environment and see that his application is contained in the main menu and can be started.

6.4. Programming with RMG tools

Programming with the tools included in the RMG environment is another solution. Indeed, it seems less tedious to be able to work entirely from the environment in place of getting out of the environment, getting to one's directory, running the editor, correcting the file(s), recompiling it/them, etc.

We discussed shortly in a preceding point -point 4.1.2- some of the different tools available in the environment. It is our purpose in this point to show the user how he can do some traditional programming -editing, compiling, running- directly from the environment. It is not our purpose to develop explanations about the prototyping tool -called the 'Fast Prototyping Facility'- or the program interpreter. One of the reasons is that we did not have the occasion of using in detail both these tools.

Three steps have been underscored. The first one concerns the edition of class description files. The second one concerns the compilation of the classes. The last one concerns the different browsers one can use to facilitate one's work.

Before starting our explanations, we must attract the attention of the reader on the fact that programming directly inside RMG can only be done on an already existing application. It is not possible for a programmer to design directly new classes and thus a new application, from the environment itself. This is due to the fact that to take into account new applications, the environment has to be recompiled completely in order to take into account the changes made in certain files -'A.menu' for example- and the presence of new classes. This can not be done from the environment itself. Thus the tools available inside the RMG environment enable only this environment to take into account changes made to classes composing applications already present in the environment. It is thus a process that enables the programmer to fix existing applications but not to create them from scratch.

6.4.1. *Editing*

In order to do some file 'editing', RMG furnishes a very powerful tool called '**DocEdit**'. This editor has a set of 'Emacs style' commands and many others uncommon ones.

This application is called by the user selecting 'DocEdit' in the '*NEW* >' menu item of the environment's main menu. A window appears on the screen, containing a certain number of important items.

The first item is the border of the window; if it is green it means that the window is not active and that the user is not able to type-in some text. To change this, the user just has to click inside the window; the border turns red, sign of the window being active.

At the top left of the window one can see the position of the RMG cursor in terms of lines and columns, in a little black box.

At the side of this box is another blue box containing the name of the file currently being edited.

At the top right, one can see an icon which is used to enlarge the window. This is done by the user clicking on it and dragging the mouse while holding down the button.

At the left of this icon is another box labeled 'HELP' which, when clicked on makes a window appear containing the description of all the text operations that can be done from the keyboard. For example holding down the 'Control' and the 'K' keys together results on the current line being deleted.

At the bottom and at the right side of the window are placed two scroll bars which can be used in order to make the text go up and down or left and right. This can be done by the user just clicking on one of the arrows or by the user clicking on the white rectangle contained in these scroll bars and dragging the mouse while holding down the button.

Just on the top of the bottom scroll bar are placed a certain number of icons. The first one at the left, a red arrow directed upwards towards an horizontal bar, enables the user to go to the beginning of the currently edited text, when clicked on. The blue arrow at its side enables the user to go to the end of the currently edited text.

Next to these two icons is a box labeled 'FIND:' that enables the user to type[10] in words to be searched across the text being edited. The two boxes labeled 'Prev' and 'Next' enable the user to find the 'FIND:' typed-in word respectively preceding or following the already found one.

When a text is present in the editor two cursors can be seen. The RMG cursor representing the mouse and a red rectangle representing the place where the text is currently typed-in : the **text cursor**.

This application has also a menu (see figure 6.1). The first useful menu item is the '***FILE*** >' menu item. It enables the user to browse through the Unix file system in order to save or load files.

---

[10] : When the user clicks on this box, the RMG cursor disappears and a little flickering white bar appears in the blue rectangle next to the 'FIND:' box. The user has then to type-in his text from the keyboard and hit the 'return' key in order to start the search

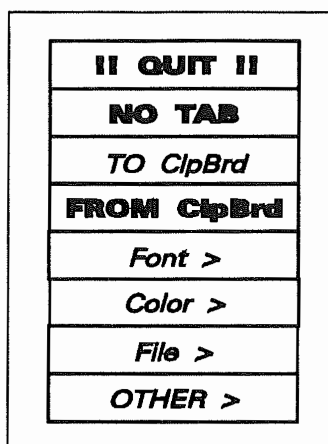


Figure 6.1. DocEdit's main menu

One can find a lot of other menu items in the submenu hidden by the '*FILE* >' menu item. For example, by going through the submenus of the 'BROWSER >' menu item, the user is able to travel through the Unix directory system and select a particular one. Going through the submenu of the '*READ from DIR* >' menu item, the user is able to select a file to load into the editor. The files displayed in the submenu are the ones contained in the currently selected directory; this directory can be checked with the '*R/W Directory* >' menu item. The user is also able to specify the name of a file he wants to load, just by typing it using the '*READ keyboard* >' menu item. The 'save >' menu item enables the user to save a file under its current name or by specifying a new name for it.

When one has loaded a file in the active editor window, one is able to work with the text just as in a normal editor. The menu offers additional possibilities to the user. By going through the set of menu items '***TO ClpBrd*** >', 'StrClpBrd', then moving the RMG cursor next to the text cursor and clicking while dragging the mouse to the right, the user can see a red highlight following the motions of the mouse. When the button is released, a new window appears labeled 'Text Clip Board' and containing the text the user currently highlighted.

By positioning the text cursor[11] somewhere in the DocEdit window and selecting the menu item '**From ClpBrd**' from the application's main menu, the user causes the appearance of the text transferred to the Text ClipBoard, at the location of the text cursor.

One is also able to change the font of the text currently displayed. This is done by selecting the menu item '***FONT*** >' and selecting a font type in its submenu. The characters and background colors can be also changed using the '***Color*** >' menu item.

[11] : This is done by positioning the RMG cursor at the desired location and clicking once

6.4.2. *Compilation*

After editing the class description file, one needs to compile these classes and to tell the environment that one wants the changes active in the environment; all this without going out of the environment. One thing that the user must keep in mind is that the environment can not be recompiled from itself. So the process only involves the recompilation of the changed classes and a dynamic linking of these classes in the environment.

The process of recompiling classes from the environment is called compiling 'dotrfiles', that is files with '.r' extensions. In order to do this correctly, things must be added at the end of one's individual makefile, before entering the environment, see table 6.6.

```
dotRFiles =  .r

DOTRFILES = <name of class>.r ... <name of class>.r ;  /* 1 */

TrainingAppl : $(dotRFiles) ;
        ld -dr $(dotRFiles) -o TrainingAppl ; /bin/rm $(dotRFiles)
        cp [CP]_* ../CP
```

Table 6.6 Additions to be made to one's individual makefile

One must list at line1, each class for which one wants recompilation available in the environment.

Inside the environment, the user must select the application '**PtyApp**' in the main menu. A window appears on the screen which gives direct access to the Unix environment. The user must go to his directory using the Unix 'cd' command and launch the compilation of the dotrfiles by typing :

make DOTRFILES

If the compilation fails for a reason or another, the user must go back to the editor, correct the errors, save the file and recompile them as dotrfiles again. This must be done until no more errors are detected.

When this step is finished successfully, one needs to tell the environment that changes have been made to some classes and that the user wants these changes to be active. This is called dynamic linking and is done through a menu item of RMG's main menu: '**Dyn Link >**'. The user selects this menu item and accesses

the character box[12] hidden behind it. In this box, the user must type the name of the class that has been changed followed by the '.r' extension. This must be done separately for each changed class. From now on, the user is able to use the newly modified application.

There are a certain number of restrictions that the user must bear in mind. A first one is that this process can not be used when the classes are not yet linked to the libraries and the environment entirely recompiled with these classes, as explained above. Indeed this could cause things to go wrong or things not to work at all.

The second important thing not to forget is that the dynamic linking is only valid in the current environment. That is if the user quits the environment and restarts it, the changes are not considered as the environment has not been recompiled completely. So we advise the user to recompile entirely the environment at least once, when he leaves it for example, in order to avoid bad surprises.

When linking dynamically error messages of the type:

Invalid output file

can occur. In this case the user has to return to the 'Dyn Link >' menu item and check that the extension of the file is '.r'. In the case of other errors, the right solution is usually getting out of the environment and recompiling it entirely. If it still does not work, the user must look at the various makefiles and the mainClass.m file in order to check their correctness.

The last thing is that this process must be used with precautions and we advise the user to often quit the environment and recompile it entirely because too frequent compilation inside RMG and dynamic linking of different classes can lead to 'misfunctioning'.

6.4.3. *Browsers*

In order to help the user fix his classes, RMG furnishes a set of browsers, as introduced in chapter 4. These browsers include :

- a class browser;
- an instance browser;
- a message browser.

---

[12] : A character box is a menu item submenu, which displays a white cursor when it is selected and that enables the user to type-in some text from the keyboard. The text input can be stopped by the user hitting the 'Return' key or depressing once the left-hand side button of the mouse

The first one enables the user to navigate across the classes available in RMG. This can help the user find a class and look at the different methods it implements and the different instance variables it declares. The class browser can be run selecting '**ClassBrowser**' in the submenu of the '*New* >' menu item.
The second one is a tool designed to track and graph the evolution of instances at run time. It enables the user to display the different addresses and values of instance variables owned by instances of classes present in the environment. The instance browser can be run selecting '**InsBrowser**' in the submenu of the '*New* >' menu item.
The last browser enables the user to find the origin and content of a given message. The message browser can be run selecting '**MsgBrowser**' in the submenu of the '*New* >' menu item.

6.5. Summary

The goal of this chapter was to explain how to create an RMG application.

First we explained what was such an application. Thus that it was composed of several classes between which existed relations such as utilization and inheritance. We explained that an application or set of classes with relations between them had to have a master-class representing the application in the environment; this master-class being obligatorily directly or indirectly subclass of an RMG class called Envir which handles the mouse and screen management.

A second point presented concerned what had to be included in a class declaration and that was different from Objective-C.

A third point presented the way to program applications not using the RMG tools. That is :

- using a Unix editor to create the class description file;
- using makefiles and a file called 'mainClass.m' to compile the classes and link them to the existing libraries;
- including the new application in the environment's main menu using the file 'A.menu'.

The last point concerned the programming of applications using the RMG classical tools which are :

- 'DocEdit' for text editing;
- 'PtyApp' and the 'Dyn Link >' menu item for the compilation of dotrfiles;
- 'ClassBrowser' to navigate across the different classes available in RMG;
- 'InsBrowser' to track the evolution of instances in the environment;
- 'MsgBrowser' to find the origin and content of a given message.

We saw that this way of creating applications could be easier because of all the tools available at hand but had to be used carefully as the inclusion of changed classes was not permanent due to the use of dynamic linking.

Chapter 7: First useful classes

Now that we have seen what was an RMG application and how to create one, it is time to review some of the basic and most useful classes available in RMG. We list and explain some of their more commonly used and more interesting methods. Other methods implemented in these classes are explained in a later chapter while introducing new notions.

A first point concerns the class 'RMGView' (7.1). A second point concerns the class 'Envir' (7.2). A third point describes the 'RMGString' class (7.3). Finally the last point concerns the 'RMGIcon' class (7.4).

7.1. RMGView

RMGView[1] is one of the basic classes of RMG and the **foundation of most RMG classes**. This class enables the management of the overall screen, while some of its subclasses manage only specific parts of screen display. In short, it contains the methods for the creation and modification of windows.

The windows resulting of the instantiation of this class are organized in a **superview-subview hierarchy**; the subviews are placed inside and on the top of the superviews and the views created later are placed on the top of the already existing ones.

RMGView is a subclass of the Objective-C root class: 'Object'. It includes some instance variables enabling to store some parameters concerning the views[2]. For example 'background_color' contains the color number of the view; 'subviews' contains the address of an ordered collection[3] containing the addresses of the subviews of the current view; etc. It also includes approximately 10 factory methods and 130 instance methods.

We first present the different interesting instance variables, then the most used factory methods and finally the different instance methods.

7.1.1. *Instance variables*

This class contains a lot of **interesting instance variables**. Note that this class implements instance methods enabling access to nearly each of these variables.

The first instance variable is '**superview**', which is of type id and is designed to contain the address of the instance's superview. This variable must not be

---

1 : Note that this class is called RMGView <u>or</u> RMGView0, depending on the version of RMG and on the RMG reference manual available

2 : Note that from now on we will use the term views. This will invariably denote an application window or any other subwindow placed within it

3 : A collection is an Objective-C object enabling the management of an arbitrary number of objects as a unit. In this case, an ordered collection keeps also the order in which the objects are stored. See [STEPSTONE, 88] for further details

confused with Objective-C's pseudo-variable 'super', which enables an instance to 'access directly' its superclass.

Another one is '**viewIcon**' which is also of type id and is designed to contain the address of iconized objects. But this variable is very often used to store other addresses, so used like a drawer where the programmer can put whatever address he will need later.

Three other instance variables available to the programmer are: '**active**', '**erased**' and '**covered**'. These three variables are booleans and are designed to denote the state of a view. The first one, 'active', indicates if the view is active or not; it is usually used in the case of the 'DocEdit' application -see chapter 6. 'erased' indicates if the view is erased or not and 'covered' if the view is partially covered by another one.

Height other instance variables give the outer and inner bounds of a view: '**out_left**', '**out_right**', '**out_low**', '**out_high**', '**in_left**', '**in_right**', '**in_low**', '**in_high**'. The difference between the outer and the inner bounds is the width of the colored frame which can be set around a view. Note that the origin of a view is the lower left corner. Note also that every view measurement is given in pixels.

7.1.2. *Factory methods*

Of course among the **factory methods**, one can find 'newIn:' which in this case creates by default an instance of RMGView represented on the screen by a view of 200 by 200 pixels and originated at the (0,0) location, thus at the lower left corner of the screen. Note that from now on, when methods are presented for the first time, the type of the arguments will be enclosed in brackets.

The more interesting factory methods are :

+ **origin:**(int)**:**(int) **extent:**(int)**:**(int) **superview:**(id) **bkgd:**(int)
+ **relative:**(int)**:**(int) **extent:**(int)**:**(int) **superview:**(id) **bkgd:**(int)
+ **Type:**(int) **extent:**(int) **superview:**(id) **bkgd:**(int)

The first one enables the creation of a view which is placed on the screen following absolute coordinates at a location specified by the two first arguments, with certain dimensions specified by the third and fourth arguments, within a certain superview -the fifth argument- and with a certain background color -the last argument. For example, if we look at a factory method 'newIn:' that could be found in a class, we could see :

```
+ newIn: anEnvironment
    {
    ...
    self = [RMGView    origin: 100: 100
                       extent: 200: 200
                       superview: anEnvironment
```

```
                bkgd: 1];
    ...
    }
```

where a view of 200 by 200 pixels is created (see figure 7.1) at location (100,100) on the screen and is placed in the superview anEnvironment, which is in our case the environment itself[4].

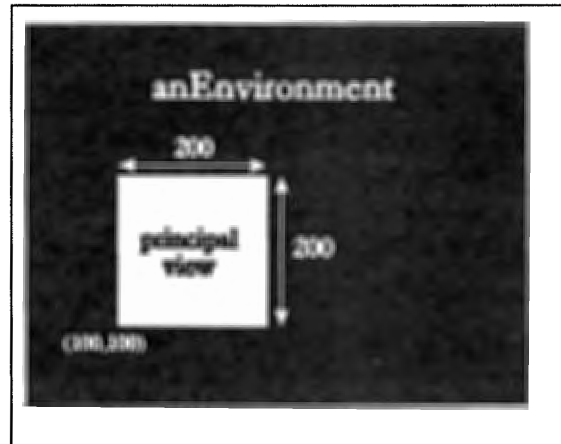


Figure 7.1 Creation of a view with the '+ origin:: extent:: superview: bkgd:' factory method

Finally the color of this view is 1, which is orange in RMG[5]. (**Example 7.1**)
Note that with this method, the views created can not exceed the bounds of their superviews. Note also that when a view is enlarged, all its subviews are enlarged proportionally.

The second factory method uses the same principles with the exception that the views are created following relative coordinates. We can extend example 7.1 and have:

```
+ newIn: anEnvironment
    {
    id aView;                                              /* 1*/
    ...
    self = [RMGView    origin: 100: 100                    /* 2*/
                       extent: 200: 200
                       superview: anEnvironment
                       bkgd: 1];
    aView = [RMGView relative: 50: 25                      /* 3 */
                       extent: 100: 100
```

[4] : Thus the variable 'anEnvironment' represents the environment's address

[5] : It is possible for the user to use the color names instead of numbers. In this optic, the reader can consult the header file 'rmgColors.h' situated in the RMG/INCLUDE directory

```
                                        superview: self
                                        bkgd: 2];
        ...
        }
```

Line 1 is the declaration of a local id variable that will contain the address of the second view we want to place on the screen.
Line 2 is the creation of the view we described in example 7.1; we will call it our principal view.
At line 3 we find the creation of a subview of our principal view. This subview is placed following relative coordinates at location (50,25)[6] in the principal view -as the specified superview is self which contains the principal view's address-; it has a dimension of 100 by 100 pixels and a red color (see figure 7.2). (**Example 7.2**)

The last interesting factory method is used to create a view that 'sticks' to one edge of its superview. This view sticks either to the left, right, bottom or top of its superview, as specified by its first argument. The second argument specifies the extent of this view and corresponds to :

- the height of the view in the case of a view sticking to the top or to the bottom edge of another one. The width of the view is in this case the length of the edge to which it sticks;
- the width of the view in the case of a view sticking to the right or to the left edge of another one. The height of the view is in this case the length of the edge to which it sticks.

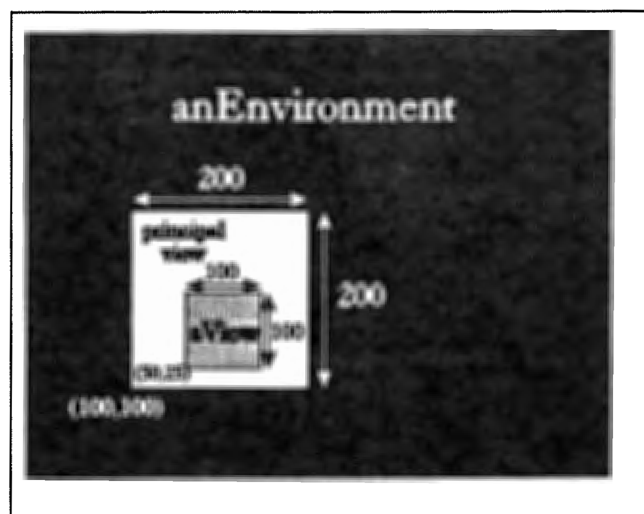


Figure 7.2 Creation of a subview of the principal view with the '+ relative:: extent:: superview: bkgd:' factory method

[6] : Its absolute coordinates being thus (150,125)

We can extend example 7.2 to have :

```
+ newIn: anEnvironment
    {
    id aView, bView;                                        /* 1*/
    ...
    self = [RMGView     origin: 100: 100                    /* 2*/
                        extent: 200: 200
                        superview: anEnvironment
                        bkgd: 1];
    aView = [RMGView relative: 50: 25                       /* 3 */
                        extent: 100: 100
                        superview: self
                        bkgd: 2];
    bView = [RMGView Type: 4                                /* 4 */
                        extent: 50
                        superview: aView
                        bkgd: 0];
    ...
    }
```

Line 1 is the declaration of two local id variables which will contain the addresses of the subviews we want to create.
Line 2 is the creation of the view we described in example 7.1. At line 3 we find the creation of a subview of our principal view.
Line 4 is the creation of the sticky subview (see figure 7.3.) that we place in the view identified by aView.

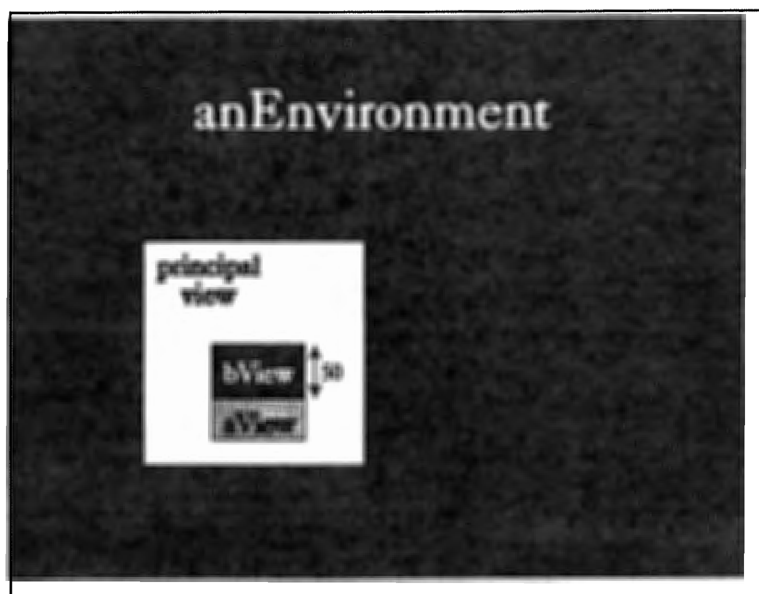


Figure 7.3 Creation of a sticky subview of aView with the '+ Type: extent: superview: bkgd:' factory method

This subview is placed at the top -... Type: 4 ...-; it has a height of 50 pixels and is black. (**Example 7.3**)

7.1.3. *Instance methods*

Among the multiples **instance methods** implemented in this class, some are of particular interest for a programmer. We review some of them, keeping in mind that other ones are introduced later, together with other subjects.

The first methods are '**insideContains:**(int)**:**(int)' and '**contains:**(int):(int)'. These methods check if the location identified by the two integers passed as arguments is contained within respectively the inner and outside bounds of the view. If this is the case it returns TRUE otherwise it returns FALSE. For example, we could send the instance of RMGView -created at example 7.3- which address is contained in aView, the message :

[aView contains: 175: 150];

The return value would be of course 'TRUE' as the point located at (175,150) is indeed situated inside the bounds going from (150,125) to (250,225). Note that these locations are measured in absolute coordinates. (**Example 7.4**)

Two other instance methods enable someone to respectively create a frame around a view and to give it a certain color. These methods are called : '**frameWidth:**(int)' and '**frameColor:**(int)'. For example, if we add a red frame to our principal view created at example 7.1, we have :

[[self frameWidth: 10] frameColor: 2];

Note that in this case the outside bounds of our principal view are unchanged, but the inner bounds are reduced of ten pixels. Note also that every subview of the view to which we are adding a frame are proportionally scaled, except the sticky views and views occupying entirely their superview.

'**bkgd:**(int)' is yet another method enabling the user to change the color of a view. While '**hide**' is a method that hides the view to which the message is sent and erases its image from the screen.

'**showAll**' and '**show**' are methods used to make views appear on the screen while they have not yet been displayed or have been erased from the screen. The difference between 'showAll' and 'show' is that the latter shows only the view to which the message is sent and not its subviews, to the contrary of 'showAll' which shows the view and its subviews. '**redraw**' is a method that redraws the views, their subviews and any view on top of them. This last method enables usually to activate changes made to a view like color changes or any other graphical changes.

The user is also able to change the place of a view within its superview. This is done with the two methods, '**move:**(int)**:**(int)' and '**moveBy:**(int)**:**(int)' which move the lower left corner of the view respectively to a particular location or by a certain amount of pixels.

The last RMGView method we introduce for the moment is '**size:**(int)**:**(int)'. This method enables the user to resize the view to which the message is sent. All the non-sticky subviews change proportionally and the lower left corner stays at the same place. Note that if the new size of the view exceeds the size of its superview, it is adapted to fit into the superview bounds.

Note that some methods require a 'show' or 'showall' to be sent to the instance afterwards, in order to see the changes appear on the screen.

7.2. Envir

Envir is a class providing the background for all the new applications that are brought up; it is really the class that implements the means to **control the overall state of the mouse**.

This class is instantiated once at the start of the environment and can be seen on the screen as a black window without frame, that covers the entire display. This window can not be moved or reduced and in short represents "*the background of the graphical utilization surface*" ([WEGENER, 90], p.34). This class enables the management of the environment through the use of the mouse. This is why each master-class in an application has to be subclass of the 'Envir' class, to enable the application's window to be managed as the environment itself, thus using the mouse.

We first present the different interesting instance variables, then the most used factory methods and the different instance methods. Finally we present some useful C functions which are implemented in this class.

7.2.1. *Instance variables*

The Envir class has only two instance variables apart from the ones inherited from RMGView. Among these two variables, the only interesting one is '**menuTree**'. This variable is of type id and is designed to receive the address of the menu of the application. Thus it enables to adjoin a menu to any application.

7.2.2. *Factory Methods*

Envir has also approximately 20 factory methods. These methods are not very useful to the reader for the moment so we do not detail them. We just want to say that some of these methods enable the creation of an instance of the environment with particular dimensions and a particular origin location. Another set of them enable to specify different RMG cursors.

The reader must know that the factory methods usually used for this class are the ones of RMGView -as 'origin:: extent:: superview: bkgd:' for example- which are inherited directly.

7.2.3. *Instance methods*

Envir furnishes about 70 different instance methods on the top of the ones inherited from RMGView. We review some of them now; other ones are introduced in a later chapter.

One can find methods very similar to the ones in RMGView and which usually are redefinitions of these methods. For example: '**bkgd:**(int)' and '**redraw**'.

A useful method is '**isId:**(id)', which takes an id as parameter and returns a boolean. The return value is 'TRUE' if the id is the address of an object in the current environment, 'FALSE' otherwise.

'**quit_app**' is a method enabling to quit the application when it is sent within a message to the master-class of the application.

'**moveView:**(id) **whileMouseIs:**(int)' enables the user to move the specified view while the left button of the mouse is in a certain state. This state is specified as argument after 'whileMouseIs:'; this argument is of type integer and can take the value 0 for 'button up' or 1 for 'button down'.

7.2.4. *C functions*

Away with these methods, various C functions are implemented in this class. These functions can be used as in a normal C program.

Among these functions, one can find a certain number of them coping with a timer. Other functions enable the user to move a view on the screen. These are '**moveViewWithBand(aView)**' and '**moveWithBand(aView)**'. The first one moves aView while the left mouse button is depressed. The second one finds the application containing aView and moves it while the left-hand side mouse button is depressed.

'**sprite_erase()**' and '**sprite_show()**' are two functions that enable respectively to erase and to show the RMG cursor on the screen.

'**sizeTopRight(aView)**', '**sizeTopLeft(aView)**', '**sizeBottomRight(aView)**' and '**sizeBottomLeft(aView)**' are four functions enabling the user to enlarge 'aView' from respectively the top right, top left, bottom right or bottom left corner of this view.

Note that, as for the methods, many other functions are available but are of no particular interest at the present moment.

7.3. RMGString

RMGString is a class designed to **display a string** within a particular view and to cope with different parameters concerning this string. For example, it enables the user to change the string color or the string font. As RMGString is a subclass of RMGView, it inherits all its methods and instance variables. To display the strings, it uses HP-Windows fonts which can be accessed by a class called FontMngr.

We first present the different interesting instance variables, then the most used factory methods and finally the different instance methods.

7.3.1. *Instance variables*

The proper[7] instance variables of this class are all relative to strings. The principal ones are two integer variables, '**dot_clr**' and '**bkgd_clr**' which are designed to keep the color of respectively the characters and the background of the string.

'**myFont**' is an id variable which is declared to keep the address of the object font of FontMngr[8], so to keep trace of the font used for a particular string.

'**aString**' is a C pointer to the memory zone containing the string itself and '**str_len**' is the length of this string.

Two other instance variables, '**xoffset**' and '**yoffset**' are designed to contain the coordinates of the string, in the view which contains it.

7.3.2. *Factory methods*

Five factory methods are available.

The preferred one is '**font:**(id) **superview:**(id) **color:**(int)' which creates an instance of RMGString in a particular specified font, in the specified superview and in the specified color. The background color in this case is the one of the superview. The displayed string is specified using an instance method : '**string:**(char*)' which is detailed below.

The font is specified by instantiating the class '**FontMngr**'. When a user requests a font, the font manager checks to see if it has already been loaded. If this is the case, it only returns the address of the font object to the user. If not, it loads the appropriate font by reading the font file on disk and returns the address of the object to the user. For example, '[FontMngr cour12x20];' returns the address of a font of 12 by 20 of 'courrier' type. Note that this class has only factory methods as there should be only one FontMngr instance -concerning one particular font- present in the system.

Usually we use RMGView factory methods to create an RMGString and then set all the parameters such as the string, the color, the font, etc, using instance methods. Note that if no font is specified, the RMGString is not displayed. Note also that if the string is too large for its superview, it is clipped accordingly.

7.3.3. *Instance methods*

RMGString implements about 30 different instance methods. Among these, one can find, as for Envir, methods that are implemented in RMGView but which are redefined considering the particular purposes of the class.

---

[7] : By proper we mean instance variables not inherited through the inheritance chain

[8] : This class is briefly detailed in the next point

Among the principal ones, we can find '**color:**(int)' which enables the user to set the color of the displayed string itself. Note that this method does not redraw the string after having changed the color, this is the user's responsibility.

We can also find '**font:**(id)' which enables the user to set the font of a string. As for 'color:', this method does not redraw the string after having made the transformation.

The most important one is of course '**string:**(char*)', which enables the user to specify the string to be displayed. The argument of this method is a C pointer to a memory zone containing the string itself. Note that this string must be a NULL terminated string; in C that is a string terminated by the '\o' character. If the user changes the length of this string without issuing a 'string:' method to the RMGString instance, the displayed string keeps the original length, thus clipping the newly displayed one. This resizing of the string can be done with the '**resize**' method, which updates the internal instance variable representing the string length.

RMGString provides also some methods to alter the position of the string in its superview. '**center**' and '**centerV**' are two of them. These two methods enable the user to respectively center horizontally and vertically the RMGString's string in its superview. '**scroll:**(int)**:**(int)' is yet another one which takes two integers as parameters. These integers represent the modification of the location of the string, in pixels, by which it will be moved to the right-hand side and upwards. '**dataOrigin:**(int)**:**(int)' specifies which is the location of the lower left corner of the string in regard of the lower left corner of its superview.

This class implements also a method, '**erase**' that enables the user to erase the string from the screen.

'**horFill:**(BOOL)' is a method taking a boolean as parameter. If this parameter is set to TRUE, the background color of the RMGString extends to the right and left edges of its view. The default situation is that the particular background color is only set where the characters are.

Note that as the majority of these methods do not redraw the screen after having been used, it is necessary to use either the '**redraw**', '**show**' or '**update**' method. The latter being a method which purpose is to erase and redraw the string on the screen.

As example of RMGString we can extend example 7.3 :

```
+ newIn: anEnvironment
    {
    id aView, bView, aString;                                /* 1*/
    ...
    self = [RMGView     origin: 100: 100                     /* 2*/
                        extent: 200: 200
                        superview: anEnvironment
                        bkgd: 1];
    aView = [RMGView relative: 50: 25                        /* 3 */
                        extent: 100: 100
```

```
                              superview: self
                              bkgd: 2];
bView = [RMGView Type: 4                              /* 4 */
                              extent: 50
                              superview: aView
                              bkgd: 0];
aString = [RMGString origin: 325: 100                 /* 5 */
                              extent: 200: 200
                              superview: anEnvironment
                              bkgd: 1];
[[[[[[[[aString string: "This is an RMGString"]       /* 6 */
                  font: [FontMngr cour12x20]]         /* 7 */
                  color: 0]                           /* 8 */
                  bkgd: 2]                            /* 9 */
                  center]                             /* 10 */
                  centerV]                            /* 11 */
                  horFill: TRUE]                      /* 12 */
                  show];                              /* 13 */

...
}
```

Line 1 is the declaration of three local id variables which will contain the addresses of the subviews we want to create and the address of the new RMGString instance.
Line 2 is the creation of the view we described in example 7.1. At line 3 we find the creation of a subview of our principal view. Line 4 is the creation of the sticky subview we place in the view identified by 'aView', as described in example 7.3

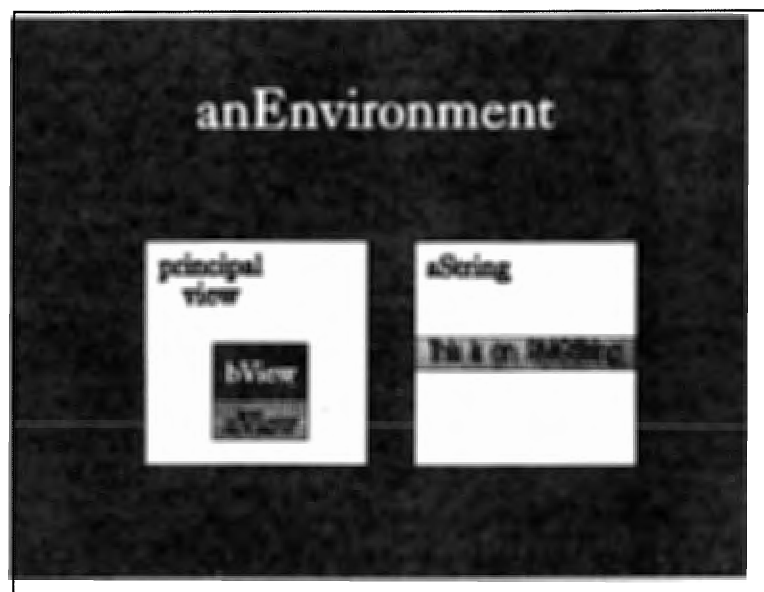


Figure 7.4 Creation of an RMGString

Line 5 creates the instance of RMGString using an RMGView factory method and thus creating a view of 200 by 200 pixels in the environment, at location (325,100). At line 6 we can see the specification of the string to be displayed: 'This is an RMGString'. It is displayed in a courrier font of 12 by 20 pixels -line 7-, in black -line 8- and with a red background color -line9. Line 10 and 11 center this string horizontally and vertically, while line 12 specifies that the background color as to be prolonged to the edges of its view. Finally line 13 makes all this appear on the screen (see figure 7.4). **(Example 7.5)**

7.4. RMGIcon

RMGIcon is a class enabling the user to display icons created with the application 'IconEditor' for example -described in chapter 4.

It is a class that displays a rectangular bit map at a certain place on the screen. This bit map has a predefined width, height and a position relative to its superview. This class is a subclass of RMGView, thus inheriting all its instance variables and various methods.

We first present the different interesting instance variables, then the most used factory methods and finally the different instance methods.

7.4.1. *Instance variables*

There are various instance variables in this class, containing some useful parameters concerning the icon. Among these we can find '**b_width**' and '**b_height**' which are both integers, designed to contain the dimensions of the icon. '**pos_x**' and '**pos_y**' are yet two other variables of type double, which are designed to contain the location of the icon in its superview.

Another interesting instance variable is '**clip**'. It is a boolean which, if set to TRUE, denotes the fact that the icon image is completely confined by its superview. Otherwise, it is displayed partially or totally outside its superview.

7.4.2. *Factory methods*

This class provides the user with about ten different factory methods. These methods are designed to cope with the creation of an RMGIcon instance containing an icon. This icon can use the same data file format as in RMG or any other. The instances are designed to cope directly with files or with an 'IconModel' instance. IconModel[9] is a class enabling the user to build a model of an icon without displaying it; this model stores the minimum attributes of the IconModel instance.

'**makeFromModel:**(id) **in:**(id)' is a method that creates an RMGIcon instance from an IconModel instance specifying also its superview as second parameter.

---

[9] : Examples of use of RMGIcon and IconModel are given in a following chapter

Another factory method of RMGIcon is '**makeIconFrom:**(char*) **in:**(id)' which creates an icon from a data file -passed as first argument- and displays it in the specified view -passed as second argument. In addition to this the user has to specify the location where the icon has to be displayed, combined with the 'show' instance method in order to make it appear on the screen. For example :

```
id myIcon;                                     /* 1 */
...

myIcon = [RMGIcon makeIconFrom: "./myIconFile.icon"
                        in: aView];            /* 2 */
[[myIcon dataOrigin: 10: 10]
        show];                                 /* 3 */
```

Line 1 defines the variable of type id which will contain the address of the RMGIcon instance.
Line 2 is the instantiation of the RMGIcon class with './myIconFile.icon' as file name and 'aView' as superview, assuming that 'aView' is a variable containing the address of an already existing view.
Finally, line 3 specifies the location where the icon has to be displayed in its superview -(10,10)- and requests the RMGIcon instance to be displayed -through the 'show' method. (**Example 7.6**)

7.4.3. *Instance methods*

Few instance methods are provided in this class, among which one can find a certain number of them that manages the obliteration, showing, etc, of an icon.

For example, '**show**' and '**erase**' respectively displays and erases an icon on and from the screen.

'**overlay**' is a method enabling the user to display on the screen only the non-zero data bytes of the bit map. The result is that whatever is under the icon is seen where the bytes of the bit map are set to zero.

'**clip:**(BOOL)' is an instance method enabling the user to set the clip state of an icon within its superview. This method takes a boolean as argument. This means that if the passed argument is TRUE, the icon will be clipped to its superview bounds.

Two methods enable to scroll the icon respectively horizontally and vertically : '**scrollx:**(int)' and '**scrolly:**(int)'. They both take as argument an integer which specifies the amount of pixels by which the icon has to be moved. '**dataOrigin:**(int):(int)' is the method to be used in order to specify the location -represented by the two integer arguments- of the icon in its superview. Note that this last method does not update the screen, so the user has to send the RMGIcon instance a message containing the '**update**' selector. The method '**width:**(int) **height:**(int)' changes the width and height of the icon; the two arguments are of integer type.

7.5. Summary

We presented in this chapter four basic classes shipped with RMG:

- RMGView;
- Envir;
- RMGString;
- RMGIcon.

We described some of their basic characteristics in order to accustom the user with this type of class. We saw that RMGView was the foundation of most RMG classes and that its purpose was the management of the overall screen. We saw also that the windows resulting of its instantiation were organized in a superview-subview hierarchy.

For Envir, we saw that it was the background for each new application being created in the environment and that its goal was to provide the user with all the means to control the state of the mouse.

We talked about RMGString and said that it was a class designed to display strings within views, providing at the same time every possible means to manipulate these strings.

Last we said that RMGIcon was a class enabling the user to display icons under the shape of bit maps.

Chapter 8: New notions about programming in RMG

In this chapter we introduce some new notions about programming in RMG. These new notions will help the user design some actions for an application (8.1) or build a menu (8.2). It also shows the user how to cope correctly with the mouse, how to iconize an application's window and finally how to use an entirely new concept : the active collection (8.3).

8.1. The actions

We see in this part what is hidden behind the word 'action' in RMG and what it means in this environment.

We talked of actions earlier without explaining exactly what it was. We said that it was something that could **change the current state of the application** and that was performed when the user clicked on an icon, on a button or selected a menu item in a menu. The 'something' that can change the current state of the application is in fact :

- a **C function** defined by the user;
- a **method** defined by the user;
- a method contained within an RMG class;
- an already defined **RMG action**.

C functions are defined using the standard C syntax. This is usually done between the part declaring the instance variables and the part defining the different methods. Note that this order has no particular importance, it is just respected in order to keep a certain clarity in the classes.
User methods are defined as explained above. While methods defined in RMG classes are already present in these classes.
Along with this, RMG provides already defined actions, as for example :

- 'moveViewAction' which moves a view in the environment;
- 'quitProgAction' which stops the execution of an application;
- etc.

These actions can either be linked to instances of the 'RMGView' class or to instances of subclasses of RMGView. They can also be linked to menu items, this is analyzed in point 8.2.

8.1.1. *C functions*

C functions must be declared somewhere as the code to be executed when a particular icon or button is clicked on. This process encompasses several steps. Note that these functions are always defined as '**static**' functions in order to

suppress possible name conflicts between the functions defined in the class and other functions included through header files.

Let us first see which are the different steps involved in this process. We will then explain why it is so.

The first step in the process is to objectify this function using the class '**RMGAction**' and the '**newFuncAction:**(<function name>)' factory method within this class. This is done once and for all in the 'newIn:' method. The following lines give an example of C function 'objectivation' :

```
...
static id <variable name> = NULL;        /* 1 */
...
if (!<variable name>)                    /* 2 */
    <variable name> = [RMGAction
                          newFuncAction:
                         <name of C function>];
                                         /* 3 */
...
```

Line 1 is the definition and the initialization to NULL of the variable which will receive the address of the new RMGAction's instance; this declaration is done as a global variable definition, so before the '@requires' clause.
Line 2 checks if the variable has already a value different from NULL, if this is the case line 3 is skipped. If not, at line 3, the variable takes the value of the RMGAction instance's address. If we take an example, we could have :

```
...
static id myFunctionAction = NULL;
...
if (!myFunctionAction)
    myFunctionAction = [RMGAction
                            newFuncAction:
                                 myFunction];
...
```

Assuming that 'myFunction' is a C function previously defined. (**Example 8.1**)

The second step is to store the address of the objectified C function in an instance variable inherited from RMGView: '**myAction**'. This is done using the instance method '**idAction:**(id)', which is also inherited from RMGView, at the instantiation of the view or the icon. It is of type :

```
...
[...[RMGView origin: ...] ...
                   idAction: <variable name>];
...
```

-in the case of a view-

...

```
[...[RMGIcon makeIconFrom: ...] ...
                    idAction: <variable name>];
...
```

-in the case of an icon-

If we extend example 8.1, we have :

```
...
static id myFunctionAction = NULL;
...
if (!myFunctionAction)
    myFunctionAction = [RMGAction
                            newFuncAction:
                                myFunction];
...
[...[RMGView origin: ...] ...
                    idAction: myFunctionAction];
...
```

-in the case of a view-

```
...
static id myFunctionAction = NULL;
...
if (!myFunctionAction)
    myFunctionAction = [RMGAction
                            newFuncAction:
                                myFunction];
...
[...[RMGIcon makeIconFrom: ...] ...
                    idAction: myFunctionAction];
...
```

-in the case of an icon- (**Example 8.2**)

The explanation of all this is contained within the mechanisms of RMG themselves and in one of RMG conventions : the left-hand side button of the mouse is for clicking on objects. When the user depresses the left-hand side button of the mouse, a message containing the selector '**leftButtonDown**'[1] is sent to the environment, which is an instance of the 'Envir' class.
The result is that the environment tries to find the top view under the RMG cursor and sends it a message containing the '**action**'[2] selector.
The view receives the message and performs the 'action' method. The result is the sending of a message containing the '**performWith:**(id)'[3] selector to the RMGAction instance which address is contained into the 'myAction' instance variable of the view that has been clicked on; while the argument passed to this method is the address of the view itself.

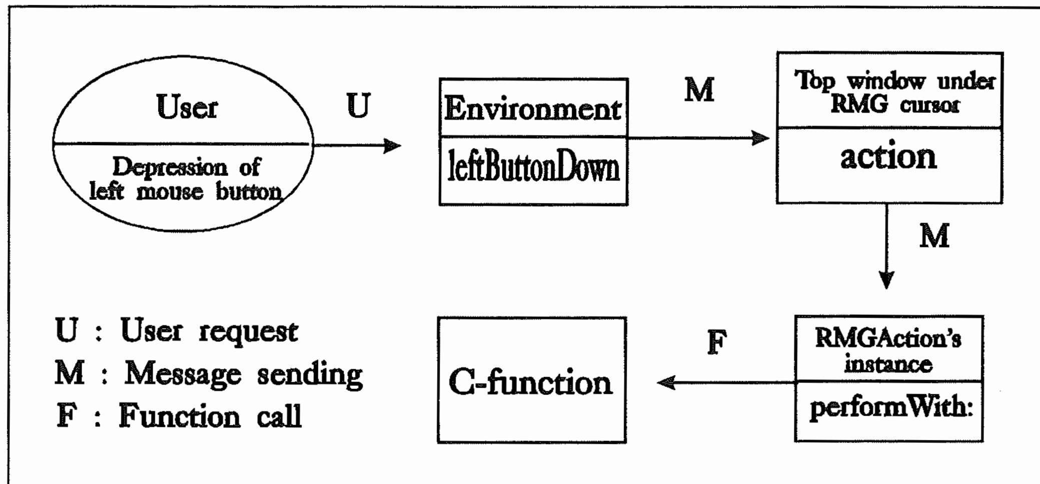
---

[1] : This is a method implemented in the Envir class

[2] : This is a method implemented in the RMGView0 class

[3] : This is a method implemented in the RMGAction class

This method, when executed, calls the installed C function, causing its execution. This C function takes as parameter, the address passed to the 'performWith:' method as argument. The process is shown at figure 8.1 (inspired of [WEGENER, 90] p.42).



<u>Figure 8.1</u> Reaction to the depression of the left mouse button in case of an action represented by a C function

A problem appears with these C functions : the arguments. The principle of objectified C functions in RMG is that if an argument is defined within the declaration of the function, it can not be anything else than the address of the view to which is linked the function; so we have :

```
static myFunction(aView) id aView
    {
    ...
    }
```

where 'aView' will be the variable containing the view's address to which the function is linked.

The C functions linked to views pose another problem. It is impossible to access directly an instance variable from inside a C function, other than the one passed as argument.
For example, if 'aView' is an instance variable containing the address of a subview of an application's window and that 'aView' is passed as argument to the C function 'myFunction', it is impossible for the user to access directly, inside 'myFunction', the address of the application's window contained in the 'self' variable.
To solve this problem, the user has two possibilities. The first one is to store 'self' in aView's viewIcon instance variable and reference it by issuing the message :

```
[aView viewIcon];
```

or by using the C syntax :

```
aView->viewIcon
```

The second solution is to use the subview-superview hierarchy, by using the 'superview message'. In our case :

```
[aView superview];
```

8.1.2. *Methods*

The methods must, as the functions, be declared as the code to be executed when a particular icon or button is clicked on.

The first step is to create an instance of a class '**RMGAction1**' using its '**sel:**(char*) **rec:**(id)' factory method. This is an instance which keeps track of the receiver and of the selector contained in the message to be sent to this receiver, when the instance receives the message containing the selector 'performWith:'; these two parameters are stored into two instance variables, namely '**selector1**' and '**receiver1**' . As for the functions this need only be done once, therefore it is usually done in the 'newIn:' method. It takes the shape :

```
...
static id <variable name> = NULL;        /* 1 */
...
if (!<variable name>)                    /* 2 */
    <variable name> = [RMGAction1 sel: "<selector
                name>" rec: <receiver's
                        address>];    /* 3 */
...
```

Line 1 is the definition and the initialization to NULL of the variable which will receive the address of the new RMGAction1's instance; this declaration is done as a global variable definition.
Line 2 checks if the variable has already a value different from NULL; if this is the case line 3 is skipped. If not, at line 3, the variable takes the value of the RMGAction1 instance's address.
For example :

```
...
static id myMethodAction = NULL;
...
if (!myMethodAction)
    myMethodAction = [RMGAction1 sel:
                                myMethod
                        rec:
                                aView];
...
```

In this case we assume that 'aView' is the instance of the class that implements the method 'myMethod'. (**Example 8.3**)

The second step is the same than for the C functions, thus the storage of the RMGAction1 instance's address into the 'myAction' instance variable -see point 8.1.1.

The explanation of the mechanism is the same than for the functions except that the RMGAction1 instance sends a message containing the stored selector - thus stored into 'selector1'- to the instance which address is stored into 'receiver1' (see figure 8.2 inspired of of [WEGENER, 90] p.42).

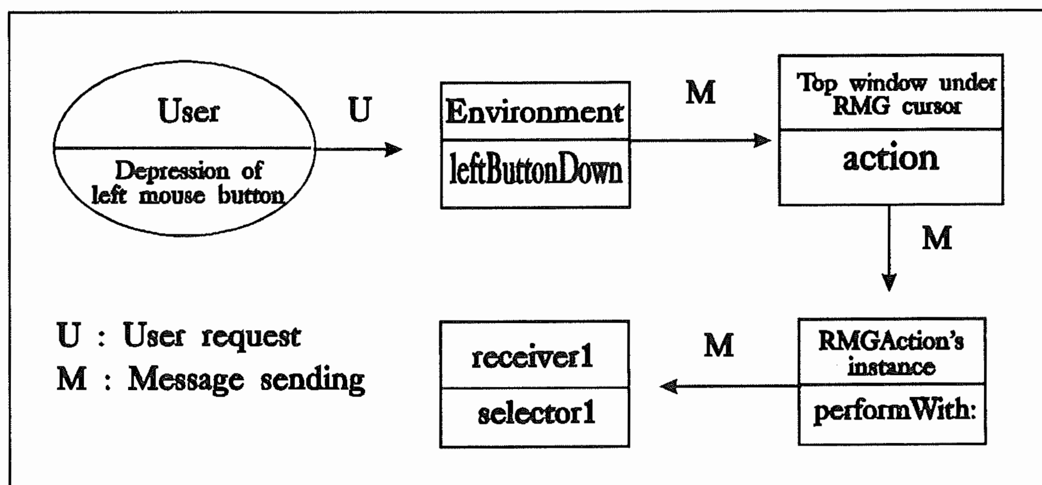


Figure 8.2 Reaction to the depression of the left mouse button in case of an action represented by a method

8.1.3. *RMG actions*

For RMG actions, the process is simpler. One has just to specify the name of the action, using the 'idAction' method. For example :

```
...
[...[RMGView origin: ...] ...
                idAction: quitProgAction];
...
```

-in the case of a view-

```
...
[...[RMGIcon makeIconFrom: ...] ...
                idAction: quitProgAction];
...
```

-in the case of an icon-

8.2. The menus

As it was often said, menus are one of the basic communication means between the user and the application, in the RMG environment. The principle of a menu is to enable the user to select a menu item in order to change the current state of the application; this is done by 'making the application execute' an action. This implies in our case that the menu has to send messages to the application, in order to reply to the menu item selection made by the user. These menus are present in almost every application under different shapes :

- **pop-up menus**;
- **fix-menus**.

The first ones are the traditional ones for which RMG provides some classes in order to implement them. In short, they are menus appearing at depression of the right-hand side button of the mouse by the user. They can contain as many menu items as necessary; if these items do not all appear at once in the menu, a simple mouse motion -up or down- enables the menu to scroll, hence the hidden menu items to be displayed. These menu items can hide submenus, appearing when the user drags the mouse rightward. Menu items of these submenus can also hide other submenus, thus enabling the user to combine as many submenus as desired or needed.
The second type was designed by COLOS members at Kiel's University[4], namely Uwe Heimburger and Detlev Wegener. This type of menu (see figure 8.3) is an independent window which has as superview the environment itself and which is linked to an application's window. These menus contain some subviews which when clicked on activate a particular action or change a simulation's parameter value. Note that multiple fix-menus can be linked to a particular application. These menus can be created using the class '**FixMenu**'.

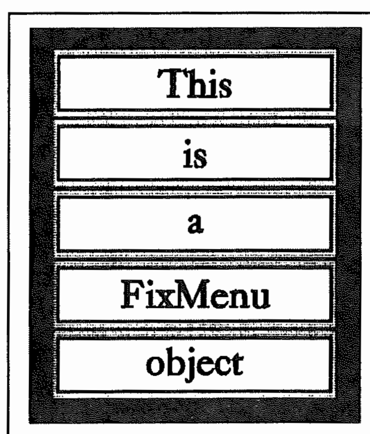


Figure 8.3 Example of fix-menu

[4] : Precisely at the Institut für die Pädagogik der Naturwissenschaften an der Christian-Albrechts Universität zu Kiel, under the supervision of Dr. Hermann Härtel

We only describe the creation of the first sort of menus, as it is the principal one in RMG[5].

8.2.1. *The specification of a menu*

The specification of a menu for an application consists in the **building of a class** -the menu class- which is part of the set of classes composing this application. So an application's menu is the instantiation of a class and is so an object itself. As we saw in chapter 6, this class is instantiated once at the instantiation of the master-class -in the 'delayInit' factory method.

As for each class defined in RMG, some particular items are required in a menu class definition. We will start by expressing one convention usually adopted by all RMG programmers : the name of the menu class is the name of the application -and so of its master-class- preceding 'MTree'. For example the menu class of an application 'MyApp' is 'MyAppMTree'.

A menu class description begins like each other class description with the header files. In this case the three traditional **header files** -'objc.h', 'rmg.h' and 'envir.h'- are still compulsory. Then follows the '**@requires**' clause listing all the classes used in the menu class, except the superclass.

Then comes the declaration of global variables with an obligatory one:

```
static id onlyInstance;
```

This is the declaration of the variable which will receive the address of the only instance of the menu class.

This is followed by the definition of the class name, of its superclass and of the message groups:

```
= <application name>MTree: EnvMTree (<user message group>,
                                      working, RMGVW,
                                      Collection, Primitive)
```

As an application's master-class had to be subclass of 'Envir', the class defining the application's menu must be subclass of '**EnvMTree**' which is the class defining the main menu associated to the 'Envir' class.

After this are defined the instance variables and the various methods. Among these methods, one must include two factory methods called : '**delayInit**' and '**getIt**'. 'delayInit' initializes various items needed by the menu and is the place where the menu is constructed, where the actions for each menu items are defined. We give here a short template of this method, which is further detailed

[5] : If the reader is interested in the second sort of menus, we advise the reading of [HEIMBURGER, 90] pages 39 to 52

later :

```
+ delayInit
    {
    static BOOL beenHere = FALSE;                    /* 1 */

    if (!beenHere)                                   /* 2 */
        {
        beenHere = TRUE;

        self = onlyInstance = [self new];            /* 3 */

        rootMenu = [<creation of the root menu>];  /* 4 */

        [...[rootMenu <creation of a menu items>]...];/* 5 */
        ...
        [...[rootMenu <creation of a menu items>]...];
        }
    return <application name>MTree;
    }
```

Line 1 declares a local variable that will be used to check that the menu has been created once and only once; this is done at line 2.
Line 3 creates the instance of the menu, where 'self' and 'onlyInstance' both receive the address of the newly created menu.
Line 4 is the construction of the root menu where the variable 'rootMenu' is an instance variable inherited from 'EnvMTree'.
From Line 5 and downwards, the menu items are defined. We see in point 8.2.2. which are the classes one can use in order to construct the menu, define the menu items and thus replace '<creation of the root menu>' and '<creation of a menu items>' in the template above.

'getIt' is a factory method which goal is to return the variable 'onlyInstance' which contains the address of the newly created menu. It is of type:

```
+ getIt
    {
    return onlyInstance;
    }
```

The class description is of course terminated by the '=:' sign.

Once the class as been defined, one must not forget to include it in the makefiles and 'mainClass.m' file as explained in chapter 6.

After this, the more important stays to do. One has to link the menu to the application itself. This is done in the master-class' 'delayInit' and 'extraNewIn' methods -see chapter 6-, where one can find :

```
+ delayInit
  {
  static BOOL beenhere = FALSE;

  if (!beenHere)
        {
        if (self == <class name>)
              {
              [<menu class> delayInit];
              beenHere = TRUE;
              }
              else
              [<class name> delayInit];
        }
  return self;
  }
```

In this case, the only instantiation of the menu is asked with : '[<menu class> delayInit];'.
'extraNewIn' is of type :

```
- extraNewIn: anEnvironment
  {
  [self showAll];
  menuTree = [<menu class> getIt];
  ...
  return self;
  }
```

where the address of the menu's instance is returned with '[<menu class> getIt]' and stored in the instance variable 'menuTree' inherited from 'Envir'.

We said earlier that an application could have only one menu. This is entirely true but must be clarified a little bit more by saying that a **view** can have only one menu. As the application's window is a view, it can dispose of one menu for itself; this menu is called the application's menu. But as the application's window contains probably one or many other subviews which are independent instances of a class, these subviews can have also their own menus.

We give in table 8.1 a summary of what has been said of a menu class.

Header Files	#include "objc.h" #include "rmg.h" #include "envir.h" *[*#include "<other header file name>"*]***
C variables declaration	*Declaration of global variables*
Declaration of classes used	@requires <class name> *[*,<class name>*]***
Class name declaration	= <class name>: <superclass name> (<personal message group>, RMGVW, Primitive, Collection, *[*,<other message groups>*]***)

Instance variables declaration	{
	Declaration of instance variables
	}

Definition of C functions	*Definition of functions*
Methods definition	+ delayInit {...} + getIt {...}
End of class description	=:

Table 8.1 What has to be included in a menu class description

8.2.2. *Which class can be used ?*

We can find two interesting classes to be used to create a menu :

- **RMGMenu**;
- **RMGMenu1**.

The first one, 'RMGMenu', is a subclass of RMGString, which instance is composed of RMGStrings of same dimensions, one on top of the other and surrounded by a border. It represents a one level menu composed of a collection of menu items. We saw examples of these menus when describing applications in chapter 4.

It has a certain number of instance variables among which the most interesting are :

- '**item_number**';
- '**receiver**'.

The first one is an integer variable designed to store the number of menu items that composes the menu. The second one is an id variable which is designed to store the address of the receiver. This receiver is the instance of the class to which the menu is linked.

RMGMenu provides also a few factory methods, among which '**char_wide:**(int) **items:**(int) **font:**(id)' is the most often used. This method creates an instance of RMGMenu which has a certain width, measured in characters, a certain number of menu items and which is displayed in a particular font.

For example :

```
...
rootMenu = [RMGMenu char_wide: 10
                    items: 3
                    font: [FontMngr
                           cour12x20]];
...
```

creates a menu of 3 menu items, this menu is of 10 characters wide and the menu items are displayed in the 'cour12x20' font. The address of the new instance is stored into 'rootMenu' as explained in point 8.1.1. (**Example 8.4**)

Once the menu has been instantiated, one must also create each menu item using instance methods. The most 'popular' one is '**at:**(int) **putStr:**(char*)' which enables the user to place a string at a particular place in the new instance of RMGMenu.

For example, if we extend example 8.4, we can have :

```
...
rootMenu = [RMGMenu char_wide: 10
                    items: 3
                    font: [FontMngr cour12x20]];
[rootMenu at: 0
          putStr: "!! QUIT !!"];
[rootMenu at: 1
          putStr: "Start"];
[rootMenu at: 2
          putStr: "Stop"];
```

(**<u>Example 8.5</u>**)

There are many other instance methods enabling the user to access the instance variables defined in this class. And it is of course possible to utilize every RMGString methods, RMGMenu being one of its subclasses, along with all the other methods present in RMGMenu.

'RMGMenu1' is a subclass of RMGMenu. It is a class enabling the building of systems of hierarchical menus. It enables the creation of menus having menu items hiding a submenu at its right, as seen in a preceding chapter. Thus the great advantage of RMGMenu1 on RMGMenu is that it enables the preview of submenus when the RMG cursor is in the right 4/5 part of the menu item.

Its use is pretty much the same as that of RMGMenu and all the methods implemented in this last class can be of course used from RMGMenu1. The only difference is the presence of submenus whose addresses are stored in the viewIcon instance variable of the menu item -as it is an RMGString. This is possible, an RMGMenu instance being a sort of array, each slot containing the informations concerning one menu item -thus the informations concerning one RMGString instance. Note that the 'viewIcon' variable is aliased '**subMENU**'; this is also the case for the method enabling to access its value, 'viewIcon', which is aliased '**subMENU**' or the method enabling to change its value, 'viewIcon:', which is aliased '**subMENU:**(id)'.
For example we can transform example 8.5 to suppress the two menu items 'Start' and 'Stop' from the main menu and transport them to a submenu hidden at the right of a menu item 'Action >' :

```
...
aSubMenu = [RMGMenu1 char_wide: 10    /* 1 */
                     items: 2
                     font: [FontMngr
                            cour12x20]];
[aSubMenu at: 0                        /* 2 */
          putStr: "Start"];
[aSubMenu at: 1                        /* 3 */
          putStr: "Stop"];
```

```
rootMenu = [RMGMenu1 char_wide: 10    /* 4 */
                     items: 2
                     font: [FontMngr
                              cour12x20]];
[rootMenu at: 0                       /* 5 */
          putStr: "!! QUIT !!"];
[[rootMenu at: 1                      /* 6 */
           putStr: "Action >"]
         subMENU: aSubMenu];
...
```

where 'aSubMenu' is a newly defined instance variable of type id which will contain the address of the new submenu.
Line 1 defines the new submenu using the same 'principles' as for RMGMenu; while lines 2 and 3 define the menu items of this submenu.
Line 4 defines the main menu with only two menu items -...'items: 2'...- and as an instance of RMGMenu1.
Lines 5 and 6 define the two menu items of the main menu, with at line 6 the specification of the submenu of the 'Action >' menu item. When the user 'calls' this submenu on the screen, the menu will appear as shown at figure 8.4.

(**Example 8.6**)

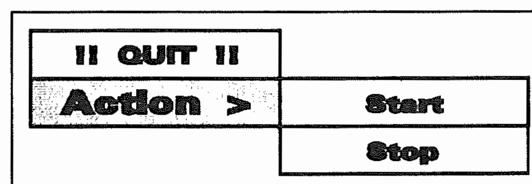


Figure 8.4 Main menu with a submenu hidden at the right of the 'Action >' menu item

8.2.3. *Actions in a menu*

In this part we talk of the actions that have to be linked to each menu item.

Each menu item is usually linked to a particular action; 'usually' because some menu items hiding a submenu do not obligatorily need to have an action linked to them. Now we need to know how to define these actions for a menu item.

As for the actions defined in point 8.1, actions linked to menu items can be :

- C functions;
- methods;
- RMG actions.

We are going to describe them one after the other.

The C functions are described in the menu class itself and must be also objectified. This is done as explained above but in our case in the menu class 'delayInit' factory method :

```
...
static id <variable name> = NULL;
...
+ delayInit
    {
    ...
    if (!<variable name>)
            <variable name> = [RMGAction
                                    newFuncAction:
                                            <name of C
                                                    function>];
    ...
    }
...
```

As we can see the process is exactly the same as for C functions representing actions linked to views or icons.
For example :

```
...
static id myFunctionAction = NULL;
...
+ delayInit
 {
 ...
 if (!myFunctionAction)
         myFunctionAction = [RMGAction
                                 newFuncAction:
                                         myFunction];
 ...
 }
...
```

(Example 8.7)

The second step concerns the linkage of these objectified C functions to menu items. This is also done through the 'idAction:' method; indeed, we must remember that RMGMenu is a subclass of RMGString which is subclass of RMGView. If we merge example 8.6 and example 8.7, it looks like :

```
static id myFunctionAction = NULL;
...
+ delayInit
 {
 ...
 if (!myFunctionAction)
     myFunctionAction = [RMGAction
                             newFuncAction:
```

```
                                        myFunction];

    ...
    aSubMenu = [RMGMenu1 char_wide: 10
                                items: 2
                                font: [FontMngr
                                        cour12x20]];
    [[aSubMenu at: 0
                putStr: "Start"]
                idAction: myFunctionAction];
    [aSubMenu at: 1
                putStr: "Stop"];

    rootMenu = [RMGMenu1 char_wide: 10
                                items: 2
                                font: [FontMngr
                                        cour12x20]];
    [rootMenu at: 0
                putStr: "!! QUIT !!"];
    [[rootMenu at: 1
                putStr: "Action >"]
                subMENU: aSubMenu];
    ...
}
```

where the action represented by 'myFunctionAction' has been linked to the menu item 'Start' in the submenu 'aSubMenu'. So when the user selects this menu item, the C function 'myFunction' is executed. (**Example 8.8**)

For methods, the process is simpler. They are defined inside the class to which the menu class is linked. In order to link these methods to particular menu items, a class called '**MTreeAct**' is used. If the menu item is selected, the instance of 'MTreeAct' sends a message containing the desired selector to the desired receiver. The desired selector is specified using MTreeAct's '**sel:**(char*)' factory method.
It is of type :

```
...
[...[<menu name> at: ...]...
            idAction: [MTreeAct: "<method
                                    name>"]]
            ...];
...
```

where '<method name>' is the name of the method one wants to be executed when the menu item is selected. If we extend example 8.8, we have :

```
static id myFunctionAction = NULL;
...
+ delayInit
```

```
{
...
if (!myFunctionAction)
    myFunctionAction = [RMGAction newFuncAction:
                                    myFunction];

...
aSubMenu = [[RMGMenu1 char_wide: 10
                      items: 2
                      font: [FontMngr
                                cour12x20]]
                      APPL: self];
[[aSubMenu at: 0
          putStr: "Start"]
          idAction: myFunctionAction];
[[aSubMenu at: 1
          putStr: "Stop"]
          idAction: [MTreeAct sel:
                                "myMethod"]];

rootMenu = [RMGMenu1 char_wide: 10
                      items: 2
                      font: [FontMngr
                                cour12x20]];
[rootMenu at: 0
          putStr: "!! QUIT !!"];
[[rootMenu at: 1
          putStr: "Action >"]
          subMENU: aSubMenu];
...
}
```

In this case, the reader notices that something has been added to the definition of the submenu 'aSubMenu', for instance '**APPL: self**'. This is done in order to set up a link between the menu and its application so that the receiver of the message defined by 'idAction: [MTreeAct sel: "myMethod"]' is correctly known; this is required by RMG. (**Example 8.9**)
Note that other classes are available to the user in order to link methods to menu items. For example '**MTreeActOne**' enables the user, through the '**sel:** (char*)**:** (char*)' factory method, to specify the method to be executed so as an argument that has to be passed to this method[6].

RMG actions are simpler to link to menu items. This is done by simply using the 'idAction:' instance method followed by the name of the action.
For example, if we extend example 8.9, we can have :

```
static id myFunctionAction = NULL;
...
```

[6] : For further informations on MTreeActOne, the reader will consult [HP, 89] p. MTreeActOne

```
+ delayInit
 {
 ...
 if (!myFunctionAction)
     myFunctionAction = [RMGAction newFuncAction:
                                 myFunction];

 ...
 aSubMenu = [[RMGMenu1 char_wide: 10
                       items: 2
                       font: [FontMngr
                                 cour12x20]]
                       APPL: self];
 [[aSubMenu at: 0
          putStr: "Start"]
          idAction: myFunctionAction];
 [[aSubMenu at: 1
          putStr: "Stop"]
          idAction: [MTreeAct sel: "myMethod"]];

 rootMenu = [RMGMenu1 char_wide: 10
                       items: 2
                       font: [FontMngr
                                 cour12x20]];
 [[rootMenu at: 0
          putStr: "!! QUIT !!"]
          idAction: quitProgAction];
 [[rootMenu at: 1
          putStr: "Action >"]
          subMENU: aSubMenu];
 ...
}
```

(**Example 8.10**)

Note that there are particular things one must include in an application's main menu under the shape of menu items. These things depend highly on the application itself and on the wish of the programmer.

One menu item that is almost compulsory is the one enabling the user to quit the application. In this case there are two possibilities :

- use the 'quitProgAction';
- define a new method.

The first possibility is the simplest and has been illustrated in the preceding point. The second one is the solution the programmer chooses if he wants to do additional things while quitting the application; for example, display something special on the screen. Anyway in this case the programmer has to send the master-class a message containing the '**quit_app**' selector which is an instance method implemented in Envir.

The other menu items present in a menu are entirely the choice of the programmer. He can design his own menu items, creating new C functions or methods.

Many classes in RMG are designed to implement menus for particular purposes. For example '**FileMTree**' is a class implementing a menu for applications using files, it provides menu items that enable to access a file browser, the possibility to load and save files, etc. This class is a subclass of 'EnvMTree' and can be used as superclass to design an application's menu class.

8.3. Additional features

We see in this point three additional features of RMG which are accessible to the programmer and can be of a certain use in some cases. The first one concerns the mouse. The second one concerns the possibility to iconize an application. Finally, the last one concerns the active collection.

8.3.1. *The mouse*

This is a short tip concerning the mouse and a particular class: '**Mouse**'.

The 'Mouse' class is a driver for the HP mouse, which implements only factory methods. Among these methods two of them can be of particular interest if one wants to get the exact location of the mouse on the screen -so the exact location of the RMG cursor- and the state of the mouse buttons. These two methods are : '**change:**(int*)**:**(int*)' and '**getButtons**'.

'change::' is a method enabling the user to store the x and y coordinates of the RMG cursor into two integer variables. Note that the two integer arguments which are passed to this method are pointers so that the instance can directly write the values into them. So we could have :

```
int x,y;
...
[Mouse change: &x: &y];
...
```

or

```
int *x,*y;
...
[Mouse change: x: y];
...
```

These two versions are different only by the fact that the two variables are defined differently 'playing' on the C syntax. **(Example 8.11)**

The second method, 'getButtons', returns the last button state of the mouse. This method must be absolutely used in combination with 'change::' as the mouse

state returned will correspond to the last inquiry about mouse position changes. So if one wants to have the current state of the mouse buttons, he has to send the message :

```
mouseState = [[Mouse   change: &x: &y]
                       getButtons];
```

where 'mouseState', 'x' and 'y' are three integer variables. The final return value of this message is an integer between 0 and 3 stored into 'mouseState', where :

0 = no buttons down
1 = left button down
2 = right button down
3 = both buttons down

8.3.2. *Iconizing an application*

Being able to iconize[7] an application appears to be very useful in some cases; mainly if one uses many applications at the same time, it enables him to 'suppress' one of them temporarily.

The process of enabling an application to be iconized is separated into two parts. The first part concerns the creation of a ghost object. A ghost object is an instance of the class '**Ghost**', which is the representation of the application on the screen when it is iconized; this representation is made of a small red rectangle with a string inside it usually representing the name of the application[8]. This instance enables the application to reappear when the icon is clicked on. Note that this class is a subclass of RMGView so inheriting of all its features. Note also that the instantiation of this class is usually done in the 'extraNewIn:' method and uses the '**stringGhost:**(char*) **In:**(id)' 'Ghost' factory method. It has the shape :

```
viewIcon = [[Ghost stringGhost: "<a string>" In: <a view name>]
                                                  viewIcon: self];
```

where '<a string>' is the string the programmer wants to see in the icon and '<a view name>' is the variable containing the address of the superview in which the programmer wants to see the icon appear. Usually the superview specified is the environment itself[9]. 'viewIcon: self' is there to specify which application must be

[7] : We recall that by 'iconizing', we mean shrinking the application's window to the dimensions of an icon, somewhere on the screen

[8] : 'Usually' because this string is specified by the programmer and so can be anything wanted

[9] : Recall that the environment's address is contained in the variable which is the argument of the 'extraNewIn:' method

iconized. The address of this Ghost instance is stored in the application's master-class viewIcon instance variable.
For example :

```
...
viewIcon = [[Ghost stringGhost: "myApplication"
                        In: anEnvironment]
                        viewIcon: self];
...                                             (Example 8.12)
```

The second step is to provide for the user a button or a menu item enabling him to launch the iconization. This has to be done by sending to the master-class a message containing the '**iconize**' selector which is a method implemented in the 'Envir' class. For example in the case of a menu, we could have :

```
...
[...[<menu name> at: <menu item number>
                putStr: "Iconize"] ...
                idAction: [MTreeAct sel: "iconize"];
```

where '<menu name>' is the name of the main menu or the name of a submenu of this main menu.
In the case of a button we could have :

```
...
id iconizeAction = NULL;
...
if (!iconizeAction)
  iconizeAction = [RMGAction1 sel: "iconize" rec: self];
...
[...[RMGString origin: ...]
              string: "Iconize"]
              ...
              idAction: iconizeAction];
```

8.3.3. *The active collection*

The **active collection** is the solution to a very annoying problem. When the application enters a loop, the user can not use the mouse anymore. This is due to the fact that when the environment is started, it enters a quasi-infinite[10] loop designed to manage the mouse; that is :

- check the RMG cursor's location on the screen and update it following the mouse motions;
- check the state of the mouse buttons and react according to this state.

[10] : Quasi-infinite because the environment can be stopped whenever the '!! QUIT !!' menu item is selected in the application's main menu

Then if an application enters a loop itself, the mouse loop is short-circuited suppressing the use of the mouse for the user. This is not a great problem if the loop is finite and stops after a short number of iterations, but if it has to stop after the user selecting a menu item or clicking on a particular screen object, so using the mouse, nothing will stop at all.

The solution to this problem is to use the active collection. This collection is a list, defined as a global variable in the Envir class, which is designed to contain instance addresses. This collection is used by the environment to send the instances listed into it, a message containing a predefined selector: '**update**'. This method 'update' contains the operations that would have been coded inside the loop one tries to replace.

At the start of the environment, the only instance present in this list is the environment itself; the other instances listed depend on the needs of the applications and the desire of the programmer. They can be placed and retrieved from the list using the two instance methods implemented in the Envir class: '**installActive:**(id)' and '**deleteActive:**(id)', these two methods taking the address of the application as argument. One can deduce that an instance receives the 'update message' as long as it is present in the active collection.

For example, if one wants to implement a loop which is entered when the user clicks on a 'Start' button and which is quitted when the user clicks on a 'Stop' button, it takes the shape of :

```
...
static id startAction = NULL;                    /* 1 */
static id stopAction = NULL;                     /* 2 */
...
id startButton, stopButton;                      /* 3 */
...
static id startFunc(aView) id aView              /* 4 */
{
...
[[aView viewIcon] installActive: [aView viewIcon]];
                                                 /* 5 */
...
}
static id stopFunc(aView) id aView               /* 6 */
{
...
[[aView viewIcon] deleteActive: [aView viewIcon]];
                                                 /* 7 */
...
}
...
+ newIn: anEnvironment                           /* 8 */
 {
 ...
 if (!startAction)                               /* 9 */
  startAction = [RMGAction newFuncAction: startFunc];
```

```
  startButton = [[[[[[[[[RMGString relative: 50: 25
/                                                   * 10 */
                                 extent: 100: 100
                                 superview: self
                                 bkgd: 2]
                                 color: 0]
                                 font: [FontMngr sysfont]]
                                 string: "Start"]
                                 center]
                                 centerV]
                                 idAction: startAction]
                                 viewIcon: self]
                                 show];
  if (!stopAction)                                  /* 11 */
    stopAction = [RMGAction newFuncAction: stopFunc];
  stopButton = [[[[[[[[[RMGString relative: 175: 25
                                                    /* 12 */
                                 extent: 100: 100
                                 superview: self
                                 bkgd: 2]
                                 color: 0]
                                 font: [FontMngr sysfont]]
                                 string: "Stop"]
                                 center]
                                 centerV]
                                 idAction: stopAction]
                                 viewIcon: self]
                                 show];
...
}
...
- update                                            /* 13 */
  {
  loop body
  }
...
```

Line 1 and 2 define two <u>global id variables</u> initialized to NULL, which will contain the addresses of the RMGAction instances linked to each buttons -'Start' and 'Stop' buttons.
Line 3 is the declaration of two <u>instance id variables</u> which will receive the addresses of each buttons.
At line 4 we find the declaration of one of the C functions; this function is linked to the 'start' button -see line 9. This function contains -at line 5- the installation of the application in the active collection. Note that the address of the application is accessed through the 'viewIcon' instance variable of the 'Start' view, in which it is stored at creation -see line 10.
Line 6 is the declaration of the other C function, which is linked to the 'stop' button and which contains the application retrieval of the active collection at line 7.

Line 8 starts the definition of the 'newIn:' factory method, which contains at line 9 the objectivation of the 'startFunc' C function.

Line 10 creates the 'start' button which is an instance of the 'RMGString' class, in order to insert a string in the displayed view. So it creates a red view of 100 by 100 pixels, situated at relative coordinates (50,25) in the application's window. Within this view is displayed in black and 'sysfont' font the string 'Start', which is centered vertically and horizontally within its view. The action 'startAction' is then attached to this view, putting also the address of the environment into the 'viewIcon' instance variable.

Line 11 objectifies the 'stopFunc' C function and line 12 defines the 'stop' button. It is red, 100 by 100 pixels and situated at relative coordinates (175,25) in its superview -the application's window. It encloses a black 'Stop' string, displayed in 'sysfont' font, centered horizontally and vertically. The action 'stopAction' is then attached to this view, putting also the address of the environment into the viewIcon instance variable. Line 13 begins the definition of the 'update' instance method. (**Example 8.13**)

8.4. Summary

Our goal in this chapter was to introduce new notions about the programming in RMG.

The first part concerned the actions. We said that an action was the execution of :

- a **C function** defined by the user;
- a **method** defined by the user;
- a method contained within an RMG class;
- an already defined **RMG actions.**

that could change the current state of the application and that was launched by the user clicking on an icon, on a button or selecting a menu item in a menu.

Next we saw that these functions or methods had to be objectified using RMGAction or RMGAction1, in order to be linked to a view using Envir's 'idAction' instance method.

In the second part, we analyzed the creation of applications pop-up menus, which are the traditional menus and for which RMG provides some classes in order to implement them.

We saw that they had to be created through the building of a menu class which had to :

- be part of the set of classes composing the application;
- be subclass of the 'EnvMTree' class;
- contain two factory methods: 'delayInit' and 'getIt'.

Next, we described which were the classes one could use to create a menu and to define the menu items, namely 'RMGMenu' and 'RMGMenu1'. Last, we described how to implement actions to be linked to menu items, again using :

- C functions;
- methods;
- RMG actions.

Our last and third part concerned additional features including the mouse, the iconization of an application and the active collection.

Concerning the mouse, we saw that a class, 'Mouse', existed and could be used to get the exact location of the mouse on the screen -so the exact location of the RMG cursor- and the state of the mouse buttons.

Concerning the iconization of an application, we saw that this had to be done in two steps. The first one was the creation of an instance of the class '**Ghost**', which is the representation of the application on the screen when it is iconized; this representation is a small red rectangle with a string inside it usually representing the name of the application. The second step was to provide for the user a button or a menu item enabling him to launch the iconizing action through the 'iconize' instance method of the 'Envir' class.

Concerning the active collection, we saw that it was the solution to the 'loop problem', that is the fact that an application enters a loop that short-circuits the environment loop which checks and updates the RMG cursor's position and state according to mouse movements, the mouse thus becoming ineffective.

Chapter 9: The Video application

We are going to look at an application we created in RMG for the COLOS project: the Video application. This will enable us to see more illustrations of what has been said until now and to discover new things about RMG programming.

The first point is a general description of the application itself (9.1). It is followed by a description of the application as it is seen on the screen (9.2). The third point analyses the application in deep (9.3). Next comes a description of all the new RMG classes we used in this application (9.4). Last we list a certain number of problems we encountered during the building of this application (9.5).

9.1. General description of the application

The general description of the application concerns three points. The first point introduces the application itself and explains why it has been created. The second point describes the hardware used and the hardware requirements. The third point describes the functionalities of the application.

Note that along these points, we talk about modes, for example 'play mode' or 'still mode'. A mode characterizes the command the video player is currently executing. So for example if we talk about the 'play mode', the player is thus currently executing the play command which starts the reading of the disk.

9.1.1. *Introduction*

We created the Video application at the Christian-Albrechts University of Kiel for the COLOS project, under the supervision of Doctor Hermann Härtel. This application is designed to manage, from the RMG environment, a **laser video disc player** directly linked to the HP workstation on one side and to a monitor on the other side. The pictures on the disc are thus not displayed on the workstation's screen itself, RMG playing only the role of 'remote control' for the video player.

The underlying goal of this application is to enable teachers and learners to **use new techniques represented by new medias**[1], along with the power of the RMG environment and its simulations. For example, this application enables a learner or a teacher to use simulation applications while controlling the video player which displays on a monitor a film of the reality illustrating the current simulations.

9.1.2. *The hardware*

The hardware concerns on one side the HP workstation -HP 9000/360 model- and on the other side a laser video disc player -Pioneer 4200. This video player has a **special output port** enabling its connection with a computer's **serial port.**

[1] : Video players, video laser disc players, audio players, etc

The connection between these two entities -the workstation and the player- requires particular settings, mainly :

- 1200 or 4800 bauds;
- asynchronous;
- 8 bit characters;
- 1 start bit, 1 stop bit.

The communication between the two entities is done using **a system of command and reply**; the user sends a command or a request and he receives back a status or a reply from the player. For example, if the user sends the player the play command, he receives a status telling if the command has been correctly performed or not; now if the user sends a request to know which is the disc frame the player is currently on, he receives a reply containing the frame number.

This command-reply system is coded using the **ASCII code**. Each command which can be performed by the video player is coded with ASCII characters ended by the carriage return -'\r'- code. This is also the case when the player sends data or status to the workstation. For example, if one wants to eject the player's disc, he has to send the player the string: 'OPRJ\r' where 'OP' stands for OPening the drawer containing the disc and 'RJ' for ReJection of the drawer. The user receives back the 'R' character if everything is alright, an error code otherwise.

Note that the communication between the video and the workstation is done through a **device file** created in the Unix operating system environment.

9.1.3. *The application's functionalities*

The principal functionality of this application is of course the control of the video player. It is designed to control it 'basically', that is :

- starting the player in play mode;
- stopping the player;
- freezing the picture on the monitor -the still mode.

Further, it is possible to enter :

- a forward mode, which plays the disk forward at high speed;
- a backward mode, which plays the disk backwards at high speed.

The user is also able to select a portion[2] of the disk he wants to work on, by specifying the starting and ending frame number. It is possible for him to navigate within this portion using the functionalities exposed above or by using a scroll bar. This scroll bar indicates the frame the player is currently on and enables the user to go back and forth to a particular frame number.

[2] : We will call this portion: the **working area**

An editor window is available for the user if he wants to enter text or remarks about a working area. The content of this text window as well as all the parameters of the working area can be saved in a file for later use.

This is thus what the application currently does. We did not have enough time to implement further functionalities, as :

- the possibility to determine several working areas at the same time, the user being able to switch from one to another;
- the possibility to open more than one text window for one working area;
- the possibility to link directly this application to a simulation in order to launch the 'disk playing' automatically when a certain state is reached in this simulation.

The video player commands which have been used are :

- 'PL\r', which puts the video in play mode;
- 'RJ\r', which stops the video player;
- 'ST\r', which puts the video in still mode;
- 'PA\r', which pauses the video;
- '250SPMF\r', which enables the video to be put into fast forward mode;
- '250SPMR\r', which enables the video to be put into fast backward mode;
- '?F\r', which is the command used to send a request to the video asking the current frame number;
- '1250SEPL\r' -where '1250' stands for a frame number- which is the command used to make the video read the disk from a certain frame number, in this case frame number 1250.

Note that if several commands have to be sent one after the other, they must be sent one by one :

- send the first one;
- wait the reply of the video player -'R'-;
- send the following one;
- etc.

If it is not done this way, the video player is 'lost' and certain commands in the video player's buffer are erased by other ones.

9.2. The application on the screen

When one looks at the application on the screen[3], he sees a blue rectangular window containing several things (the user can find a screen copy of the application in the appendices).

The first thing is the orange title bar containing the 'Video Control' string and situated at the top of the window. This bar enables the user, when clicked on[4], to move the application's window across the screen.

The second very noticeable thing is the set of six buttons occupying the left side of the window. These are the buttons enabling the control and management of the video player. One sees :

- a '**Play**' button, enabling the user to start the video player in play mode;
- a '**Stop**' button, enabling the user to stop completely the player;
- a '**Still**' button, which freezes the current picture on the monitor, so making the video player enter the still mode;
- a '**Fwd**' button, which enables the user to play the disk forward at high speed;
- a '**Bwd**' button, which enables the user to play the disk backward at high speed;
- a '**Text**' button, enabling to make the text window appear on the screen.

The still mode, forward mode and backward mode can be stopped by clicking again on the button which makes the player enter this mode. Note that the forward and backward modes can be entered only from the play mode and that the buttons change color in order to indicate that they are activated. Note also that the text window can be erased temporarily from the screen by clicking again on the 'Text' button; in this case, the typed-in text is of course kept in the window.

At the right of the window, one can see the scroll bar which is automatically activated in play mode, moving according to the 'player motions' on the disk. In this scroll bar, one can spot several items :

- a graduation;
- two arrows;
- a grey vertical scroll bar between these arrows;
- a blue bar inside the grey scroll bar.

Each of these items has a particular purpose. The graduation represents a portion of the disk frame numbers and is updated following the current position of the player on the disk.

---

[3] : Note that the application is started by selecting the 'Video' menu item in the '*NEW* >' menu item of the environment's main menu.

[4] : When this is the case, the RMG cursor changes its shape

By clicking on the grey scroll bar, the user switches the player into pause mode; this stops the player on the current disk position and makes the picture disappears from the monitor[5]. The user can stop the pause mode by clicking again on the scroll bar.
While in pause mode, the user can utilize the two arrows situated respectively on the top and at the bottom of the scroll bar, to go forward or backwards on the disk. The picture is only updated when the pause mode is stopped.
The blue bar within the scroll bar indicates the currently red disk frame; it follows the player's motions on the disk and is so updated accordingly. When this bar is moving, the user is able to pick it up -by clicking on it and holding the button down- and by dragging the mouse up or down the user is able to make the player follow the mouse motions. During this operation, the player is set automatically in still mode and reenters the previous mode at the new disk location when the button is released. Note that this feature can be used while in play or pause mode.

We look now at the application's main menu (see figure 9.1).
Two of the menu items are familiar: '**!! Quit !!**' and '**Icon**' which respectively quits and iconizes the application. The '**Change Buttons**' menu item enables the user to change the buttons strings to icons. For example, instead of having the 'Play' string in the play button (see figure 9.2 a), one has a button with an icon representing the play command (see figure 9.2 b).

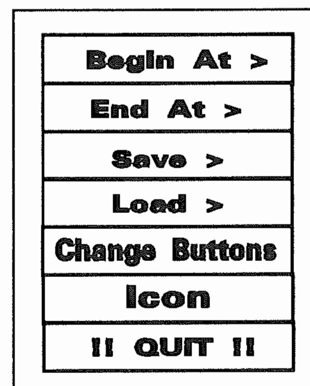


Figure 9.1 The Video application menu

The two following menu items, '**Load >**' and '**Save >**', enable the user to save or load files containing working areas parameters. The principle of these two menu items is that the user has to select them and reveal their submenu which is in fact a text box where the user is able to type-in the file names.

[5] : Note that this mode, as for the buttons, is indicated by the scroll bar changing color

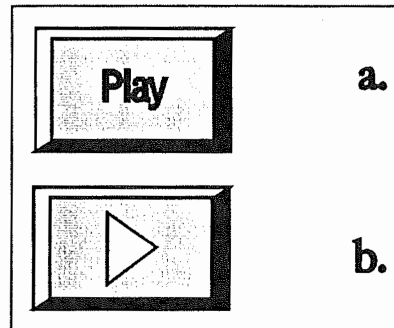


Figure 9.2 The effects of the 'Change Buttons' menu item

The two last menu items enable the user to specify the working area he wants to work on; '**Begin At >**' specifies the beginning of the area and '**End At >**' specifies its end. This is also done via a box hidden at the right of each of these menu items, into which the user has to enter the numbers using the keyboard. Note about the working area, that all the implemented video commands take place inside the specified bounds. For example, the player starts reading the disk at the value specified by 'Begin At >' and stops[6] at the value specified by 'End At >'. If no values are specified using these menu items, the working area is set to the entire disk.

9.3. The application in deep

Now we are going to look deeper in the application as we have done in chapter 7 for RMG basic classes.

The Video application is composed of two classes :

- **Video;**
- **VideoMTree.**

where Video is the master-class and VideoMTree the menu class[7].

This part presents first the different interesting instance variables. It is followed by the presentation of the factory methods and the different instance methods. Next comes the description of each action defined in this application. After this, we illustrate what has been said in chapter 8 concerning the mouse, the menus and the active collection.

9.3.1. *Instance variables*

There are several instance variables in the master-class which are interesting - none are present in the menu class. Among these variables we can find six of

[6] : If the user does not stop the player before

[7] : Note that the reader will find the full code of these two classes in the appendices

them which are declared as id variables and which contain the addresses of the different subviews of the application :

- '**topBanner**' will contain the address of the title bar;
- '**scroll**' will contain the address of the grey scroll bar;
- '**scrollScale**' will contain the address of the view designed to contain the graduation;
- '**bar**' will contain the address of the blue bar within the scroll bar;
- '**arrowB**' and '**arrowT**' will contain the addresses of respectively the bottom and top arrows of the scroll bar.

Seven other id instance variables are defined in order to contain the addresses of the application subviews which represent the command buttons :

- '**play**' will contain the address of the 'Play' button;
- '**stop**' will contain the address of the 'Stop' button;
- '**still**' will contain the address of the 'Still' button;
- '**inpText**' will contain the address of the button enabling the text edition window to appear;
- '**fwd**' will contain the address of the 'Fwd' button;
- '**bwd**' will contain the address of the 'Bwd' button;
- '**ed**' will contain the address of the text edition window itself.

The other instance variables are less important for the comprehension of the class.

9.3.2. *Factory methods of the 'Video' class*

One can find only two factory methods in the 'Video' class; these are the classical ones: '**newIn:**' and '**delayInit**'. They are detailed one after the other.

'newIn:' is the factory method designed to create the application's window and all its subviews. Within this method are done several things, mainly :

- the creation of the window;
- the creation of the window's subviews;
- the objectivation of C functions to be linked to particular views as actions;
- the initialization of variables;
- the setting of various parameters concerning the connection between the player and the workstation.

The creation of the window's subviews concerns the creation of the title bar, the scroll bar and the various buttons composing the application. The objectivation of the C functions mainly includes the objectivation of functions implementing the commands to be sent to the video player and functions coping with the management of the window.

'delayInit', in our case is used to initialize the application's menu and initialize two new RMG cursor for the application.

9.3.3. *Factory methods of the 'VideoMTree' class*

The 'VideoMTree' class implements also two classical factory methods : '**delayInit**' and '**getIt**'.

'delayInit' is the factory method which initializes the menu itself, which is a menu containing seven different menu items. Four of these menu items hide submenus. These submenus are in fact boxes enabling the user to type-in a file name or a frame number value from the keyboard.

'getIt' is simply a factory method that returns the address of the menu previously created.

9.3.4. *Instance methods*

In this point we analyze the instance methods of the Video class one after the other. Note that the VideoMTree class does not have any instance methods.

'**extraNewIn**' is probably the most important instance method. In our case its goal is multiple; it is designed to :

- get the address of the application's menu;
- create the Ghost instance in order to be able to iconize the application;
- create the instances which represent the various icons that can be displayed in the buttons -instead of strings.

It also issues the message '[self showAll]' to the class' instance in order to make all the views appear on the screen.

'**iconify**' is the method which calls the method 'iconize' in Video's superclass in order to iconize the application.

'**readDevice**' is the method used to read the information sent by the video player into the device file. In this method, two cases are separated :

- if the 'read data' is a number which is not preceded by any letter, then the data is a frame number;
- if the 'read data' is a letter -different from a carriage return- or a number preceded by a letter, then the data is an error message.

'**moveBar**' is a simple method enabling the bar within the scroll bar to be moved from a certain amount of pixels. This amount is contained in a global variable; this explains that this method does not require any argument.

'**update**' is the method called periodically by the system when the application is placed into the active collection. In our case it updates the position of the bar within the scroll bar when the video is in play mode or in fast forward or fast backward mode. Three cases are separated :

- the video is in play mode or fast forward mode and then the bar's position is simply updated and moved of a certain amount. This amount is calculated by subtracting the previously red frame number from the current frame number;
- the bottom of the working area is reached, the only possibility is to stop the player;
- the start of the working area is reached when the player is in fast backward mode, the only possibility is to stop.

'**startRun**' and '**stopRun**' are two methods which respectively installs and retrieves the application from the active collection.

'**buttonFace:**(id) **color1:**(int) **color2:**(int)' is a method[8] that enables to create shaded buttons on the screen, as seen in figure 9.2. This method takes three arguments. The first one is the address of the view which has to be converted with the method. The second and third ones are the two colors which will be the border colors of the button.

'**initScale:**(id) **low:**(int) **high:** (int)' initializes the graduation of the scroll bar. This initialization is done into the view specified as first argument. The two following arguments specify the bounds of the scaling. Note that in this method, eight different instance variables are used to hold the addresses of the instances representing the graduation's numbers. This solution was chosen because we had problems at execution when we tried to store these addresses in an indexed variable.

'**scale:**(id) **low:**(int) **high:**(int)' enables to rescale the graduation. This is done in the view specified by the first argument and the rescaling is done from the integer specified as second argument to the integer specified as third argument.

'**testScrlPnt**' is a method which tests the location of the bar within the scroll bar. Two cases can be considered :

- the bar is at the end of the scroll bar and is going down. In this case, if the end of the working area is not reached, the scroll bar is rescaled downwards and the bar is reset to the beginning of the scroll bar;
- the bar is at the beginning of the scroll bar and is going up. In this case, if the beginning of the working area is not reached, the scroll bar is rescaled upwards and the bar is reset to the end of the scroll bar.

---

[8] : This method was designed by Uwe Heimburger and Detlev Wegener at the Kiel's University for their work in the COLOS project and is widely used nowadays by COLOS programmers

'**posOnDisk**' enables a frame number to be transformed into a string in order to send it to the video player with a particular command. This is used mainly to position the player on a particular frame.

'**input:**(char*)' is the method used to set the starting point of the working area. It is called from the menu after entering a string composed of numbers at the 'Begin At >' menu item; this string is passed to it as argument and is transformed into an integer.

'**output:**(char*)' is the method used to set the end of the working area. It is also called from the menu after entering a string composed of numbers at the 'End At >' menu item; this string is passed to it as argument and is transformed into an integer.

'**saveFile:**(char*)' is the method called from the menu enabling the user to save all the parameters concerning a working area and the text linked to it, in the file which name is passed as argument and which is typed-in at the 'Save >' menu item of the application's menu. Note that the parameters concerning the working area are saved into the file '<file name>.vcn' and the text into the file '<file name>.vct'.

'**loadFile:**(char*)' is the method called from the menu and enabling the user to load previously saved parameters concerning a working area and the text linked to it. These parameters and text are loaded from the file which name is passed as argument and which is typed-in at the 'Load >' menu item of the application's menu.

'**quit_app**' enables to quit the application. This method uses Video's superclass 'quit_app' method and does additional things. For example, it sends a message containing the 'quit_app' selector to the text editor window if it exists. This method is called after the user having selected the '!! QUIT !!' menu item of the application's menu.

'**videoTest**' tests that a command sent to the video has been correctly executed. This is done by checking that the 'R' character arrives into the device file, sent by the video player.

'**changeButtons**' enables the buttons to be changed from strings to icons and vice-versa. It is called after selection of the 'Change Buttons' menu item of the application's menu.

'**textWindow:** (id)' is the method used to create or control -make it appear or disappear- the text editor window on the screen. The argument passed to the method is the address of the environment itself, so that this window can be created independently from any other application's window and inside the environment window itself. This text window is in fact an instance of the 'DocEdit' class, without its menu.

'**viewError:**(int) **into:**(char*)' is designed to display a box on the screen which contains an error message[9]. This error message is the message which number is passed as first argument and which is related to the file which name is passed as second argument.

---

[9] : These error messages concern mainly file errors

9.3.5. *The Actions*

Several actions have been defined together with the views of the application. These actions are all represented by C functions as explained in chapter 8. These actions are explained one by one in this point.

'**actionSelf**' is the action attached to the application's window itself and is the objectivation of the '**ac_Self(aView)**' C function. Its goal is only to make the application's window and all its subviews as the first of the hierarchy of views present on the screen; this is done using the 'RMGView' instance method: '**popToTop**'.

'**moveAction**' is attached to the title bar of the application, which address is contained in 'topBanner' and is the objectivation of the '**ac_Move(aView)**' C function. Its goal is also to pop the application's window to the top of the view hierarchy and to enable the user to move the application's window across the screen. This last thing is done using an RMG predefined C function: '**moveWithBand(aView)**'.

'**actionPause**' is attached to 'scroll' which is the instance variable containing the address of the grey scroll bar. It is the objectivation of the '**ac_Pause(aView)**' C function which goal is to send the order 'PA\r' -PAUSE- to the video player if it is not yet in pause mode, otherwise it resets the video in play mode. Note that this action is not available while the video player is in still mode.

'**scrollBarAction**' is attached to the bar within the scroll bar and is the objectivation of the '**ac_ScrollBar(aView)**' C function. This function enables the user to pick up this bar and to drag it upward and downward in order to go forward or backward on the disk. It can only be used in play or still mode.

'**scrollBarUp**' is attached to the arrow -which address is contained in 'arrowB'- situated at the top of the scroll bar and is the objectivation of the '**ac_ScrollBarU(aView)**' C function. This function enables the user to make the player go backward on the disk.

'**scrollBarDown**' is attached to the bottom arrow -which address is contained in 'arrowT'- and is the objectivation of the '**ac_ScrollBarD(aView)**' C function. This function enables the user to make the player go forward on the disk.

'**actionPlay**' is attached to the 'Play' button -which address is in the 'play' instance variable- and represents the '**ac_Play(aView)**' C function. Its goal is to send the video the PLAY command -'PL\r'- if the video is not currently in the play mode. The video will begin reading the disk at the frame number 0 if no other frame number is specified with the 'Begin At >' menu item of the application's menu.

'**actionStop**' is attached to the 'Stop' button -which address is in the 'stop' instance variable- and is the objectivation of the '**ac_Stop(aView)**' C function, which goal is to send the video the STOP command -'RJ\r'- in order to stop it completely.

'**actionStill**' is attached to the 'Still' button -which address is in the 'still' instance variable- and is the objectivation of the '**ac_Still(aView)**' C function,

which goal is to send the STILL command -'ST\r'- to the video player. Note that this C function can not be performed in pause mode.

'actionFwd', attached to the 'Fwd' button -which address is in the 'fwd' instance variable-, is the objectivation of the **'ac_Fwd(aView)'** C function. This function sends the video player the FAST FORWARD command: '250SPMF\r'. Note that '250SP' is to change the reading speed and 'MF' to specify that it has to read in 'Multi-speed Forward' mode. It is not available in still mode.

'actionBwd' is attached to the 'Bwd' button -which address is in the 'bwd' instance variable- and is the objectivation of the **'ac_Bwd(aView)'** C function. This function is designed to send the video player the FAST BACKWARD command: '250SPMB\r'. It is not available in still mode.

'actionInpText' is attached to our last button: 'Text', which address is contained into the 'inpText' instance variable. This action is the objectivation of the **'ac_text(aView)'** C function which goal is to call the 'textWindow:' instance method in order to create the text editor window if it does not exist or to make it appear or disappear if it already has been created.

The last action is **'superAction'** which is attached to the outline around a button -created with 'buttonFace: color1: color2:'. This action represents the **'superAct(aView)'** C function which executes the action linked to the superview of the view which address is passed as argument to the function. This is done so that the button's action is also executed when the user clicks on the colored surround of the button.

9.3.6. *The mouse*

We use the mouse parameters -location, state of the buttons- in several cases. In particular we use the 'Mouse' class, as explained in the preceding chapter, via its 'change::' and 'getButtons' methods. For example, in the 'ac_ScrollBar' C function, we use 'getButtons' to detect if the left button is still depressed and if it is so, we calculate the new position of the bar in the scroll bar using the 'change::' method and its second argument which is the change in the Y axis. This gives :

```
...
while ([[Mouse change:&changex:&changey]getButtons] == 1)
        {
        calculate new position of the bar with variable 'changey'
        }
...
```

In this case, the first thing done is to get the changes of the RMG cursor's location and put them into the two variables 'changex' and 'changey' -with the 'change::' method-; note that these two variables are already defined in the RMG header files. After this, the state of the buttons is 'asked' -with the 'getButtons' method-, if it equals 1 then it means that the left button is depressed and that the 'while loop' must be entered thus calculating the new position of the bar with the variable 'changey'.

9.3.7. *The menu*

The menu is defined classically as explained previously.
It is an instance of 'RMGMenu1' and is composed of seven menu items.

The first one is '!! QUIT !!', to which is linked the method 'quit_app' of the Video class :

```
[[[rootMenu at: 0 putStr: "!! QUIT !!"]
        center]
        idAction: [MTreeAct sel: "quit_app"];
```

The second menu item is 'Icon' to which is linked the method 'iconify' of the Video class :

```
[[[rootMenu at: 0 putStr: "Icon"]
        center]
        idAction: [MTreeAct sel: "iconify"];
```

The next menu item is 'Change Buttons' to which is linked the method 'changeButtons' of the Video class :

```
[[[rootMenu at: 0 putStr: "Change Buttons"]
        center]
        idAction: [MTreeAct sel:
                        "changeButtons"];
```

'Load >' is the next menu item, to which is linked a submenu called 'entryPad'. This submenu is only a text box in which the user can enter text; in this case, the entered text will be file names. These two items look like :

```
entryPad = [[[EntryPad char_wide: 12
                    max_wid: 12
                    font: aFont]
                    action: ac_load]
                    viewIcon: rootMenu];
...
[[[rootMenu at: 0 putStr: "Load >"]
        center]
        subMENU: entryPad];
```

This entryPad is an instance of an RMG class: '**EntryPad**', which is in fact an RMGMenu with one menu item; this menu item is modifiable by the user and enables him to enter text strings. It is instantiated with EntryPad's '**char_wide:**(int) **max_wid:**(int) **font:**(id)' factory method. The first argument of this method specifies the number of characters that the entry pad displays on the screen, in other words the number of characters the user sees at once. The second argument specifies the maximum number of characters that the entry pad can

contain. The third argument specifies the font in which these characters are displayed.
This method creates the entry pad but does not attach any action to it, this is why we use the 'action:' instance method[10] to specify the C function which has to be executed when the entry pad is selected in the menu. In this case the function is '**ac_Load(aView)**' which enables the user to enter a string into the entry pad and then sends a message to the instance of Video containing the 'loadFile:' selector with the entered string as argument.
The three last menu items -'Save >', 'End At>' and 'Begin At>'- follow the same principle as for 'Load >'. Each of them has a submenu which is in fact an instance of EntryPad to which is attached a C function by which the user is able to enter a string, this string being passed as argument to a method of the Video instance. This functions and methods are :

- 'ac-Save(aView)' and 'saveFile:' for the 'Save >' menu item;
- 'ac_Input(aView)' and 'input:' for the 'Begin At>' menu item;
- 'ac_Output(aView)' and 'output:' for the 'End At>' menu item;

9.3.8. *The active collection*

The active collection is used in our case for the scroll bar. Indeed, in order to make the scroll bar follow the 'reading' of the player on the disk, one must continually send frame requests -'?F\r'- to the video. This can be done by entering a loop which sends the frame request, reads the result, updates the scroll bar and so on. The problem with this is that it does not stop, as the mouse loop -see chapter 8- is short-circuited by the loop described above; the user is not able to click on the stop button to stop the video and the loop.

This is a problem which can be resolved by using the active collection. We place the application in the active collection whenever the video player is started, using the 'startRun' method. We retrieve the application from the active collection whenever the video player is stopped, using the 'stopRun' method. So when the application is placed into the active collection, the system sends the master-class a message containing the 'update' selector. In our case, the 'update' method :

- sends a frame request to the video player;
- reads the reply of the video player, for instance a number if no error occurs;
- updates the position of the bar within the scroll bar;
- makes tests about the position of the player within the working area.

[10] : Owned by RMGView, to which EntryPad is linked by inheritance

9.4. The classes used

In this part, we describe briefly the various RMG classes used in this application. This enables us to introduce new tips about RMG programming which can be of a certain use for the reader.

We start by describing the Fixtur17 class, then comes the ModStrI class, followed by the RMGLine class. A last point considers the problem of using ones own icons in a program with the IconModel, RMGIcon and SysIcon classes.

9.4.1. *Fixtur17*

We used, apart from the traditional classes described in chapter 7, several classes accessible to the user in the RMG environment. The first of these classes is '**Fixtur17**' which enables the user to **use RMG predefined 17 by 17 pixels icons** in their own program, these icons representing arrows pointing right, left, etc. This class is composed of factory methods, each of these representing the display of a particular icon.

For example, if one sends the message :

```
[Fixtur17 topRtIcon: self extent: 17 type: 2];
```

where 'self' contains the address of a view, an arrow pointing diagonally to the top right is created in 'self' with an extent of 17 pixels and sticking to the right of the view.

In our case we used this class to display the two arrows respectively on the top and at the bottom of the scroll bar. This was done by using two factory methods :

- '**up_arrow:**(id) **extent:**(int) **type:**(int)';
- '**down_arrow:**(id) **extent:**(int) **type:**(int)'.

Note that this class helps only the user display icons, it does not attach any actions to them. This is done by the programmer as for buttons -this is explained in chapter 8-, using the methods defined in RMGView, as Fixtur17 is linked to it through inheritance.

For example, in the case of the top arrow, we have in the 'newIn:' method :

```
...
if(!scrollBarUp)
    scrollBarUp = [[RMGAction newFuncAction: ac_scrollBarU]
                                    myCursor: iconCursor];
arrowB = [[[[[Fixtur17 up_arrow: scroll extent:17 type: StickTop]
                        viewIcon: self]
                        bkgd: BLUE]
                        idAction: scrollBarUp]
```

show];

...

In this case the first thing done is the objectivation of the C function to be attached to the arrow, in this case 'ac_scrollBarU'.
The other instance method used, '**myCursor:**(id)', is a method of the 'RMGAct' class which is the superclass of RMGAction. This method sets the RMG cursor associated with this action to the icon which address is contained in the variable passed as argument -see point 9.4.4. and 'SysIcon' class for more precisions.
Next comes the instantiation of Fixtur17, which in this case displays an arrow pointing upwards, sticking to the top of the scroll bar identified by the 'scroll' instance variable, in a blue background, to which is attached the 'scrollBarUp' action. Note that 'StickTop' and 'BLUE' are two constants defined in the RMG header files and both correspond to the integers identifying respectively the fact that a view sticks to the top of another one and the blue color.

9.4.2. *ModStrI*

'**ModStrI**' is another class we use in our program. This class has been developed by Uwe Heimburger and Detlev Wegener at Kiel's University[11] and is a specialization of the RMG class 'RMGModStrI'. The advantage of ModStrI is that it has certain methods which are simplified compared to RMGModStrI. Anyway its goal is to **write a modifiable long integer in a view**.

We use this class to display the numbers composing the graduation of the scroll bar. We instantiate ModStrI using the 'font: superview: color:' factory method of the RMGString class, as these two classes are linked by inheritance.

We use also instance variables among which one can find '**string:** (char*): (char*)' which is a method implemented into RMGModStrI. This method enables the user to specify two strings which are displayed respectively in front and after the integer.

'**dataOrigin:**(int):(int)' is used to specify the location where the integer has to be displayed, in our case at the right of the scroll bar. This is a method inherited from RMGString.

We use '**setIValue:**(int)' to specify the value of the integer to be printed. Note that after using this method, one has to issue a message to the ModStrI instance containing a 'show' selector, as using the 'setIValue:' changes the number in the instance variable designed to contain it but does not update the screen.

'**setRJust:**(int)' is a method that enables to specify the size of the number in order to have right justification of the decimal point.

For example, one of the integers composing the graduation was defined in the

[11] : Precisely at the Institut für die Pädagogik der Naturwissenschaften an der Christian-Albrechts Universität zu Kiel, under the supervision of Dr. Hermann Härtel for the COLOS project

'initScale: low: high:' method as :

```
...
str1 = [[[[[[ModStrI font: aFont superview: aView color: BLACK]
                    string: "": "-"]
                    dataorigin: 0: 346]
                    setIValue: temp]
                    setRJust: 5]
                    show];
...
```

where 'str1' is an instance variable which contains the address of the ModStrI's instance; where 'aFont' identifies an instance of FontMngr; where 'aView' identifies the view where the graduation has to be displayed; where 'temp' is the integer to be displayed.

9.4.3. *RMGLine*

'**RMGLine**' is the next class we use, that **draws a line between two points** which are specified by the user.

We instantiate this class using its factory method: '**superview:** (id) **color:** (int)' and the instance method: '**relP1:** (int): (int) **length:** (int): (int)'. This last method determines in relative coordinates the location of the two points between which the line has to be drawn. The two first arguments -let us say x and y- determine the location of the first point, relative to the inner bounds of the line's superview. The two last arguments -let us say lX and lY- determine the value to be added to the location of the first point to find the second point's location. In other words, the first point's location in absolute coordinates relative to the whole screen is determined as :

(x+superview's inner left bound, y+superview's inner low bound)

The second point's location in absolute coordinates relative to the whole screen is determined as :

(x+superview's inner left bound+lX, y+superview's inner low bound+lY)

For example, if one wants to draw an horizontal black line 100 pixels long at location (10,10) into a view -let us call aView the variable containing the address of this view- which lower left corner is at location (10,10) on the screen, he has to issue the message :

```
[[[RMGLine superview: aView color: 0]
        relP1: 10: 10 length: 100: 0]
        show];
```

Which causes the first point to be at absolute coordinates' location (20,20) -(10+10,10+10)- and the second point at (120,20) -(10+10+100,10+10).
Note that RMGLine is a subclass of RMGView, enabling the user to utilize any of this class methods.

9.4.4. *Displaying personal icons*

In order to **display personal icons**, one has to do a certain number of things and use a certain number of classes. In our case, we do these different things in order to display icons on the different buttons controlling the video player and in order to change the application cursors.

The first thing we do is to create the icon itself using the IconEdit application - see chapter 4- and saving it in a file.

The second thing we do is to create an instance of the class '**IconModel**' which **stores some attributes concerning an icon**, such as its dimensions or the file name in which it is stored. We do just this in the 'extraNewIn:' method, for example :

```
modelPlay = [[IconModel readFile: "/usr/RMG/DATA/ICONS/Play.icon"]
                                        hotSpot: 0: 0];
```

where modelPlay identifies the IconModel instance of the icon which has to be displayed in the 'Play' button. In this case, we use the '**readFile:**(char*)' factory method and the '**hotSpot:**(int):(int)' instance method. The first one just creates an instance of IconModel with the specified file, while the second one sets the reference point of the icon. This reference point being relative to the upper left corner of the icon. For example, when the icon is shown at a location (x,y), it is the reference point that is put on the top of this location (x,y).
This creation of an instance of IconModel is required by RMGIcon or any other RMG class enabling to use user-defined icons.

The third thing we do is use these instances of iconModel to **define RMGIcon instances or SysIcon instances**. The first ones are the icons displayed in the buttons, while the second ones are icons used as RMG cursors in the application.
In the case of RMGIcon, we have for example in the 'changeButtons' method :

```
...
iconPlay = [[[RMGIcon makeFromModel: modelPlay
                      in: play
                      at: 25: 4]
                      idAction: superAction]
                      show];
...
```

where the factory method 'makeFromModel: in: at::' is used in order to create the instance of RMGIcon by specifying the IconModel to be used, the view in which it has to be displayed and the location in this view. We use also 'idAction:' to attach to this icon the 'superAction' action, which enables the action of the icon's superview to be executed when the icon is clicked on.

In the case of SysIcon, we have for example in the 'delayInit' method :

```
...
iconCursor = [[SysIcon newIn: NULL]
                              icon: modelCursor mask: modelCursor];
...
```

where '**newIn:**(id)' is used as factory method and has NULL as argument. This is done this way because we want this cursor to appear only when the user clicks on the desired screen object, to make the user notice that a particular action is currently performed by him clicking on this screen object. The instance method '**icon:**(id) **mask:**(id)' is used to specify the icon which has to be used as RMG cursor, in our case 'modelCursor'.

9.5. Problems encountered

We are going to talk about the main problems we encountered in the creation of this application. The first one is the problem of creating and setting up an accurate device file. The second problem is the one of quitting the application without problems. The third point is a short critique of our application.

9.5.1. *Creating and setting a device file*

One of the great problems of this application was to create a device file for the video player and to set the correct parameters concerning the connection between the video player and the workstation.

The creation of the device file was done using the Unix '**mknod**' command with a certain number of arguments, this is not explained as it is not our purpose in this work.

The greatest problem was to find the way to change the different default parameters of the device file in order to satisfy the requirements about the connection between the player and the workstation. This is done mainly at the beginning of the 'newIn:' method using the C function: '**ioctl()**' :

```
...
#include "sys/termio.h"                    /* 1 */
...
fp = open("/dev/ttyd00,O_RDWR);            /* 2 */
...
errflag = ioctl(fp,TCGETA,&param);         /* 3 */
param.c_cflag |= CLOCAL;                   /* 4 */
param.c_cflag &= HUPCL;                    /* 5 */
```

```
param.c_cflag ^= 0000011;                /* 6 */
param.c_iflag |= IXON;                   /* 7 */
param.c_iflag |= IXOFF;                  /* 8 */
param.c_iflag &= ECHO;                   /* 9 */
param.c_cc[VMIN] = 0x0;                  /* 10 */
param.c_cc[VTIME] = 0xff;                /* 11 */
errflag = ioctl(fp,TCSETA,&param);       /* 12 */
...
```

Line 1 specifies a header file to be included, this file defining certain variables and constants required by the ioctl() function.
Line 2 opens the already created device file which is situated in the '/dev' directory. This file is open in read and write mode.
Line 3 loads in the 'param' variable which is defined in the 'termio.h' header file, the various parameters of the device file, this is specified by the second argument: 'TCGETA'.
With line 4 starts the setting of the various parameters. At line 4, a field of the 'param' variable which represents the control modes -'c_cflag'- for the device file is set to 'CLOCAL' which means that the communication is done on a local line, not on a dial-up line via modem.
Line 5 specifies by default that the connection must be closed after the last close of the device file. Line 6 sets the baud rate to 4800 bauds.
Line 7 starts the setting of the parameters in a field -'c_iflag'- which represents the various input modes. Line 7 and 8 sets this field to enable start/stop of output control -'IXON'- and start/stop of input control -'IXOFF'. Line 9 enables an echo when a character is received.
Line 10 represents the minimum number of characters that should be received when the read operation is satisfied, so when the characters are returned to the user. In our case we set it to 0.
Line 11 represents a timer that is used to timeout data transmissions. In our case it is set to 255.
Finally line 12 rewrites the new settings contained in the 'param' variable into the device file using the 'TCSETA' request.
The new parameters are then active on the connection and this until the closing of the device file which is done in the 'quit_app' method.

9.5.2. *Problems while quitting the application*

We had problems while quitting the application.

Usually, when one sends a message containing the 'quit_app' selector to an application, it is quitted freeing each instance variable, each variable and all the space occupied by the application itself.

Our problem was mainly due to the fact that certain id variables containing the addresses of some instances of classes were not liberated when the application

was left, thus not liberating the space occupied by the instances they were identifying.

We discovered that these variables were the ones containing the addresses of the instances of IconModel, of RMGIcon and the instance variable containing the address of the text editor window.

In order to solve this problem, we redefined the 'quit_app' method in which we do several additional things :

- quit the text editor window;
- free different variables;
- quit the application itself.

Quitting the text editor window is done by sending the instance of DocEdit we created, a message containing the selector 'quit_app' :

```
...
[ed quit_app];
...
```

where 'ed' contains the address of the instance of DocEdit.
Freeing the different variables is done by sending the instances they represent a message containing the 'free' selector, for example :

```
...
[modelPlay free];
...
```

The application is quitted by sending its superclass the message containing the 'quit_app' selector :

```
...
[super quit_app];
...
```

We are insisting on this point so that future programmers pay enormous attention to the variables they define and the fact that many errors at execution can come only from the fact that instances and variables have not properly been freed.

9.5.3. *Critique*

We would like in this point to make a short critique of the video application.

This application was the first serious one we realized under RMG and is far from being perfect. It is the case specially for the scroll bar which is far from being accurate in certain cases. It is also the case with the video player. Indeed, this application is only adapted to one video player, the Pioneer 4200. This is due

to the fact that the set of commands varies from video to video and that we had direct access to only one of them. Anyway, this problem can be easily solved by changing the constants representing the different commands at the beginning of the Video class.

Concerning the video player again, a very restricting point is that this application works properly only with a plugged in and powered Video player and if the proper device file exists in the Unix file system.

We probably could have done things faster and better by using more existing RMG classes, but the fact is that these classes are not very easy to use at the first approach, due to the fact that a lot of methods are implemented into one class, causing the user to be a little confused when it comes to using them as sometimes their use is not very well explained.

9.6. Summary

In this point, we looked at an application we created under RMG for the COLOS project, the Video application.

First we gave a general description of the application itself, seeing that it was designed to control from the RMG environment a laser video disc player. We saw also that it was designed to enable teachers and learners to use new techniques represented by new medias, along with the power of the RMG environment and its simulations. We saw also that the hardware consisted in an HP workstation and a video player, which were linked together using the workstation's serial port and the video player's special compatible output port; this connection required also particular settings in order to work properly.

The communication between these two entities was done using a command and reply system, using the ASCII code, through a device file.

We saw further that this application was designed to control the video player basically, that is sending play, stop or still commands to the player. It was also designed to enable the user to control the video player in fast forward or fast backward mode. The user could also :

- specify a working area which bounds the video player could not exceed;
- use a text editor window in order to type-in text concerning a working area;
- save and load the parameters and text concerning a working area.

A second point described what the user could see on the screen while using the application.

The third point tried to describe the application a little deeper. We saw that it was composed of two classes :

- the master-class : Video;
- the menu class : VideoMTree.

We then described the interesting instance variables, factory methods, instance methods and actions declared or implemented in these classes. This was followed by an illustration of what had been said in chapter 8 about the mouse, menus and the active collection, all this applied to our application.

The fourth point concerned the description of the new RMG classes we used in our program. We could find :

- 'Fixtur17' which enables the user to use RMG predefined 17 by 17 pixels icons in their own program, these icons representing arrows pointing right, left, etc;
- 'ModStrI' which displays a modifiable long integer in a view;
- 'RMGLine' which draws a line between two points which are specified by the user.

We also described how we had used icons designed with the RMG IconEdit application, to be displayed in views or to be defined as RMG cursors. This involved the three classes: IconModel, RMGIcon and SysIcon.

Our last point talked about problems we encountered while designing our application. This involved :

- the creation of the device file and the setting of the various parameters concerning the connection;
- the 'freeing' of various variables and instances;

This point was terminated by a short critique concerning our application.

THIRD PART: Illustrating telecommunication principles under RMG

Chapter 10: Representing computer telecommunication under RMG

This chapter is mainly centered on the presentation of the university of Namur's task in the COLOS project: representing computer telecommunication under RMG.

Telecommunication is a subject taught at Namur's university to many students, these courses mainly touching the **OSI[1] model of the ISO[2]**. This model is not at all easy to understand for students, specially as it is a matter where practical works and real simulations are very difficult to build. That is the reason why Namur wants to build RMG simulations for the COLOS project[3], concerning different aspects of the OSI model, in order to help students perceive this subject more easily.

We start by presenting shortly the OSI model by giving OSI basics (10.1). We end by giving the different steps and projects which are germinating in Namur's COLOS team, concerning the representation of the OSI model under RMG (10.2).

10.1. OSI basics

We are going to underline briefly the basics of the OSI model. (From [TANENBAUM, 89] and [HENSHALL, 88]).

OSI is **a standard** which has been developed by the ISO which is responsible for developing a wide range of standards covering a lot of technical matters. The aim of OSI "*is to provide communication-based user services that operate between heterogeneous computer systems*" ([HENSHALL, 88]). This is done by helping the user in two ways :

- suppressing his dependency towards a commercial supplier;
- offering wide possibilities and services concerning communications between people : the exchange of files, electronic mail, etc.

We try now to describe this model in a few words.

In order to communicate with each other, two users on different computers utilize on their own computer the services of OSI stacks. Each stack is divided into **seven layers** (see figure 10.1) and was created by ISO as a reference model -the **ISORM**- in order to simplify the understanding of the OSI model.

---

[1] : Open Systems Interconnection

[2] : International Standard Organization

[3] : Under the supervision of Professor Philippe van Bastelaer

Layer 7	**Application**
Layer 6	**Presentation**
Layer 5	**Session**
Layer 4	**Transport**
Layer 3	**Network**
Layer 2	**Link**
Layer 1	**Physical**

Figure 10.1 The seven layers of the OSI reference model

Each layer has an active element -a process, a piece of hardware, etc- which is called an **entity;** entities in the same layers but on different machines are called **peer entities**. For example, the layer 4 -transport layer- entities are called transport entities.

The purpose of an entity is to **offer and implement services for the upper layers**. At this stage the advantage is that the layer using a lower layer's service is completely unaware of how this service is implemented. An entity offering services to another upper entity is called the **service provider**, while the user of the services is called the **service user**. The services offered are accessible at **SAPs** or Service Access Points, in the interface between two layers.

Further, a stack's layer entity in one machine has a conversation with the same layer entity in the stack of the other machine. This conversation is ruled by conventions and principles known as **the protocol**, these entities exchanging messages known as **PDUs** or Protocol Data Units. For example, two Transport layers exchange T-PDUs -or Transport Protocol Data Units. In reality, in a conversation between two entities no data is directly transferred from a layer to its correspondent in the other machine. The data is passed with interface control informations to the lower layer through one of the interface's SAP, the upper layer requesting thus a service from the lower layer. The set constituted by the data and the control information is called an **IDU** or Interface Data Unit and is composed of a **SDU** or Service Data Unit and an **ICI** or Interface Control Information (see figure 10.2, inspired of [TANENBAUM, 89] p.22). The process described above is performed until the lowest layer is reached, where is situated the physical communication medium through which actual communication occurs.

In the seven layers, the top one, the Application layer, differs slightly from the other ones in that it makes OSI services available to the users of the computer system on which it resides. This layer includes a lot of system-independent applications enabling the user to do file transfer, E-mail, remote job management, etc.

The Presentation layer performs certain particular functions among which it is concerned with the solving of problems concerning the syntax and the semantics of the transmitted informations.

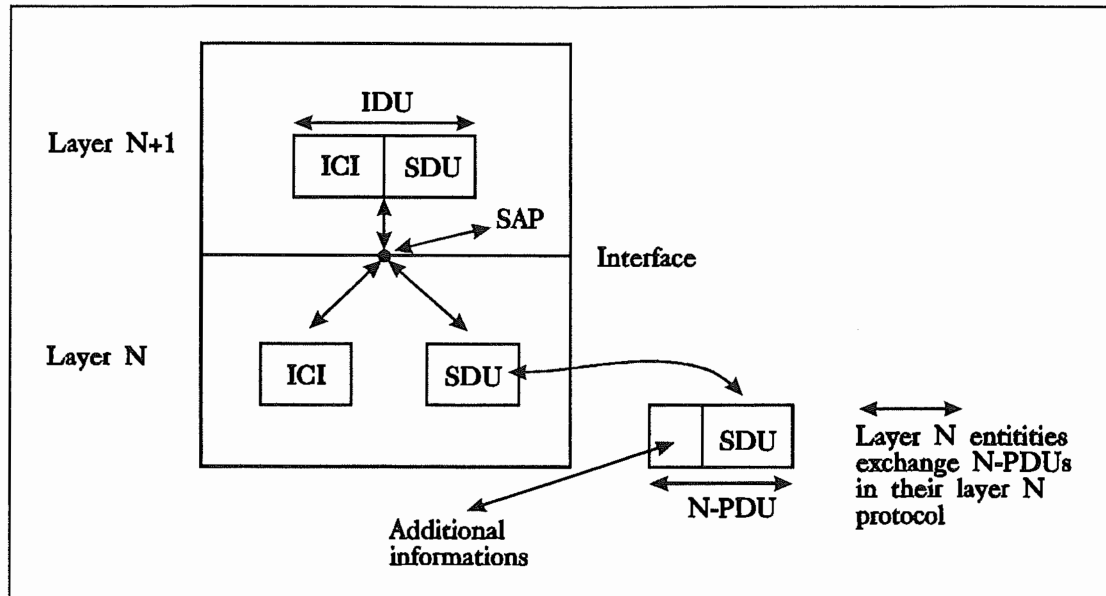


Figure 10.2 Relations and activities between layers at an interface

The Session layer's main goal is to allow users to establish a session between themselves.

The Transport layer is designed to accept data from the session layer, possibly split them into smaller pieces, pass them to the lower layer which is the Network layer and finally ensure that all the pieces arrive correctly at the other end.

The Network layer is mainly designed to manage the operations of the subnet.

The Data Link layer tries to make a transmission facility appear like a transmission line free of errors.

Finally the Physical layer is concerned with transmitting raw bits on a transmission media.

10.2. The different steps in representing the OSI model under RMG

We are now going to stress the different steps which can be seen in representing the OSI model under RMG, for the COLOS project. Two of these steps or part of them are already started, the other ones are projects that Namur's COLOS team tries and will try to realize.

The two applications which have already started are :

- 'OSI on X-25';
- 'Ftam';

The first application was realized by Véronique Nachtergaele and ourselves. It concerns the simulation of the opening and closing of a connection in the OSI model, between two machines, both of them linked to the X-25 network. This application is explained in length in the following chapter.

The second application was realized by Dominique Corbugy and Joël Denis. It is a simulation which enables the user to "*familiarize himself with certain FTAM*[4] *primitives in combination with a simple state machine enabling to more easily understand the system*[5]" ([CORBUGY, 90]).

Namur's COLOS team has several projects concerning OSI simulation under RMG; this can be divided into several steps.

The first step consists in the adaptation of the OSI on X-25 application in order to be able to present the various **primitives used** when a connection is opened or closed. A second step will make the different **PDU**s appear but without the detail of their composition. The third step will enable the user to make a zoom on the PDU in order to see its composition.

A following step will consider the proper **functionalities and the protocols** of the second, third and fourth layer of the OSI model. This will be followed by the development of the **Session and Presentation layers**. Yet another step will develop illustrations through different applications like **X-400** for example.

The COLOS team would like also to consider the OSI model implemented on a **LAN** (Local Area Network), consider the **interconnection** of networks and the **routing** problems. This could also be deepened by the simulation of specific applications like **EDI** (Electronic Data Interchange). But also by the simulation of disciplines as different as **N-ISDN** (Narrow Band Integrated Service Digital Network) and **B-ISDN** (Broad Band Integrated Service Digital Network) or **network management**.

These are few of the projects which are germinating in Namur and which can be of a great interest for the educational world and the COLOS project.

10.3. Summary

This chapter's main goal was to present the task of Namur's University in the COLOS project: representing computer telecommunication under RMG and specially representing the OSI model.

We started by describing briefly the OSI model and its principles. We saw that it was a standard designed by ISO and that its goal was to provide communication-based user services that operate between heterogeneous computer systems. We saw also that this model 'operated' through seven layer stacks, the layer's entities 'talking' to each other using special protocols and implementing some services for the layer entities situated above themselves.

We ended by giving few of the projects that are in the minds of the COLOS team members in Namur.

[4] : File Access, Transfer and Management

[5] : For further informations on the 'Ftam' application, the reader will consult the thesis presented by D.Corbugy and J.Denis: [CORBUGY, 90]

Chapter 11: The OSI on X-25 application

This chapter presents the 'OSI on X-25' application which was created by Namur's COLOS team. This application was created following the principle of a scenario and was programmed in the RMG environment, building new classes and using already existing ones. Note that this application is still a prototype and has still to be perfected.

The first point presents the principles of a scenario (11.1). The second point presents the 'OSI on X-25' scenario (11.2). In the third point we see some details concerning the application's implementation (11.3). Point 11.4 to 11.9 describe the various classes composing the application. Finally point 11.10 makes a brief critique of the 'OSI on X-25' application.

11.1. The principles of a scenario

To build our application we used the principles of a scenario. This method can be compared to the method followed by Kel Crossley and Les Green presented in [CROSSLEY, 90].

This method has to be divided into three different steps :

- deciding what the application does;
- deciding what the application is composed of on the screen;
- designing the scenario, by specifying the linking of different screens.

The first step is to build a document specifying **what the application does** and its various functionalities. This must be done thoroughly to suppress any possibilities of mistakes or misunderstandings which can lead to obsolete work in the future. In the case of simulations designed to help CAL -Computer Assisted Learning-, we think that the best way to do this is that teachers and programmers discuss the matter in length, in order to design the best primary specifications, eventually asking students to participate in order to have a user point of view.

The second step is to **design the application graphically**, that is decide of the screen objects which composes the application -for example using menus, buttons, etc. This is a very important step as we are building simulations in a graphical environment and as the perception of the subject by the student depends highly on the graphical presentation of this application.

The third step involves the results of the two preceding ones and intends to **build a 'film'** of what happens when the student is using the application. This film uses the graphical design made at the second step as basis and the specifications made at the first step as goal to achieve and as line of conduct for the progress of the simulation.

Though we lack references on this subject, we think that this is a good outline for a method intended to design simulation applications, specially in a graphical

environment like RMG. It enables one to see what are the approximate results appearing on the screen, before the programming is done.

11.2. The 'OSI on X-25' scenario

In this point, we present the 'OSI on X-25' scenario.

We start by presenting the goal of the application. This is followed by the presentation of the screen composition. Finally, the evolution of the simulation is presented in a last point.

11.2.1. *Goal of the application*

The application's goal is to simulate the **opening or closing of a connection** between two layer entities situated in two different machines, following the principles of the **OSI model**; both machines are thus working as an ISO reference model -see chapter 10. These machines are linked to a network via an entry node which works also following the OSI model principles, but is implemented as an ISO reference model containing only the three lower layers[1] -Physical, Link and Network. Note that this application is designed to be used by the teacher as a support and illustration for his course and not by the learners themselves.

Several hypothesis are made concerning the application, in order to simplify the subject :

- the simulated connections are based on the X-25 network;
- the connections can not be initiated from the Physical layer, from the Session layer and from the Presentation layer;
- the connections initiated between a machine and its entry node can not be initiated between layer 3 entities;
- the closing of a connection -or disconnection- can only be initiated from the highest layer connected;
- the opening of a connection can not be stopped or aborted during its negotiation;
- the Application layer -layer 7- is divided into two parts the SASE -Specific Application Service Element- and CASE -Association Control Service Element.

In this application no primitives nor PDUs are shown; the only goal is to show the user what happens when two users try to open or close a connection between their machines while using the OSI standards.

The user is able to choose either to open a connection, either to close a connection. In any case, any of these possibilities can be chosen, the application

[1] : For more informations or details the reader can consult [TANENBAUM, 89] or [HENSHALL, 88]

reacting accordingly. For example, the user can ask to close a connection though there is no connection opened; in this case an error message is displayed.

If the user chooses to open a connection and if it is possible, he is requested to choose two layers, each one part of one of the two machines or entry nodes included in the connection 'opening'. Note that the user can choose between two modes for this opening : continuous or step by step. In the first case, no interaction is required between the user and the application. In the second case, the opening is done one step after the other and the user has to ask the application to 'activate' the following step.

If the user chooses to close a connection and if it is possible, he is requested to choose one layer only, from which he wants the disconnection to be initiated. The disconnection is only available in step by step mode.

Each step of the process is explained via messages or warnings displayed in a 'message box'. Questions can also be asked to the user concerning the evolution of the simulation.

11.2.2. *Screen composition*

The screen composition is made of six different objects :

- four stacks;
- an ellipse;
- a message box.

A stack is a rectangle representing a machine or an entry node. It is itself composed of smaller rectangles, each of them representing a layer which has its own color. In particular, the machines are represented by stacks composed of seven layers, where the Application layer is divided in two parts -the SASE and the CASE-; the entry nodes are represented by stacks composed of three layers.

Between the layers are placed smaller rectangles which represent the interfaces between the layers (see figure 11.1).

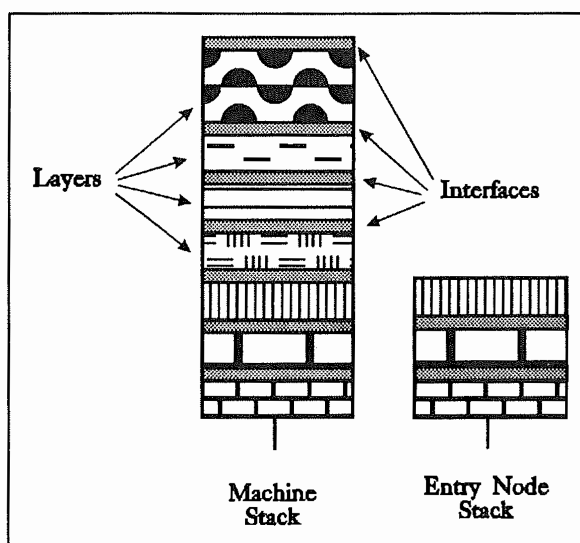


Figure 11.1 The different stacks

A layer's color is in pastel shade when this layer is not involved in a connection; otherwise, its color turns to be brilliant.

The ellipse is there to represent the X-25 network and has no other utility for the moment.

The message box is there to receive messages, warnings or questions designed to help the user utilize the application. The application window can be seen in figure 11.2[2], where the box in the lower right corner is the message box. Note that the names of the layers are displayed at the left-hand side of the application window.

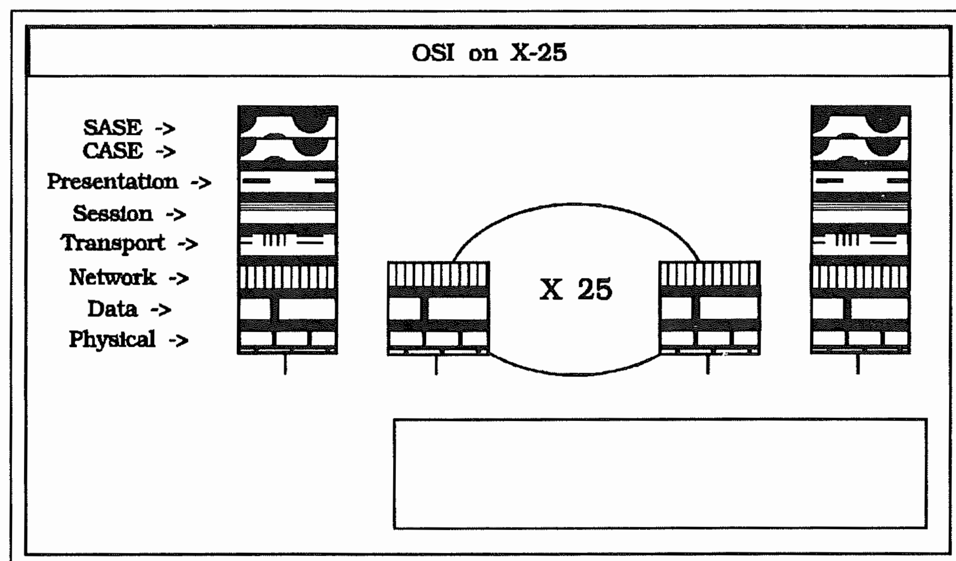


Figure 11.2 The 'OSI on X-25' application's screen

11.2.3. *Designing the scenario's evolution*

We give here the different steps and explanations concerning the design of the scenario's evolution. We do not give any hints about the implementation, this is done later; specially we do not specify if a user-application interaction is done using a menu or using a button. This is done so that the reader can follow this step without thinking about RMG implementation.

When the application is started, no connection is established, thus each layer is in its pastel color.

Three possibilities are always offered to the user at any moment :

- quit the application;
- open a connection;
- close a connection.

If the first possibility is chosen, the application is closed.

[2] : The reader can also find a screen copy of the application's window in the appendices

In the second case, if it is possible for the user to open a connection, a message appears in the message box :

Select the Source Layer

and the user has to select the layer from which the connection has to be initiated. After selecting the source layer, a second message appears in the message box :

Select the destination layer

Here, the user has to select the layer which has to be the layer answering the connection request.
If the two selected layers can not be connected, a message is displayed in the message box :

These layers can not be connected

When the two layers are selected and can be connected, a rectangle with a red outline appears between the two selected layers and at the same time they start to flicker; all this is done to indicate that the connection opening is in progress. A message is also displayed in the message box :

Opening <layer name> connection by layer <layer number N> entities, between layer <layer number N+1> entities

where '<layer name>' is the name of the layer currently involved in the connection opening; where '<layer number N>' is the number of the layer currently involved in the connection opening; where '<layer number N+1>' is the number of the upper layer for which the connection is opened at the current level. This goes on with the rectangle with a red outline going down progressively until the bottom layer is reached or the first layer where a connection is already established.
For example, if the user chooses to open a connection between layer five entities - assuming that no connection is already opened-, the 'initiating' side being the left hand side machine stack and the 'responding' side being the right hand side machine stack, the first step in the simulation is that a rectangle with a red outline appears between the layer five entities, the two layers begin to flicker and the string :

Opening session connection by layer 5 entities, between layer 6 entities

appears in the message box (see figure 11.3).

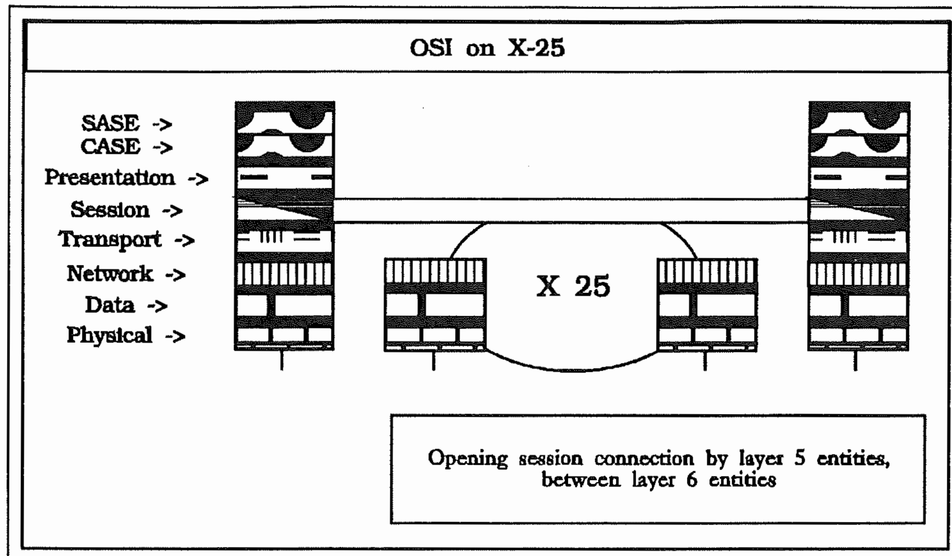


Figure 11.3 Opening connection between layer 5 entities

The next step involves layer four. The rectangle with a red outline disappears between the layer 5 entities and appears between layer four entities, the two layers begin to flicker and the string :

Opening Transport connection by layer 4 entities, between layer 5 entities

appears in the message box (see figure 11.4).

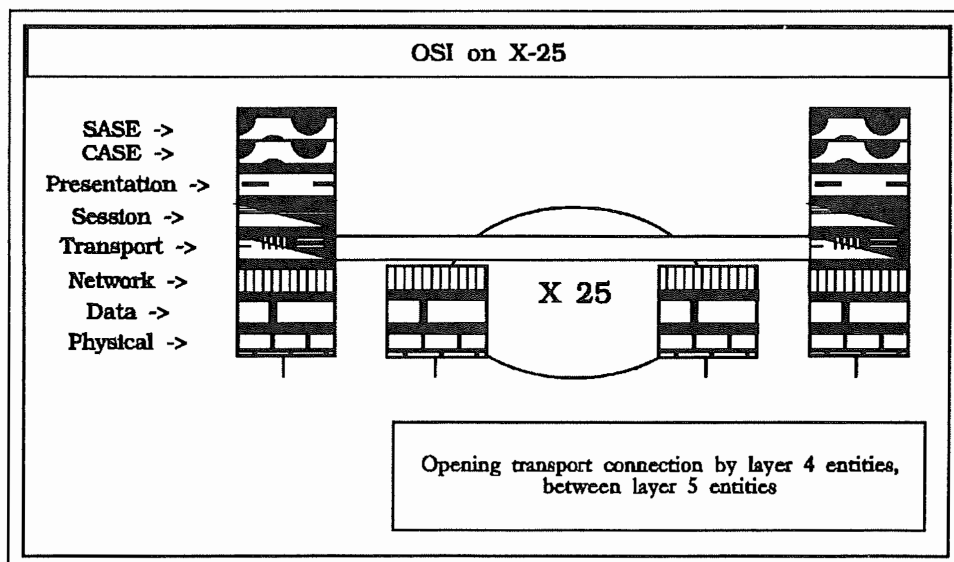


Figure 11.4 Opening connection between layer 4 entities

This keeps on until layer one is reached and thus until a connection opening is in progress between the layer one entity situated in the machine stack and its entry node stack (see figure 11.5). **(Example 11.1)**

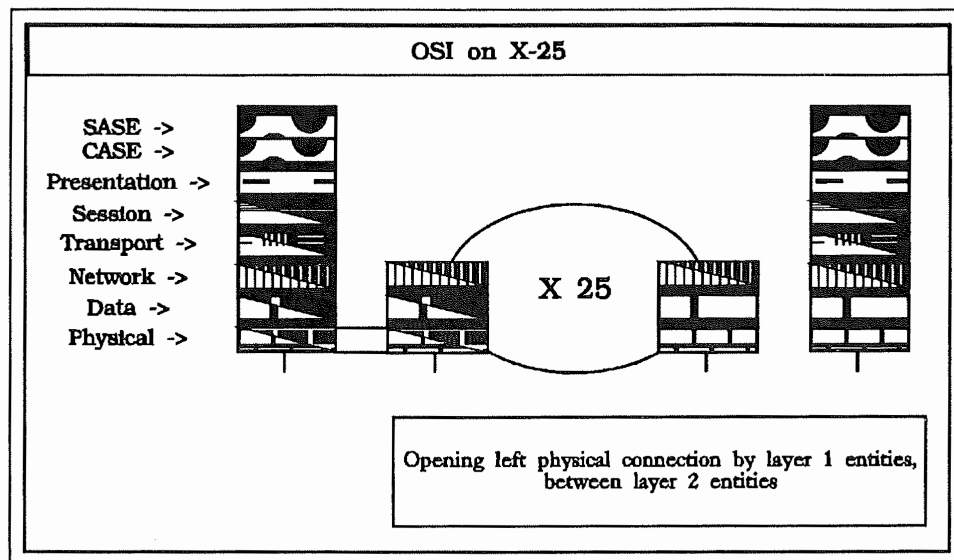


Figure 11.5 Opening left physical connection

When a connection is opened at each level in the stacks, starting from the two selected layer entities, various things change on the screen :

- the color of the layers involved in a connection turn to brilliant;
- the rectangle with a red outline disappears;
- a white vertical bar appears in the middle of the layers involved in a connection, symbolizing the fact that a communication means is available for the upper layers.

Further, when a connection is opened between layer one entities, a white bar appears between the machine stack and its entry node stack, symbolizing the fact that a connection is established between them. A white bar also appears on the top of the entry node stack between itself and the X-25 network, when a connection is established between layer three entities. This symbolizes the fact that a connection is established through the X-25 network.
A string is also displayed into the message box, explaining the current step. It is of type :

<layer name> connection opened by layer <layer number N> entities, between layer <layer number N+1> entities

where '<layer name>' is the name of the layer where the connection is opened; where '<layer number N>' is the number of the layer where the connection is opened; where '<layer number N+1>' is the number of the upper layer for which the connection is opened at the current level.

If we extend example 11.1, we see the creation of a white bar between the left hand side machine stack and its entry node stack. We see also that the two layer one entities change color and that a vertical white bar appears in their middle.

The string :

Physical connection opened by layer 1 entities,
between layer 2 entities

is displayed in the message box (see figure 11.6). The same thing happens with the layer two entities.

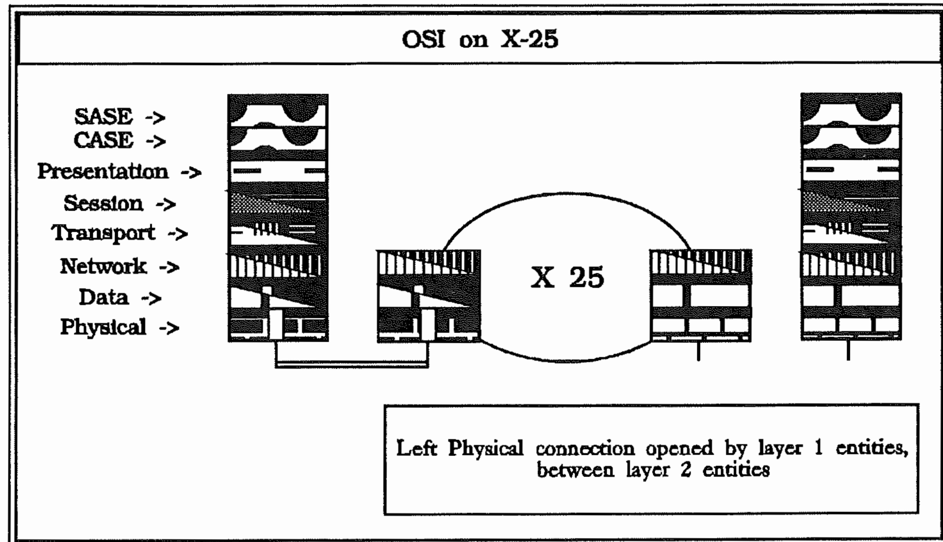


Figure 11.6 Left physical connection created

When the simulation reaches layer three, the connection is not immediately opened at that level, it must first be opened between the right hand side machine stack and its entry node stack. This is done exactly the way described above (see figure 11.7).

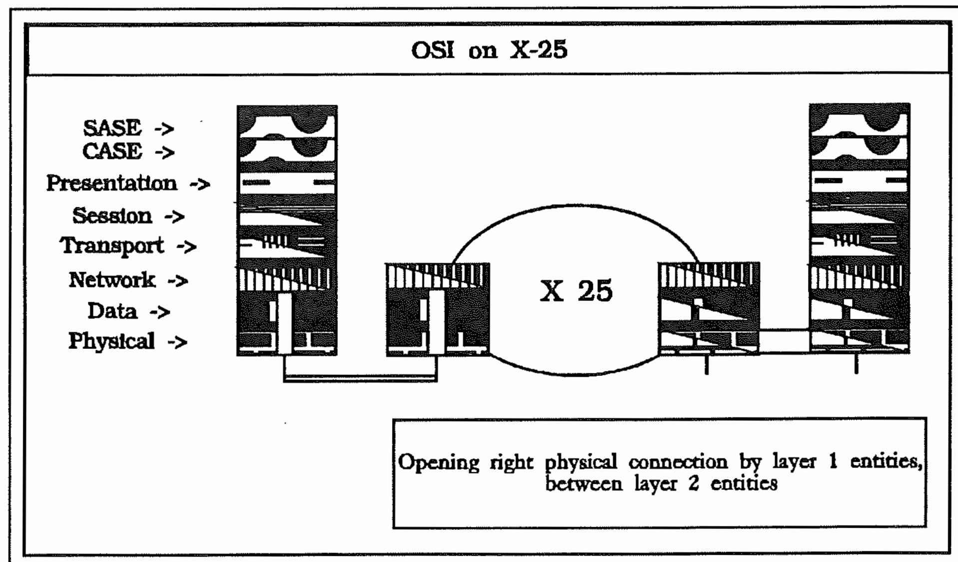


Figure 11.7 Opening right physical connection

The connections are opened normally as described above until the simulation reaches layer three, where the four layer three entities are opened simultaneously,

creating also the white bar linking each entry node stack to the X-25 network (see figure 11.8).

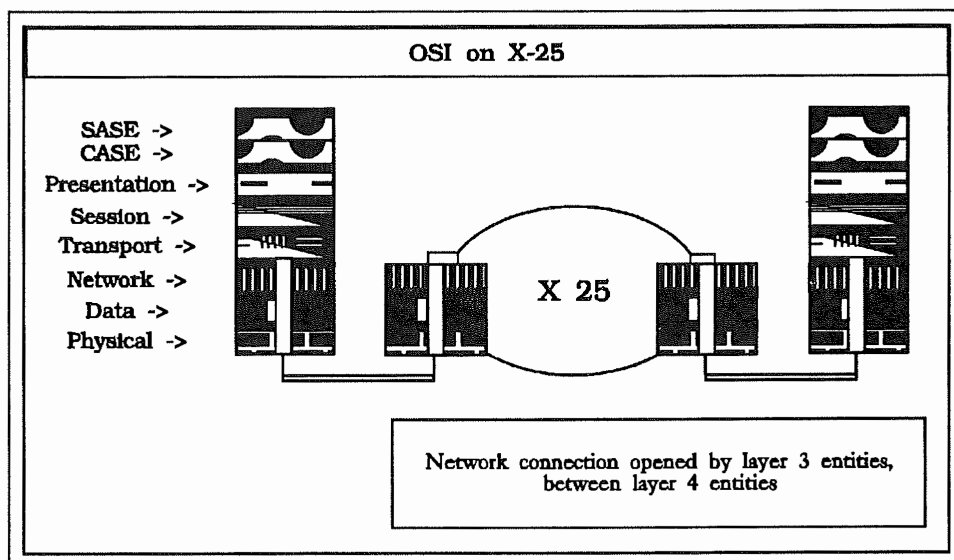


Figure 11.8 Network connection created

The simulation goes on until the final state is reached, that is the entire connection is opened from the layer 5 entities (see figure 11.9). **(Example 11.2)**

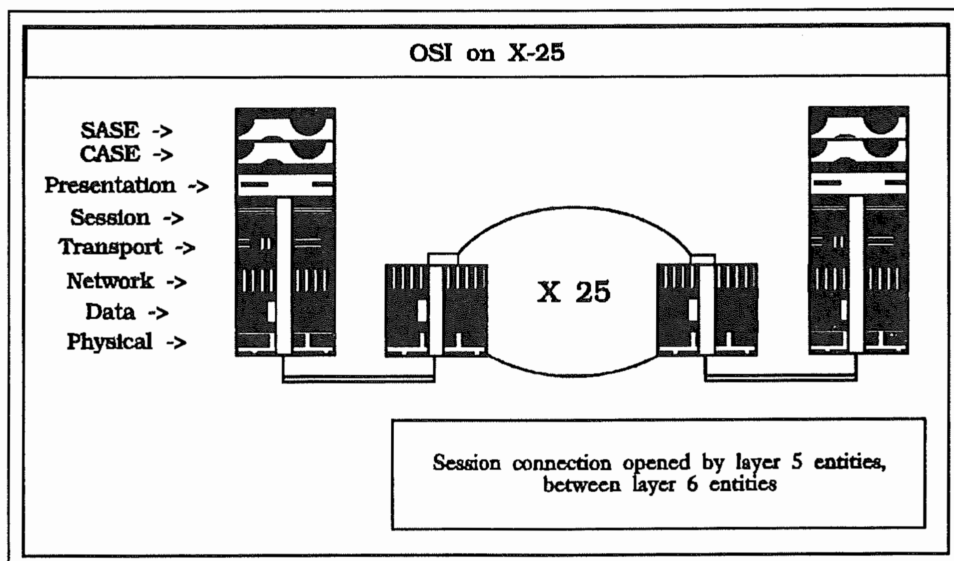


Figure 11.9 Connection completely opened for layer 5 entities

If it is impossible to open any connections when the user requests it -because a connection is opened using each layer of each stack-, the following warning message appears in red in the message box :

Nothing is disconnected, connection impossible

The third case listed above concerns the closing of a connection or disconnection. If it is possible for the user to close an existing connection, a message appears in the message box :

Select the source layer

At this stage, the user has to select the layer from which he wants the disconnection to be initiated.
If the selected layer is a layer from which a disconnection can not be initiated, a message appears in the message box :

These layers can not be disconnected

When the layer is selected and when the disconnection is possible, a rectangle with a red outline appears between it and its counterpart in the other machine stack or entry node stack[3]. At the same time they start to flicker, all this to indicate that the disconnection is in progress. A message is also displayed in the message box :

Closing <layer name> connection by layer <layer number N> entities, between layer <layer number N+1> entities

where '<layer name>' is the name of the layer currently involved in the disconnection; where '<layer number N>' is the number of the layer currently involved in the disconnection; where '<layer number N+1>' is the number of the upper layer for which the disconnection is made at the current level.
This goes on until layer 5 is reached, if the source layer entity is above layer 4 entities. At this moment, the layer entities above layer 4 entities which are concerned by the disconnection, change their colors back to pastel shade, the white bar representing the connection is erased and the string :

<layer name> connection closed by layer <layer number N> entities, between layer <layer number N+1> entities

is displayed in the message box, where '<layer name>' is the name of the layer currently disconnected; where '<layer number N>' is the number of the layer currently disconnected; where '<layer number N+1>' is the number of the upper layer for which the disconnection is made at the current level.
When the concerned layer entity is underneath layer 5, a confirmation is asked to the user by the intermediary of a message :

Disconnection of <layer name> connection by layer <layer number N> entities, between layer <layer number N+1> entities ?

[3] : This depends on the chosen layer

appearing in the message box, where '<layer name>' is the name of the layer currently involved in the disconnection; where '<layer number N>' is the number of the layer currently involved in the disconnection; where '<layer number N+1>' is the number of the upper layer for which the disconnection is made at the current level.
Note that if layer three is concerned by the disconnection, the white bar between the entry node and the X-25 ellipse is erased. Note also that if layer one is concerned by the disconnection, the bar between the stack and the entry node is erased.
For example, if one wants to disconnect the connection made in example 11.1 and 11.2, one has first to choose the source layer. Once this is done, the layer five entities begin to flicker and a rectangle with a red outline appears between them. The string :

Closing session connection by layer 5 entities, between layer 6 entities

is displayed in the message box (see figure 11.10).

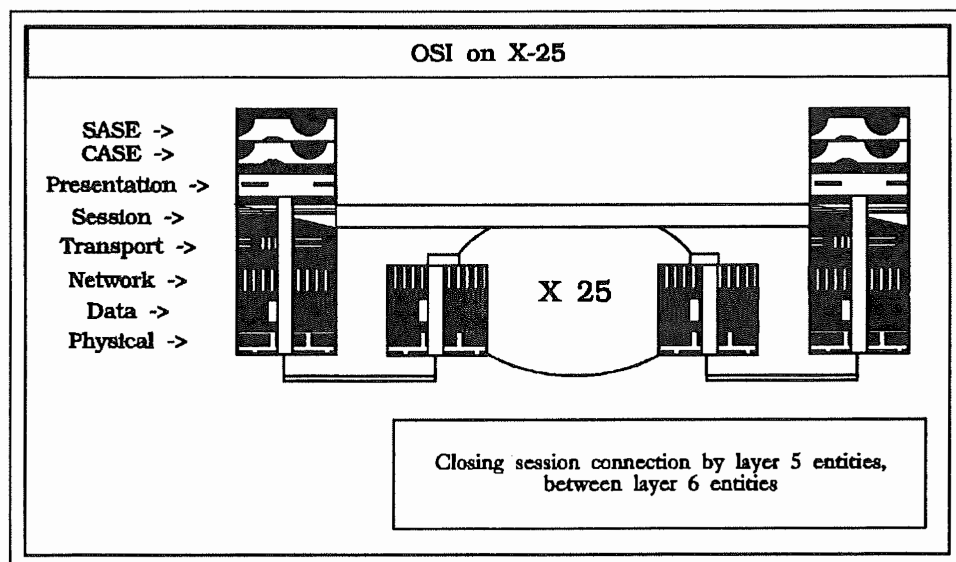


Figure 11.10 Disconnecting from layer five entities

The connection is then closed for the layer 5 entities, reverting their colors to pastel shade and erasing the white rectangle in their middle. A message is also printed inside the message box :

Session connection closed by layer 5 entities, between layer 6 entities

In the next step, the user is prompted with a question displayed in the message box (see figure 11.11) :

Disconnection of transport connection by layer 4

entities, between layer 5 entities ?

If the user replies affirmatively, layer four entities are disconnected like explained above. Otherwise the disconnection is stopped at this stage.

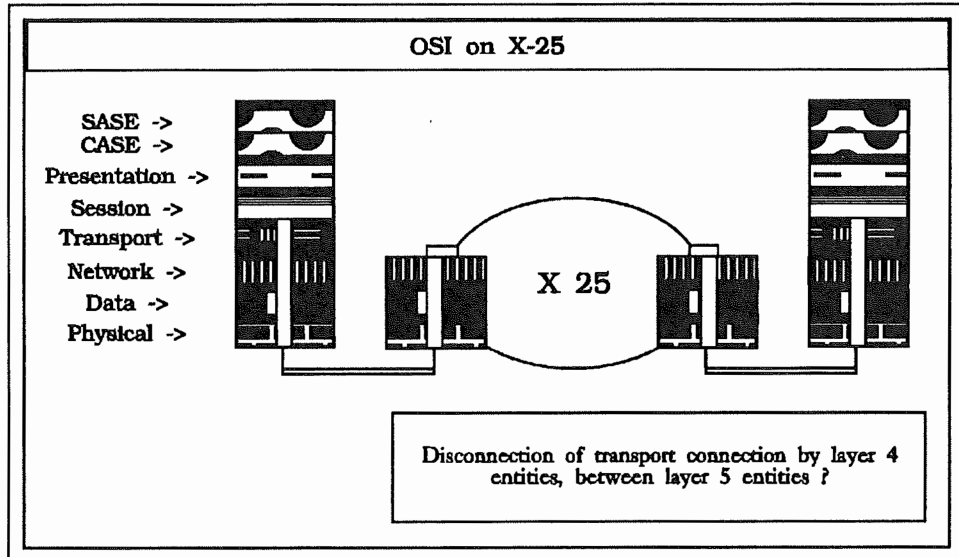


Figure 11.11 **The user is prompted with a question concerning the disconnection of layer 4**

At this stage, the disconnection keeps on, depending on the user's wish. If everything has to be disconnected, the final result is that each layer's color is reverted to pastel shade; the white bars linking the machine stacks to their entry node stacks and the white bars linking the entry node stacks to the X-25 network are erased, so as the white bars in the middle of each layer. (**Example 11.3**)

If no connection is opened when the user requests a disconnection, a warning message appears in red in the message box :

Nothing is connected, disconnection impossible

and nothing happens further.

11.3. The implementation

In this point, we touch the problem of the application's implementation in RMG, based on what has been said so far through the scenario.

11.3.1. *User-application interaction*

We omitted in the preceding part to speak of the user-application interaction. We speak of the user selecting layers or asking for a connection, etc, but we do not tell how this is represented within the application. To specify this, we base

ourselves on what can be seen in several RMG applications, that is pop-up menus, buttons, icons, etc.

We can distinguish **4 different interactions** :

- selecting a connection opening or closing;
- selecting the layers;
- passing from one step to the other;
- replying questions.

For the first interaction, we choose to create a pop-up menu for the application. This menu contains at least two new menu items, concerning a connection opening or closing :

- '**Connection >**';
- '**Disconnection**'.

The first menu item enables the user to make the application simulate a connection opening. It hides a submenu which enables the user to select one mode between :

- **continuous**;
- **step by step**;

In the continuous mode, no interaction between the user and the application is required during the simulation. To the contrary, in the step by step mode an interaction between the user and the application is required in order to jump from one step to the other in the simulation.
The second menu item, 'Disconnection' enables the user to make the application simulate a disconnection in step by step mode only.
For these functionalities the solution of the menu is chosen so that the user can constantly access them and so that the screen representation of these functionalities do not hamper the user perception of the simulation by being constantly present on the screen.

The second interaction concerns the selection of the different layers involved in a connection or a disconnection. This is done by the user clicking directly on the desired layer when the message asking him to do so appears on the screen[4].

The third interaction involves the fact that the user has to ask the simulation to 'execute' the next step. This concerns of course the step by step mode discussed above. This is done using a button labeled '**Continue**' which appears in the application's window whenever it is necessary. The user has only to click on this button in order to make the application 'execute' the next step of the simulation.

[4] : Recall that this message is: '**Select source layer**' or '**Select destination layer**'

The fourth and last interaction involves replies to be given to the application concerning questions of the type :

Disconnection of transport connection by layer 4 entities, between layer 5 entities ?

during a disconnection. These questions require only negative or positive replies which are represented on the screen by two buttons, on which the user has to click, displayed only when a reply is needed and respectively labeled '**Yes**' and '**No**'.

11.3.2. *The application's classes*

Now that we have our scenario; now that the interactions are determined, we must decide which are the classes composing our application.

Several classes are designed in order to represent the different screen objects of the application. A class is created for the stacks, '**OsiStack**', enabling them to be at instantiation seven layer stacks or three layer stacks.

A class is designed to represent the layers, '**Layer**'; so each layer contained in a stack is the instantiation of this class and thus an independent object. Each interface between the layers is the instantiation of a class : '**Interface**'.

The white bar representing the connection and which is situated in the middle of a layer when a connection is opened, is an instance of a class '**Pipe**'.

Of course we find also the master-class, '**Osi1**', which manages the whole application and the menu class, '**Osi1MTree**', which enables the creation of the application's menu.

The hierarchy of these classes is represented at figure 11.12, where classes are represented underneath there superclass.

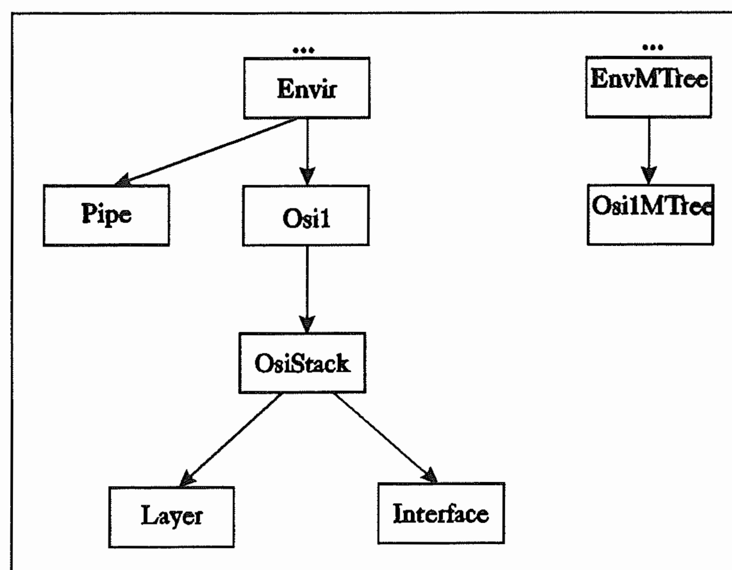


Figure 11.12 Hierarchy of the classes composing the 'OSI on X-25' application

We decided to create so many classes for this application in order of being able to reuse easily these classes in the future applications that are to come in Namur's COLOS team.

11.4. The 'Osi1' class

We are going to look at the 'Osi1' class which is the master-class of this application.

This part first presents the different interesting instance variables. It is followed by the presentation of the factory methods and the different instance methods. Next comes the description of each action defined in this application. Finally we detail briefly the new classes we used in our class.

11.4.1. *Instance variables*

This class is composed of several instance variables, among which some of them are more interesting than others.

A first one is an id instance variable: '**topBanner**'. This variable is designed to contain the address of the title bar placed in the application's window and containing the string : 'OSI on X-25'.

'**stackColl**' is an id instance variable which is designed to contain the address of the collection[5] containing the addresses of every stack appearing in the application; each of these stacks being an instance of the 'OsiStack' class.

'**sourceStack**' and '**destStack**' are two id instance variables which are designed to contain the addresses of the stacks which are involved in the opening or closing of a connection.

Three other id instance variables are designed to contain addresses of other screen objects appearing in the application's window. '**textWin**' contains the address of the message box; '**x25Net**' contains the address of the ellipse figuring the X-25 network; finally, '**x25Str**' contains the address of the string appearing inside this ellipse.

The three last instance variables we talk about here are also id variables and are designed to contain the addresses of the buttons which enable the user to jump from one step to another in the simulation or reply to a question -see point 11.3. These variables are :

- '**stepButton**' which contains the address of the 'Continue' button;
- '**idButtonOk**' which contains the address of the 'Yes' button;
- '**idButtonNot**' which contains the address of the 'No' button.

---

[5] : Recall that a collection is a sort of indexed variable which in this case contains addresses. For further detail, the reader will consult [STEPSTONE, 1]

11.4.2. *Factory methods*

We find two factory methods in this class: '**newIn:**(id)' and '**delayInit**'.

'newIn:' is the factory method designed to create the application's window and all its subviews. Within this method are done several things, mainly the :

- creation of the window;
- creation of the window's subviews;
- objectivation of C functions to be linked to views as actions;
- initialization of variables;

The creation of the window's subviews concerns the creation of :

- the title bar;
- the X-25 ellipse and string;
- the various layer names situated on the left-hand side of the application's window;
- the two machine stacks;
- the two entry node stacks;
- the message box.

The objectivation of the C functions mainly includes the objectivation of functions to be linked to the application's window itself.

'delayInit' is again used to instantiate the application's menu.

11.4.3. *Instance methods*

In this point we analyze the instance methods of the Osi1 class, one after the other.

The first one is '**extraNewIn:**(id)' which in this case gets the address of the application's menu and creates the Ghost instance in order to enable the user to iconize the application. It also issues the message '[self showAll]' to the class' instance in order to display on the screen all the created views.

'**iconify**' is the method which calls the method 'iconize' in Osi1's superclass, in order to iconize the application.

'**startDiscStep**' is an instance method which sets the application into the active collection if a simulation is not currently performed and if a connection is already established. The application is set into the active collection in order to enable the user to select the source layer from which a disconnection is initiated; it is done by calling the method 'startRun'. It is necessary to use the active collection at this stage, in order to enable the RMG cursor's location and the mouse buttons state to be checked permanently until the user clicks on one of the layers. This method is 'called' from the 'Osi1Mtree' instance when the appropriate menu item is selected.

If a disconnection can not be initiated because no connection is established, a warning message -'Nothing connected, Disconnection impossible'- is displayed into the message box.

'startCon' is a method which sets the application into the active collection if a simulation is not currently performed and if a connection is possible. The application is set into the active collection in order to enable the user to select the source and destination layers from which a disconnection is respectively initiated and responded to; it is also done by calling the method 'startRun'. This method concerns the continuous mode and is directly 'called' from the application's menu instance when the appropriate menu item is selected. If no connection is possible a warning message -'Nothing is disconnected, Connection impossible'- is displayed into the message box.

'startConStep' is very like the 'startCon' method, except that it concerns the step by step mode. It is also directly 'called' from the application's menu instance when the appropriate menu item is selected.

'startRun' displays the string 'Choose the source LAYER' into the message box and installs the application into the active collection. **'stopRun'** is the method that removes the application from the active collection.

'update' is the method 'executed' when the application is in the active collection. In this case, it sends a message to the instance of the Osi1 class containing one of the selectors :

- 'selectOne' in the case of a disconnection;
- 'selectTwo' in the case of a connection;
- 'selectChoiceD' in the case of a reply to a question involving the 'Yes' or 'No' buttons.

'selectTwo' enables the user to select the source and destination layers for a connection opening. This is done by controlling the mouse buttons state. If the left mouse button is depressed, each instance of the Layer class is tested in order to see if the location of the RMG cursor at the 'depression moment' is contained within the bounds of one of them. If this is not the case nothing happens. If this is the case and that :

- only one layer is selected, the message 'Choose the Destination LAYER' is displayed in the message box and the user has to select the second layer;
- two layers are already selected, the application is retrieved from the active collection using the 'stopRun' method and the 'connect' method is 'called' next.

'selectOne' enables the user to select the source layer for the closing of a connection. This method follows the same principles as for the 'selectTwo' method.

'selectChoiceD' is the method enabling the user to click on one of the two buttons -'Yes' or 'No'- in order to reply to the questions asked when a disconnection is performed -see point 11.3. Note that when the disconnection

reaches layer three entities and the lower ones, the question is asked for the connection of each machine stack with its entry node stack. This method follows the same principles as for 'selectOne' or 'selectTwo' except that no layers are tested in this case but only the two buttons present on the screen. If the 'Yes' button is clicked on the simulation is continued normally. If the 'No' button is clicked on the disconnection is stopped at the stage of the concerned layers.

'**connect**' tests if the connection is possible between the two chosen layers and if it is possible to initiate a connection from this layer level[6]. If a connection is possible and that the step by step mode has been requested, the 'Continue' button is created, displayed on the screen and the simulation is launched by sending the source stack[7] a message containing the 'openConStep' selector. If a connection is possible and that the continuous mode has been requested, the source stack is placed into the active collection in order to enable the simulation to be carried on without any user-application interactions. If the connection is impossible, a warning message -'These layers can not be connected'- is displayed into the message box.

'**disconnect**' tests if the disconnection is possible from the selected layer. If the disconnection can be performed, the 'Continue' button is created, displayed on the screen and the simulation is launched by sending the source stack a message containing the 'openDiscStep' selector. If the disconnection is impossible, a warning message -'These layers can not be disconnected'- is displayed into the message box.

'**buttonFace: color1: color2:**' is a method[8] that enables to create shaded buttons on the screen, as seen in figure 9.2. This method takes three arguments. The first one is the address of the view which has to be converted with the method. The second and third ones are the two colors which will be the border of the new button.

11.4.4. *The actions*

We analyze briefly the different objectified C functions one can find inside this class.

'**actionSelf**' is the action attached to the application's window itself and is the objectivation of the '**ac_Self(aView)**' C function. Its goal is only to make the application's window and all its subviews as the first view of the hierarchy of views present on the screen; this is done using the RMGView method : '**popToTop**'.

'**superAction**' is attached to the outline around a button -created with 'buttonFace: color1: color2:'. This action represents the '**superAct(aView)**' C

6 : See point 11.2 for the restrictions concerning the opening of a connection

7 : The source stack is the stack which contains the layer entity from which the connection must be initiated. Its address is contained in the 'sourceStack' instance method

8 : This method was designed by Uwe Heimburger and Detlev Wegener at the Kiel's University for their work in the COLOS project and is widely used nowadays by COLOS programmers

function which executes the action linked to the superview of the view which address is passed as argument to the function. This is done so that the button's action is also executed when the user clicks on the colored surround of the button itself.

'**actionStep**' is the action attached to the 'Continue' button. This actions can be the objectivation of either '**ac_StepC(aView)**' in the case of a connection opening, either '**ac_StepD(aView)**' in the case of a connection 'closing'. The first of these C functions enables the user to pass to the following step of the connection simulation. The second of these C functions enables the user to pass to the following step of the disconnection simulation. In this case, if the concerned layer is inferior to five, it displays also in the message box a question of the type : 'Disconnection of transport connection by layer 4 entities, between layer 5 entities ?', creates the 'Yes' and 'No' buttons and sets the application into the active collection in order to enable the user to reply to the question by clicking on one of the buttons.

11.4.5. *The classes used*

Classes which have not yet been presented have been used in this class :

- '**OrdCltn**';
- '**RMGEllipse**'.

The first of these classes is an Objective-C class. It enables **the management of an arbitrary number of objects as a whole**, these objects being maintained in the order in which they were added to the collection.

We use two different factory methods :

- '**with:**(int)';
- '**new**'.

The first is a factory method inherited from its superclass : '**Cltn**'. It enables the creation of an instance of OrdCltn composed of values. The number of values is passed as argument. It is thus used as :

```
[OrdCltn with: 8, "Physical ->", "Data ->", "Network ->",
"Transport ->", "Session ->", "Presentation ->", "CASE -
>", "SASE ->"];
```

The second factory method is also inherited from its superclass : 'Cltn'. It enables the creation of an empty instance of OrdCltn.

We also use two instance methods :

- '**add:**(id)';
- '**at:**(int)'.

The first one enables the user to add an object which address is passed as argument, at the end of the already existing collection. For example :

```
[aCollection add: anAddress];
```

where 'aCollection' contains the address of an instance of OrdCltn and 'anAddress' contains the address of any instance.
The second one returns the object's address situated in the collection at the location passed as argument -an integer. For example :

```
anAddress = [aCollection at: 0];
```

where the address located in the first slot of the collection is placed into the 'anAddress' variable.
Note that for this class, many other methods are available enabling the user to cope with the different values stored into the collection.

The second class -'RMGEllipse'- is an RMG class which enables the user to **draw an elliptical shape** inside a view. This class is a subclass of RMGView and so inheriting all its methods and instance variables.
This class defines some instance variables mainly in order to contain various arguments concerning the ellipse itself. This is the case of '**xRadius**' and '**yRadius**' for example, which are of type integer and are designed to contain respectively the horizontal and vertical radius of the ellipse.
To instantiate this class, the user can utilize RMGEllipse '**new**' factory method which creates an instance with default parameters. But the user can also utilize RMGView factory methods. We did just that and used the 'relative:: extent:: superview: bkgd:' factory method.
This class provides also many instance methods enabling the user to fix various parameters concerning the ellipse. Among these we can find '**xRadius:** (int) **yRadius:** (int)' which sets the half width and the half height of the ellipse to the values passed as arguments.
'**origin:** (int)**:** (int)' sets the center of the ellipse to the location identified by the two values passed as arguments. '**circleSolid:** (BOOL)' enables the user to specify if the ellipse has to be drawn as a solid or as an outline only.
We used this class in the 'newIn:' method, to create the X-25 network representation :

```
x25Net = [RMGEllipse relative: 550: 50          /* 1 */
                     extent: 400: 500
                     superview: self
                     bkgd: BLUE];

[[[[[x25Net     xRadius: 190 yRadius: 110]  /* 2 */
                origin: 150: 210]           /* 3 */
                circleSolid: TRUE]          /* 4 */
                show];
```

Line 1 creates an instance of RMGEllipse using RMGView's factory method. This creates a view in which the ellipse is placed.
Line 2 sets the dimensions of the ellipse, while line 3 sets its center at the location (150,210).
Line 4 specifies that the ellipse has to be drawn as a solid.

11.5. The 'Osi1MTree' class

We look at the 'Osi1MTree' class, subclass of 'EnvMTree', which is the menu class of this application. This class implements only two classical factory methods : **'delayInit'** and **'getIt'**. It does not have any proper instance variables, nor any instance methods.

'delayInit' is the factory method which initializes the menu itself. It is a menu containing four different menu items :

- **'!! Quit !!'**;
- **'Icon'**;
- **'Connection >'**;
- **'Disconnection'**.

The first menu item enables the user to quit the application, while the second one enables him to reduce the application's window to an icon.
The third menu item enables the user to start the simulation of a connection opening. It hides a submenu which contains the two menu items **'Step by step'** and **'Continuous'** enabling the user to start a connection opening respectively in step by step mode -by the instance of Osi1MTree sending a message to the instance of the Osi1 class containing the 'startConStep' selector- and in continuous mode -by the instance of Osi1MTree sending a message to the instance of the Osi1 class containing the 'startCon' selector.
The fourth menu item enables the user to start a disconnection in step by step mode only. This is done by the instance of Osi1MTree sending a message to the instance of the Osi1 class containing the 'startDiscStep' selector.

'getIt' is simply a factory method that returns the address of the menu previously created.

11.6. The 'OsiStack' class

We look at the 'OsiStack' class, subclass of Osi1, which is the class implementing methods and declaring instance variables characterizing each stack of the application.

This part first presents the different interesting instance variables. It is followed by the presentation of the factory methods and the different instance methods. Next

comes the description of each action defined in this application. Finally, we present briefly a new class we used in 'OsiStack'.

11.6.1. *Instance variables*

This class has several instance variables, among which we can find '**clColorArray**' and '**opColorArray**' which are two id instance variables designed to contain the addresses of two instances of the Objective-C class '**IntArray**'[9]. These arrays contain the numbers identifying the different pastel colors of the layers in the case of clColorArray and the numbers identifying the different brilliant colors of the layers in the case of opColorArray.

'**layerColl**' and '**interfaceColl**' are two id instance variables designed to contain the addresses of instances of the 'OrdCltn' class. These ordered collections contain the addresses of each layer[10] composing a stack in the case of layerColl and the addresses of each interface[11] included in a stack in the case of interfaceColl.

'**nameColl**' is an id instance variable designed to contain the address of an instance of the 'OrdCltn' class, this instance containing the various names of the layers to be displayed at the left-hand side of the application's window.

'**destiStack**' is an id instance variable which contains the address of the stack with which a connection opening or a disconnection has to be made. It has a value only after the user asks for a connection opening or a disconnection and selects the layers accordingly -as explained above.

'**entryNode**' is an id instance variable which contains the address of a machine stack entry node stack.

'**aLink**', '**ellLink**' and '**phyLine**' are three id instance variables which contain the addresses of respectively :

- the rectangle with a red outline which is created between the two stacks involved in a connection opening or closing;
- the link between an entry node stack and the X-25 ellipse;
- the link between a machine stack and its entry node stack.

Various instance variables are declared to be used internally for 'private' use, like :

- '**work**' which is a boolean instance variable denoting the fact that a simulation is currently going on or not;
- '**actionType**' which is an integer instance variable identifying the type of simulation going on : connection opening or disconnection;

---

[9] : This class is briefly explained in point 11.6.4

[10] : Thus instance of the 'Layer' class

[11] : Thus instance of the 'Interface' class

- '**skip**' and '**back**' which are respectively a boolean and an id instance variable used in order to enable the opening or closing of a connection between a machine stack and its entry node stack, on both sides;
- '**winText1**' and '**winText2**' which are instance variables containing the addresses of the two subwindows composing the message box[12];
- '**curLayer**' which is an id instance variable containing the address of the layer currently concerned with the connection opening or closing, so the layer where something is currently 'happening';
- '**workLayer**' which is an id instance variable containing the address of the layer from which the connection opening or closing is initiated.

Finally, '**stackType**' is an integer instance variable which contains the number of layers which characterizes a stack. For example, the value of stackType is 7 in the case of a machine stack.

11.6.2. *Factory methods*

This class has two factory methods :

- '**new:** (id) **at:** (int): (int) **type:** (int) **strings:** (id)';
- '**new:** (id) **at:** (int): (int) **type:** (int) **multiplex:** (int): (int)'.

The first factory method enables the user to create an instance of OsiStack in which some strings are displayed. This is the case with the different layer names appearing on the left-hand side of the application's window.
This stack is placed in the view which address is passed as first argument, at the location identified by the two integers passed as second and third arguments. The fourth argument is the type of stack needed, thus it specifies the number of layers desired. Finally, the last argument is the address of the collection which contains the strings to display inside this stack.

The second factory method enables the user to create an ordinary instance of OsiStack, like the machine or entry node stacks. This stack is placed in the view which address is passed as first argument, at the location identified by the two integers passed as second and third arguments. The fourth argument is the type of stack needed, thus it specifies the number of layers desired. The two last arguments are not currently used but are designed to pass as argument the layer numbers between which multiplexing is possible in order to make some changes to the layers appear on the screen.

11.6.3. *Instance methods*

We review now each instance method implemented in the OsiStack class.

---

[12] : Indeed the message box has to be separated in two parts as the messages to display are very long and in order to limit the length of the message box itself. This enables the displaying of two strings, one on top of the other

'**loadColor**' is an instance method which 'builds' the instances of the 'IntArray' class with the different layer colors. The addresses of these instances are put into the 'clColorArray' and 'opColorArray' instance variables.

'**loadLayerNames**' is an instance method which 'builds' the instance of the 'OrdCltn' class with the different layer names. The address of this instance is put into the 'nameColl' instance variable.

'**(int)layerContains:** (int): (int)' is a method which checks if the location identified by the two values passed as arguments is inside the bounds of one of the instances of the Layer class composing the stack. This method returns an integer which has the value zero if the location is not situated inside any of the stack layers. In case the location is situated inside one of the stack's layers, the number[13] of this layer is returned.

'**update**' is the method 'executed' when the instance of the OsiStack class is in the active collection[14]. In this case it sends a message to itself containing the selector 'openConnect' in the case of a connection opening. The type of simulation to be performed in continuous mode is known by the 'actionType' instance variable. This system is adopted in case the active collection is required for other purposes.

'**openConnect**' is the instance method which starts the simulation of a connection opening in continuous mode. It is separated into two phases.
The first phase concerns the downward progression of the connection opening. That is creating the rectangle with a red outline between the two stacks involved in the connection, if it is not yet created; otherwise, displaying it at the level of the current layers[15] concerned with the connection opening. It makes also the current layers flicker and displays a message of type: 'Opening Transport connection by layer 4 entities, between layer 5 entities', in the message box.
The second phase concerns the upward progression of the connection opening, when the rectangle with a red outline has reached the lowest layer of the stack or the highest layer already connected. It suppresses the rectangle with a red outline linking the two stacks involved in the connection, stops the flickering of the current layers and changes their color to brilliant shade. It displays also a message of type: 'Physical connection opened by layer 1 entities, between layer 2 entities', in the message box and creates the white bar which is situated in the middle of each layer where the connection is already opened. It creates also the white bars between the machine stack and its entry node stack, if the current layers are the Physical layers. It creates the white bar between the entry node stack and the X-25 network, if the current layers are the Network layers.
Note that the message containing the 'openConnect' selector is only sent to the source stack, thus the stack from which the connection is initiated.

'**openConStep**' is the method which starts the simulation of a connection opening in step by step mode. Its functionalities are the same than for

---

[13] : By number we mean the level of the layer inside the ISO reference model. For example 7 is the number of the Application layer

[14] : An instance of the OsiStack class is put into the active collection by the instance of the Osi1 class

[15] : By current layers, we mean the layers where something is currently happening in the simulation

'openConnect', except that this method is 'executed' independently from the active collection. As for the 'openConnect' method, the message containing the 'openConStep' selector is only sent to the source stack, thus the stack from which the connection is initiated. This method returns an integer which identifies the current status of the connection simulation.

'replyConnect' is a method which makes an instance of the Layer class flicker or change its color, depending on the case. A message containing the 'replyConnect' selector is sent by the source stack of the connection opening to the destination stack of the connection opening. This method returns the address of the concerned instance of the Layer class.

'openDiscStep' is the method which starts the simulation of a disconnection. If the current layers are superior or equal to 5, the message '[self discStepUp]' is sent by the instance of OsiStack to itself. If the current layers are inferior to 5, the message '[self discStepLowOk]' is sent by the instance of OsiStack to itself. This process is separated in two, as the three top layers of the ISO reference model do not need any confirmation to be disconnected.

'discStepUp' is the method which simulates the disconnection of the three top layers of the ISO reference model.
The first phase concerns the downward progression of the disconnection. That is creating the rectangle with a red outline between the two stacks involved in the connection, if it is not yet created; otherwise, displaying it at the level of the current layers concerned with the disconnection. It makes also the current layers flicker and displays a message of type: 'Closing Transport connection by layer 4 entities, between layer 5 entities', in the message box.
The second phase concerns the upward progression of the disconnection, when the rectangle with a red outline has reached the layer 5. It suppresses the rectangle with a red outline linking the two stacks involved in the disconnection, stops the flickering of the current layers and changes their color to pastel color. It displays also a message of type: 'Physical connection closed by layer 1 entities, between layer 2 entities', in the message box and suppresses the white bar which is situated in the middle of each layer where the connection is opened. It suppresses also the white bar between the machine stack and its entry node stack, if the current layers are the Physical layers. It suppresses the white bar between the entry node stack and the X-25 network, if the current layers are the Network layers.
This method returns an integer which identifies the current status of the disconnection simulation. Note that a message containing the selector 'discStepUp' is only sent to the source stack.

'discStepLowOk' is the method which simulates the disconnection of the four lower layers of the ISO reference model. This is made by taking in count the choices of the user himself.
The user replies to the question of type: 'Disconnection of transport connection by layer 4 entities, between layer 5 entities ?' -see point 11.3. If the reply is positive, the current layers are disconnected by changing their color to pastel shade, suppressing the white bar which is situated in the middle of each layer where the connection is opened and displaying in the message box the message of type:

'Physical connection closed by layer 1 entities, between layer 2 entities'. Possibly, it suppresses also the white bars between the machine stack and its entry node stack, if the current layers are the Physical layers; it suppresses the white bar between the entry node stack and the X-25 network, if the current layers are the Network layers.
This method returns also an integer which identifies the current status of the disconnection simulation. Note that a message containing the selector 'discStepLowOk' is only sent to the source stack.

'**replyDiscStepLowOk**' is a method which makes an instance of the Layer class change its color, following the situation. A message containing the 'replyDiscStepLowOk' selector is sent by the source stack of the disconnection process to the destination stack of the disconnection process only if the four lower layers are concerned. It returns an integer which identifies the current status of the disconnection's simulation.

'**replyDisconnect**' is a method which makes an instance of the Layer class flicker or change its color, following the situation. A message containing the 'replyDisconnect' selector is sent by the source stack of the disconnection process to the destination stack of the disconnection process only in the case of the three upper layers of the ISO reference model. It returns the address of the concerned instance of the Layer class.

In this class are also implemented series of methods which goal is to store a value into certain instance variables :

- '**sEntryNode:** (id)' which stores the address of the entry node passed as argument into the 'entryNode' instance variable;
- '**setCurLayer:** (int)' which stores the number passed as argument into the 'currentLayer' instance variable; this represents the initiating layer number ;
- '**setWorkLayer:** (int)' which stores the number passed as argument, as the current layer number into the 'workLayer' instance variable;
- '**setSkip:** (BOOL)' which stores the boolean passed as argument into the 'skip' instance variable;
- '**back:** (id)' which stores the address passed as argument into the 'back' instance variable;
- '**setDestiStack:** (id)' which stores the address of the destination stack into the 'destiStack' instance variable;
- '**setTextWin:** (id): (id)' which stores the addresses of the two subwindows composing the message box into the 'textWin1' and 'textWin2' instance variables;
- '**setAction:** (int)' which stores the integer passed as argument, representing the type of simulation to be performed, into the 'actionType' instance variable;
- '**setWorking:** (BOOL)' which stores the boolean passed as argument into the 'work' instance variable.

In this class are also implemented some methods which goal is to return the value of certain instance variables :

- '**askEntryNode**' which returns the address of the entry node stored into the 'entryNode' instance variable;
- '**askCurLayer**' which returns an integer which is the number of the initiating layer which is stored into the 'currentLayer' instance variable;
- '**askLayerColl**' which returns the address of the collection containing the addresses of each instance of the Layer class composing the stack; this address is stored into the layerColl instance variable;
- '**working**' which returns the boolean stored into the 'work' instance variable, specifying if the stack is currently involved in a simulation or not;
- '**askSkip**' which returns the boolean stored into the 'skip' instance variable;
- '**askStackType**' which returns an integer which is the stackType instance variable;
- '**askIdNameColl**' which returns the address of the collection containing the names of the different layers, contained into the 'nameColl' instance variable;
- '**askDestiStack**' which returns the address of the destination stack of the connection opening or closing, stored into the 'destiStack' instance variable.

'**testConnect**' is an instance method which tests if a connection is already established at the current layer. It returns a boolean which is TRUE if the connection is already established and FALSE otherwise.

'**testConUp**' is an instance method which tests if a connection is established at any level. It returns a boolean which is TRUE if a connection is already established and FALSE otherwise.

'**searchWidthLink:** (id)' is a method which calculates the width of the rectangle with a red outline appearing between two stacks at a connection opening or closing. This width is calculated between the stack to which the message is sent and the stack which address is passed as argument. The width is returned as an integer.

'**searchXOriginLink:** (id)' is a method which calculates the X coordinate where the rectangle with a red outline has to be located. This coordinate is the right edge of the leftmost stack involved in a connection opening or closing. The coordinate is returned as an integer.

'**searchYOriginLink:** (id)' is the same method as 'searchXOriginLink:' except that it returns the Y coordinate.

'**connectionDone**' is a method which tests if a connection is opened using every layer in the stack, thus preventing an other connection opening. The returned value is a boolean which is TRUE if all the layers are involved in a connection, FALSE otherwise.

'**deConnectionDone**' is the same method as 'connectionDone' except that it checks if each layer is disconnected, thus suppressing the possibility of another disconnection.

'**createLink:** (int): (int) **length:** (int): (int)' is the method that creates a white bar between an entry node stack and the X-25 ellipse or between a machine stack and an entry node stack. This bar is placed at the location identified by the two first arguments; its width is specified by the third argument and its height by the fourth one. It returns the address of the created bar.

'**freeEllLink**' hides and frees the link between an entry node stack and the X-25 ellipse.

'**freePhyLine**' hides and frees the link between a machine stack and an entry node stack.

'**flickering**' is a method which makes the current layer flicker and sets various parameters concerning this instance of the Layer class -see point 11.7.

11.6.4. *The classes used*

A class has been used in this class which has not yet been presented : '**IntArray**'.

It is a class which enables to **manage indexed variables of integer type**. We used the '**with:** (int)' factory method to instantiate it. This method is inherited from its superclass: '**Array**'. It enables the creation of an instance of OrdCltn, composed of values which number is passed as argument. For example :

```
anArray = [IntArray with: 4, 1, 2, 3, 4];
```

which creates an instance of the 'IntArray' class, containing the four integers: 1,2,3,4; where 'anArray' is an id variable which contains the address of the newly created instance.

We used also the '**intAt:** (int)' instance method in order to access the values stored into the 'IntArray' instance. This method returns the value which is stored at the location passed as argument. For example :

```
anInteger = [anArray intAt: 0];
```

which stores the value situated at the position 0 in the array identified by 'anArray' into the integer variable 'anInteger'; so in this case the value 1.

11.7. The 'Layer' class

We look at the 'Layer' class, subclass of OsiStack, which is the class implementing methods to create and manage the different layers composing a stack.

This part first presents the different interesting instance variables. It is followed by the presentation of the factory methods and the different instance methods.

11.7.1. *Instance variables*

This class declares four different instance variables.

The first of them is '**conWith**' which is an id instance variable designed to contain the address of the other instance of the 'Layer' class with which the layer is connected.

'**aPipe**' is an id instance variable which is designed to contain the address of the white bar which is created in the middle of a layer when this layer is concerned by an opened connection.

'**closeColor**' and '**openColor**' are two integer instance variables. The first one is designed to contain the number identifying the pastel color of the layer, while the second one is designed to contain the number identifying the brilliant color of the layer.

The last instance variable is '**flickering**' which is a boolean variable taking the value TRUE if the layer is currently flickering or FALSE otherwise.

11.7.2. *Factory method*

The only factory method implemented in the 'Layer' class is: '**new:** (id) **at:** (int): (int) **color1:** (int) **color2:** (int) **height:** (int) **width:** (int)'. This method creates the instance of the 'Layer' class in the view which address is passed as first argument; it is created at location identified by the second and third arguments. The two arguments introduced by 'color1:' and 'color2:' are the numbers identifying respectively the pastel color and the brilliant color of the layer, while the two last arguments are the height and the width of the layer.

11.7.3. *Instance methods*

We review each instance method implemented in this class.

'**flicker**' is an instance method which installs the layer into the active collection in order to make it flicker. '**flick**' is an instance method which returns the value of the 'flickering' instance variable.

'**update**' is the method 'executed' when the layer is in the active collection. It hides the layer from the screen if it is currently shown and vice versa, thus producing a flickering of the layer. '**stopRun**' retrieves the layer from the active collection.

'**changeColor**' changes the color of the layer from pastel to brilliant and vice versa depending on the case.

'**connectionState**' is an instance method which tests the state of a layer to see if a connection is opened using this layer or not. It returns a boolean which is TRUE if a connection is opened using this layer, FALSE otherwise.

'**inConnectWith:** (id)' stores into the 'conWith' instance variable, the address passed as argument which is the address of the layer with which the layer is connected. '**inConnectWith**' returns the address contained into the 'conWith' instance variable.

'**createPipe**' is the instance method which creates the white bar in the middle of the layer when this layer is concerned by an opened connection. '**deletePipe**' is the instance method which deletes this white bar.

11.8. The 'Interface' class

The 'Interface' class is a small class, subclass of OsiStack, which manages the creation of an interface between two layers. It was created as a stand-alone class in order to enable future developments of the application.

This class has only one instance variable: '**aPipe**', which is designed to contain the address of the white bar which is created in the middle of the interface when it is concerned by an opened connection. This white bar is also created across the interfaces in order to notify the user that some communication means is installed between the layers separated by the interface.

This class implements one factory method: '**new:** (id) **at:** (int)**:** (int) **color:** (int) **height:** (int) **width:** (int)'. This method enables the creation of an instance of the 'Interface' class in the view which address is passed as first argument and at the location within this view identified by the second and third arguments. It is created in the color passed as fourth argument and has as height and width the two last arguments.

Two instance methods are implemented in this class :

- '**createPipe**';
- '**deletePipe**'.

The first one is the instance method which creates the white bar in the middle of the interface when it is concerned by an opened connection. The second one is the instance method which deletes this white bar.

11.9. The 'Pipe' class

The 'Pipe' class is a small class, subclass of Envir, which manages the creation of the white bar which is created in the middle of an interface or a layer when they are concerned by an opened connection.

This class has no instance variables and implements only one factory method: '**new:** (id) **at:** (int)**:** (int) **color:** (int) **height:** (int) **width:** (int)'. This method enables the creation of an instance of the 'Pipe' class in the view which address is passed as first argument and at the location within this view identified by the second and third arguments. It is created in the color passed as fourth argument and has as height and width the two last arguments.

Two instance methods are implemented in this class :

- '**modifyHeight:** (int)';
- '**modifyHeight2:** (int)'.

The first one is the instance method which modifies the overall height of the pipe and sets it to the integer passed as argument. The location of the pipe is not changed. The second instance method modifies also the overall height of the pipe and sets it to the integer passed as argument, while it changes also the location of the pipe on the Y axis, setting it to the value computed from the difference of the current height of the pipe and the new one passed as argument.

11.10 Critique

We would like here to make a small critique of this application.

This application is not perfect at all, as it is only a first prototype. Many errors and details must be changed. These errors concern the simulation of the OSI standard and also details concerning the progress of the simulation. For example, when the user has to select one of the layer entities in order to specify from which layer entity the connection or disconnection has to be initiated, the message :

Select the source layer

is displayed into the message box. In fact this message should be :

Select the source layer entity

This application has been presented to the COLOS members at a COLOS meeting and several constructive remarks have been made concerning the presentation :

- the source and destination layer entities should not behave in the same way;
- the flickering is tiring;
- the message box should be separated from a 'question box' and a 'warning box';
- a fix-menu would be more agreeable than a pop-up menu;

Other remarks concerned also the functionalities of the application :

- it could be interesting to let the user do some 'mistakes' while using the application, in order to see what happens;
- it could be interesting to illustrate the connection-less 'orientation';

In short, this application is a good start for the Namur COLOS team, but must be improved on many points of view.

11.11. Summary

The goal of this chapter was to present the 'OSI on X-25' application which was created by Namur's COLOS team following the principle of a scenario. It was then programmed in the RMG environment, building new classes and using already existing ones.

A first point presented the principles of a scenario which can be divided into few different steps :

- deciding what the application will do;
- deciding what the application will be composed of on the screen;
- designing the scenario 'by hand'.

The second point applied these principles to our case to build the 'OSI on X-25' scenario. We saw that the goal of this application was to simulate the opening or closing of a connection between two layer entities situated in two different machines, following the principles of the OSI model. All the functionalities of the future application were also stated one by one.
We saw also that the screen composition was to be made of :

- four stacks;
- an ellipse;
- a message box.

where a stack is a rectangle representing a machine or an entry node and the ellipse represents the X-25 network.
We tried afterwards to describe the scenario's evolution.

The third point described the implementation of the scenario. We saw what the user-application interaction was of four kinds :

- selecting a connection opening or closing;
- selecting the layers;
- passing from one step to the other;
- replying questions.

Next we saw which were going to be the classes of the application. We saw that six classes were designed :

- 'Osi1', the master-class;
- 'Osi1MTree', the menu-class;
- 'OsiStack', representing a stack of the application;
- 'Layer', representing a layer within a stack;
- 'Interface' representing the interface between two layers;

- 'Pipe', which instance represents an opened connection and is situated in the middle of a layer under the shape of a white bar.

The following points described these classes, in terms of instance variables, factory methods, instance methods, actions and new RMG classes used.

Finally we made a brief critique of this application.

Chapter 12: Conclusion

The RMG environment is a very powerful tool enabling the creation of highly interactive simulations and programs. The problem is that the lack of books, references and works concerning this environment makes its apprenticeship very difficult and very slow to anyone wanting to develop applications with it. We hope that this thesis constitutes a step towards easier comprehension and mastering of the RMG mechanisms. We do not pretend to impose it as an absolute reference but only as a contribution to the work done by many programmers in the COLOS project.

We tried to give a simple and general approach to Object-Oriented mechanisms, insisting on the Objective-C language, in order to enable the user to get accustomed to the principles and language on which RMG is built. We tried also to give a brief description of RMG, in order to introduce this environment to the reader.

We then tried to describe the basic RMG programming principles and supply the future RMG programmer with a guide which can be of a certain help when he starts programming applications in RMG. This guide is far from being complete and is in fact only an introduction to the very rich world of the RMG environment and of RMG programming. It is only, in our case, the result of our working experience. For clarity and concision purposes, we did not want to introduce and give an explanation of each RMG class; this is why we introduced only the basic ones. Further, we had to give certain informations concerning 'advanced' programming in RMG, though this work is only designed to be a beginner's guide to RMG. We tried also to give an illustration of what can be made in this environment by presenting an application we realized for the COLOS project : the 'Video' application.

We wanted to present the plans of the university of Namur for the COLOS project, concerning the telecommunication field. We presented also one of these plans which became an application: 'OSI on X-25'. This application is of course far from being perfect as it is in fact a first prototype; many things are not yet settled and many errors subsist.

The COLOS project seems to be a very good and useful experience. It enables teachers to dispose of high quality simulation applications which are not intended to replace the teacher or the teacher's course but to help him in his task.

This project regroups a small but international 'board' of programmers and teachers. This enables to develop applications whose creation is the grouped work of both these categories of people, thus joining the pedagogical experience of some of them to the programming skills of the others. Thanks to a nice internal organization of the project each member is aware of the works of every other members. This enables the exchange of advices, of programming tricks, of classes. It enables also a coordination on the user-application interfaces, on the subjects treated by each members, etc.

The environment used in the project -RMG- is a very powerful tool indeed. We think that the only black spot is the difficulty of apprenticeship and the time required to master it. Once this gate has been jumped over and that one masters sufficiently the environment, one is able to construct very powerful applications. But we think

that this environment keeps on being difficult to handle, even for an experienced user.

We think that the use of an Object-Oriented environment for the creation of this type of graphical simulations is a good solution. The object-Oriented approach enables to module the applications as they are really seen on the screen, each screen object really being an individual and independent object in the system. Of course this does not show for the teacher or the learner using the simulation, but it does make a difference for the programmer which is confronted with the creation of these applications.

A problem is that the requirements towards the project are numerous, specially on a graphical point of view. More and more complicated graphics are used, requiring more and more speed. This is in part the explanation of two choices made for the future by the COLOS project in collaboration with the Hewlett-Packard company :

- use of new HP RISC machines;
- use of the X-Window environment together with existing software development kits, to create applications.

The already existing applications will be translated and migrated to the new environment by COLOS programmers. We hope that RMG will keep on being used because we think that apart from the problems it leads to, it keeps being a very powerful environment.

Bibliography

[BOURNE, 82] Steve BOURNE;
The Unix System;
Addison-Wesley Publishing Company;
Reading (Mass.);
1982;

[CORBUGY, 90] Dominique CORBUGY and Joël DENIS;
Contribution à l'enseignement assisté par ordinateur dans le cadre des télécommunications. Etude du transfert, de l'accès et de la gestion de fichiers par FTAM;
Namur;
Année académique 1990 - 1991.

[COX, 87] Brad J. COX;
Object Oriented Programming, an Evolutionary Approach;
Addison-Wesley Publishing Company;
Reading (Mass.);
April 1987.

[CROSSLEY, 90] Kel CROSSLEY and Les GREEN;
Le design des didacticiels;
Translated from English by Alain PERRAUDIN;
ACL-Editions;
Paris;
1990.

[DUFF, 90] Chuck DUFF and Bob HOWARD;
"*Migration Patterns. Moving to object-oriented technology is more involved than simply buying a compiler*";
BYTE;
Volume 15 Number 10;
October 1990;
pp. 223 - 232.

[FAZARINC, 89] Zvonko FAZARINC;
RMG User's First Aid Kit;
Hewlett-Packard Company;
Palo Alto (Cal.);
July 1989.

[GOLDBERG, 88] Adele GOLDBERG and David ROBSON;
Smalltalk-80, the language and its implementation;
Addison-Wesley Publishing Company;
Reading (Mass.);
1988.

[GROFF, 89] James GROFF and Paul WEINBERG;
Unix, une approche conceptuelle;
Translated from English by Dimitri STOQUART;
QUE InterEditions;
Paris;
1989.

[HEIMBURGER, 90] Uwe HEIMBURGER;
Entwicklung zweier interaktiver Simulationsanwendungen zu Lehr- und Lernzwecken unter Verwendung einer objektorientierten Graphikumgebung: Benutzerschnittstelle und Realisierung physikalischer Gesetze;
Institut Für Informatik und Praktische Mathematik, Christian-Albrechts-Universität;
Kiel;
Mai 1990.

[HENSHALL, 88] John HENSHALL and Sandy SHAW;
OSI EXPLAINED. End-to-end Computer Communication Standards;
Ellis Horwood Limited;
Chichester (England);
1988.

[HP, 89] HEWLETT-PACKARD COMPANY;
RMG, a Tool Kit For Development of Visualization Courseware;
Reference Manuals N° 1 and N° 2;
Hewlett-Packard Company;
Palo-Alto (Calif.);
1989.

[HUNTER, 86] Bruce HUNTER;
Introduction à C;
Translated from English by Dominique PITT;
Sybex;
Paris;
1986.

[KERNIGHAN, 88] KERNIGHAN and Brian W. and Dennis M. RITCHIE;
The C Programming Language;
2nd Edition;
Prentice-Hall;
Englewood Cliffs;
1988.

[MASINI, 89] G.MASINI and A.NAPOLI and D.COLNET and D.LEONARD and K.TOMBRE;
Les langages à objets: langages de classes, langages de frames, langages d'acteurs;

Interéditions;
Paris;
1989.

[MEYER, 88] Bertrand MEYER;
Object-Oriented Software Construction;
Prentice Hall International Series in Computer Science, C.A.R. Hoare series editor;
Englewood Cliffs;
1988.

[SCHILDT, 90] Herbert SCHILDT;
Teach Yourself C;
Osborne McGraw-Hill;
Berkeley;
1990.

[STEPSTONE, 88] THE STEPSTONE CORPORATION;
Objective-C compiler with IC pack 101: Foundation class library. Objective-C 3.3 Reference Manual;
Productivity Products International Inc;
1988.

[TANENBAUM, 89] Andrew S. TANENBAUM;
Computer Networks;
2nd Edition;
Prentice-Hall International Editions;
Englewood Cliffs;
1989.

[TELLO, 89] Ernest R. TELLO;
Object-Oriented programming for A.I.: A guide to tools and system design;
Addison-Wesley Publishing Company;
Reading (Mass.);
1989.

[PUGH, 90] John PUGH and Wilf LALONDE;
"*Object-Oriented Programming in Smalltalk (Basics)*", *Technology of Object-Oriented languages and systems: PROCEEDINGS*;
TOOLS '90;
CNIT;
La Défence Paris;
Novembre 1989.

[WEGENER, 90] Detlev WEGENER;
Entwicklung zweier interaktiver Simulationsanwendungen zu Lehr- und Lernzwecken unter Verwendung einer objektorientierten Graphikumgebung: Konzeption und Realisierung der rundlegenden Klassen;

Institut Für Informatik und Praktische Mathematik, Christian-Albrechts-Universität;
Kiel;
Mai 1990.

Appendices

Appendix 1 : Video application listing
Appendix 2 : OSI on X-25 application listing
Appendix 3 : Example of mainClass.m file
Appendix 4 : Example of individual makefile
Appendix 5 : Example of the environment's makefile
Appendix 6 : Screen copy of the Video and OSI on X-25 applications
Appendix 7 : RMG directory structure
Appendix 8 : Example of A.menu file

Appendix 1 : Video application listing

In this appendix, we give the listing of the Video application. It encompasses the 'Video' class and the 'VideoMTree' class.

```
# include "objc.h"
# include "rmg.h"
# include "envir.h"
# include "stdio.h"
# include "stdlib.h"
# include "fcntl.h"
# include "sys/ioctl.h"
# include "sys/termio.h"
# include "sys/types.h"
# include "sys/stat.h"
# include "string.h"
# include "FontAccess.h"


@requires: RMGView, RMGAction, VideoMTree, DummyEdMT, RMGString, FontMngr, Fixtur
           Ghost, RMGLine, Mouse, DocEdit, RMGIcon, IconModel, SysIcon;

#define BUTTON_LINE_NO      3
#define ENDDISK             50000

        /* Initialisation of the constants representing the different commands t
          Video                                                                */

        /* !! if you want to change these commands change also command SE (searc
                   method posOnDisk                                           !! */

#define PLAY "PL\r"
#define STOP "RJ\r"
#define STILL "ST\r"
#define PAUSE "PA\r"
#define FASTFWD "2503FMF\r"
#define FASTBWD "2503FMR\r"
#define FRAMEREQ "[illegible]\r"      /* command used to ask the current frame number */

        * Declaration of all the action variables

static id operAction      = NULL;
static id scrollBarDown   = NULL;
static id scrollBarAction= NULL;
static id scrollBarUp     = NULL;
static id actionSelf      = NULL;
static id moveAction      = NULL;
static id actionPlay      = NULL;
static id actionStop      = NULL;
static id actionPause     = NULL;
```

```
static id actionStill     = NULL;
static id actionInpText   = NULL;
static id actionFwd       = NULL;
static id actionBwd       = NULL;

        /* Declaration of the variables that are to be used to define an IconMode
                of an icon                                                     */
static id modelPlay, modelStill, modelStop, modelFwd, modelBwd, modelCursor, mod

        /* Declaration of the variables representing the icons used in the 'chang
                option of the menu
static id iconPlay, iconBwd, iconFwd;
static id iconStill, iconStop;

        /* Declar. of variables representing the icons of the two cursors used in
static id iconCursor, arrowCursor;

static int fp;
static int errflag;
static int scrollpoint;
static int position;
static unsigned int length;
static int framenum;
static int datanum;

        /* Declar. of var. used to test the current mode of the Video player
static int pl      = 0;
static int pa      = 0;
static int st      = 0;
static int scb     = 0;
static int savestr = 0;
static int loadstr = 0;
static int forw    = 0;
static int back    = 0;

static int inputval;
static int outputval;
static int butt;

        /* Structure used to change the different parameters of the connection
struct termio param;

static char resultat[5];
static char strSave[12];
static char strLoad[12];



= Video: Envir(barchi,working,RMGVW,Primitive,Collection)
{
       /* var. used in the layout of the appl. window */
   id topBanner, scroll, scrollScale, bar,arrowB, arrowH;
       /* var. representing the action buttons of the appl. */
   id play, stop, still, inpText, ed, fwd, bwd;
       /* var. representing the scaling numbers of the scroll-bar */
   id str1, str2, str3, str4, str5, str6, str7, str8;
   id aFont;

   int buttonStr;
   int textOn;
}


/****************************************************************************
 * Define a func. assigned to the appl. window and others to get the appl.
 *         to the first level of windows, if hidden
 ****************************************************************************
static id ac_Self(aView) id aView;
        {
        if (([aView covered]) || ([aView hidden]))
                                {
                                [aView popToTop];
                                }
        return aView;
        }

/****************************************************************************
 * Same as above but assigned to topBanner and enables also to move the        *
 *         appl. around                                                       *
 ****************************************************************************
static id ac_Move(aView) id aView;
        {
        if    ([aView->superview covered]) || ([aView->superview hidden]))
                        {
                        [aView->superview popToTop];
                        }
```

```
        moveWithBnd(aView->superview);
        }

/*********************************************************************************
 * Define a function where the bar in the scollBar can be used to go forward    *
 * and backward on the disk                                                      *
 *********************************************************************************
static id ac_ScrollBar(aView) id aView;             //  where aView is the bar in th
        {
        int position, positions, diff, initcoory, i, oldfr;

        id bView;

        bView = aView->superview;
        initcoory = ([aView viewLow] - [bView viewLow]);

        if ((pl == 1) && (pa != 1) && (forw != 1) && (back != 1))
                {                                          // can use it only in play
                 write(fp,STILL,3);
                [[bView superview]videoTest];
                pa = 1;

                while ([[Mouse change:&changex:&changey]getButtons] == 1) // whi
                        {                                                   // upd

                        oldfr = framenum; // keep trace of old framenumber if ne
                        framenum = framenum - changey;

                        if ((framenum > outputval) || (framenum < inputval)) //
                                        {
                                        framenum = oldfr;
                                        }
                                        else
                                        {
                                        scrollpoint = scrollpoint + changey;
                                        [bView->superview testScrlPnt];

                                        //framenum = framenum - changey;
                                        [bView->superview moveBar];
                                        }
                        }

                if (initcoory != scrollpoint) // if place of bar as changed
                        {
                        if (st == 0) // position on disk & keep still mode
                                        {
                                        length = 10;
                                        [bView->superview posOnDisk];
                                        }
                                        else
                                        {
                                        length = 8;
                                        [bView->superview posOnDisk];
                                        }

                        }
                pa = 0;
                }
        return aView;
        }

/*********************************************************************************
 * Function defining an action, assigned to bottom arrow, to go forward on the d
 *********************************************************************************
static id ac_ScrollBarD(aView) id aView;
    {

    if ((pa == 1) && (forw != 1) && (back != 1)) // only active if in pause mode
            {
            while ([[Mouse change:&changex:&changey]getButtons] == 1)
                    {
                    framenum = framenum + 1;
                    if (framenum > outputval)
                                    {
                                    framenum = framenum - 1;
                                    }
                                    else
                                    {
                                    scrollpoint = scrollpoint - 1;
                                    [aView->viewIcon testScrlPnt];

                                    [aView->viewIcon moveBar];
                                    }

            length = 1; // first test new value, if correct then position on dis
            [aView->viewIcon [illegible]];

    return aView;
    }

*********************************************************************************
```

```
 * Function defining an action, assigned to top arrow, to go backward on the dis
 ******************************************************************************
static id ac_ScrollBarU(aView) id aView;
   {

   if ((pa == 1) && (forw != 1) && (back != 1))
           {
           while ([[Mouse change:&changex:&changey]getButtons] == 1)
                   {
                   framenum = framenum - 1;
                   if (framenum < inputval)
                                  {
                                  framenum = framenum + 1;
                                  }
                                  else
                                  {
                                  scrollpoint = scrollpoint - 1;
                                  [aView->viewIcon testScrlPnt];

                                  [aView->viewIcon moveBar];
                                  }
                   }
           length = 8;
           [aView->viewIcon posOnDisk];
           }
   return aView;
   }

/*****************************************************
 * Define a function to send order -Play- to video      *
 *       -> order is : PL (in ASCII alpha characters)     *
 *****************************************************/
static id ac_Play(aView) id aView;
        {
        id sView;

        if(pl == 0) // send PLAY only if player not in that mode
                {
                sView = aView->superview;
                [[aView bkgd: GREYMAGENTA]redraw]; // update color of button

                scb = inputval; // redscale scroll-bar
                [sView scale: (sView->scrollScale) low: scb high: (scb + 340)];

                if (inputval == 0) // play from beginning
                                 {
                                 write (fp, PLAY, 3);
                                 [sView videoTest];
                                 }
                                 else
                                 {
                                 length = (12 * (sizeof(char)));
                                 [sView posOnDisk];
                                 }
                scrollpoint = ([sView->scroll viewLow] + 363); // le + 363 = ???
                [sView moveBar];

                [sView startRun]; // position appl. in activeCollection so that
                                 // update scroll-bar

                pl = 1; // play mode on
                pa = 0; // pause & still mode off
                st = 0;
                }
        return aView;
        }

/************************************************************************
 * Define a function to send order -Pause- to video                      *
 *       -> order is : PA (in ASCII alpha characters)                     *
 *          this action can't be used if PLAY has not been activated first *
 ************************************************************************/
static id ac_Pause(aView) id aView;
        {
        id bView;

        if ((pl == 1) && (st != 1)) // able to pause only in play mode & not(sti
                    {
                    if(pa == 0) // switch to pause mode else restart play mode
                               {
                               [[aView bkgd: GREYMAGENTA] redraw];

                               bView = aView->viewIcon;                 /* point
                               pa = 1; // pause mode on

                               write (fp, PAUSE, 3);
                               [[aView superview] videoTest];
                               }
                               else
                               {
                               [[aView bkgd: GREY] redraw];
```

```
                                pa = 0; // pause mode off

                                write (fp, PLAY, 3);
                                [[aView superview] videoTest];

                                forw = 0; // forw & back mode off because if in th
                                back = 0; // then restart play mode after switchin
                                [[[aView superview]->fwd bkgd: GREY_7] redraw]; //
                                [[[aView superview]->bwd bkgd: GREY_7] redraw]; //

                                }

        return aView;
        }

/*****************************************************************
 * Define a function to send order -Stop- to video          *
 *      -> order is : RJ (in ASCII alpha characters)        *
 *****************************************************************/
static id ac_Stop(aView) id aView;
        {
        id sView;

        [[aView bkgd: GREYMAGENTA] redraw];

        if((pl == 1) || (st == 1))
                {
                write(fp,STOP,3);

                sView = aView->superview;
                [sView videoTest];

                [[sView->play bkgd: GREY_7]redraw]; // resetting all buttons to
                [[sView->still bkgd: GREY_7]redraw];
                [[sView->scroll bkgd: GREY] redraw];
                [[sView->fwd bkgd: GREY_7] redraw];
                [[sView->bwd bkgd: GREY_7] redraw];

                [sView stopRun]; // remove appl. of activeCollection

                pl = 0; // setting all modes to off
                pa = 0;
                st = 0;
                forw = 0;
                back = 0;
                framenum = inputval;
                }

        [[aView bkgd: GREY_7] redraw];
        return aView;
        }

/*****************************************************************
 * Define a function to send order -Still- to video         *
 *      -> order is : ST (in ASCII alpha characters)        *
 *****************************************************************/
static id ac_Still(aView) id aView;
        {
        if ((pl == 1) && (pa != 1)) // only in play & not(pause) modes
                    {
                    if(st == 0)
                                {
                                [[aView bkgd: GREYMAGENTA]redraw];
                                st = 1;

                                write(fp,STILL,3);
                                [[aView superview] videoTest];
                                }
                                else
                                {
                                [[aView bkgd: GREY_7]redraw];
                                write(fp,PLAY,3);
                                [[aView superview] videoTest];

                                st = 0;
                                forw = 0; // same purpose as for pause
                                back = 0;
                                [[[aView superview]->fwd bkgd: GREY_7] redraw];
                                [[[aView superview]->bwd bkgd: GREY_7] redraw];
                                }
                    }

        return aView;
        }

/*****************************************************************
 * Define a func. to send order -Fast Forward- to Video          *
 *        -> order is : 250BF (to change speed MF                *
 *****************************************************************/
static id ac_Fwd(aView) id aView;
        {
        char *order;
```

```
        if ((pl == 1) && (st == 0)) // not available in still mode
                {
                if (back != 1)
                        {
                        if (forw == 0)
                                {
                                write (fp, FASTFWD, 8);

                                [[aView superview] videoTest];
                                [[aView bkgd: GREYMAGENTA] redraw];

                                forw = 1;
                                }
                                else
                                {
                                if (pa == 1)
                                        {
                                        order = PAUSE;
                                        }
                                        else
                                        {
                                        order = PLAY;
                                        }
                                write(fp, order, 3);
                                [[aView superview] videoTest];

                                [[aView bkgd: GREY_7] redraw];
                                forw = 0;
                                }
                        }
                }
        }

/**************************************************************
 * Define a Func. to send order -Fast Backward- to Video        *
 *          -> order is : 250SPMR                                *
 **************************************************************/
static id ac_Bwd(aView) id aView;
        {
        char *order;

        if ((pl == 1) && (st == 0))
                {
                if (forw != 1)
                        {
                        if (back == 0)
                                {
                                write (fp, FASTBWD, 8);

                                [[aView superview] videoTest];
                                [[aView bkgd: GREYMAGENTA] redraw];
                                back = 1;
                                }
                                else
                                {
                                if (pa == 1)
                                        {
                                        order = PAUSE;
                                        }
                                        else
                                        {
                                        order = PLAY;
                                        }
                                write(fp, order, 3);
                                [[aView superview] videoTest];

                                [[aView bkgd: GREY_7] redraw];
                                back = 0;
                                }
                        }
                }
        }

/**************************************************************
 * Func. that calls the method to create a text window         *
 **************************************************************
static id ac_Text(aView) id aView;
        {
        [[aView superview] textWindow: [aView viewIcon]];
        }

 **************************************************************
 * Func. used to perform the action of the superview I'm in *
 **************************************************************
[illegible]
        {
        [illegible]
        return aView;
        }

[illegible]
```

```
	[self delayInit];

/* Change the parameters of the connexion to have :
   4800 bauds, 1start bit, 1stop bit, 8 bits.          */

	fp = open("/dev/ttyd00",O_RDWR);
	if (fp == -1)
		{
		printf("Can't open device file.\n");
		exit(0);
		}

	errflag = ioctl(fp,TCGETA,&param);
	param.c_cflag |= CLOCAL;
	param.c_cflag &= ~HUPCL;
	param.c_cflag ^= 0000011;
	param.c_iflag |= IXON;
	param.c_iflag |= IXOFF;
	param.c_iflag &= ~ECHO;
	param.c_cc[VMIN] = 0x0;
	param.c_cc[VTIME] = 0xff;
	errflag = ioctl(fp,TCSETA,&param);

/* Initialise certain important variables              */
	inputval = 0;       // begin at
	outputval = ENDDISK;// end at
	scb = 0;            // begin scaling at 0
	pl = st = pa = forw = back = 0; // all modes off
	butt = 1;           // buttons are set to strings ( not icons )
	textOn = 0;         // no text window created

/* Define the main view for the application*/

	if (!actionSelf)
			{
			actionSelf = [[RMGAction newFuncAction: ac_Self] myCursor
			}
	self = [self origin: 200: 200 extent: 200: 400 superview: anEnvir bkgd:

	[[[[self frameWidth: 5]
		frameColor: CYAN]
		idAction: actionSelf]
		viewIcon: self];

	aFont = [FontMngr sysfont];

/* Define the top banner of the application, where the title
   will take place and the enlarging icon                       */

	if (!moveAction)
			{
			moveAction = [[RMGAction newFuncAction: ac_Move] myCursor
			}
	topBanner = [[[[[[[[RMGString Type: StickTop extent: 17 superview: self
					string: " Video Control "]
					font: aFont]
					color: BLACK]
					horFill: TRUE]
					center]
					centerV]
					show]
					idAction: moveAction];


/* Define the scroll bar of the application                    */

	if (!actionPause)
		actionPause = [[RMGAction newFuncAction: ac_Pause] myCursor: ico
	scroll = [[[RMGView relative: 173 :0 extent: 17: 380
			superview: self
			bkgd: GREY]
			idAction: actionPause]
			show];

/* Define the area within self where thw scaling numbers will be printed
	scrollScale = [[[[RMGView relative: 126: 0 extent: 47: 380 superview: se
				idAction: actionSelf]
				viewIcon: self]
				show];

	[self initScale: scrollScale low: scb high: (scb + 340)];

/* Define the bar in the scroll-bar
	if (!scrollBarAction)
		scrollBarAction = [[RMGAction newFuncAction: ac_ScrollBar] myCur
	bar = [[[[RMGView relative:0:370 extent:17:10
			superview: scroll
			bkgd: DARKBLUE]
			viewIcon: self]
			idAction: scrollBarAction]
```

```
                        show];

/* Initialise different parameters, so that the bar within the scrollBar .
   can move according to framenumbers reached

        [scroll viewIcon: bar];
        scrollpoint = [bar viewLow];
        framenum = 0;

// Place on top and bottom of scrollBar two arrows- so that the user can go forw

        if(!scrollBarUp)
                scrollBarUp = [[RMGAction newFuncAction: ac_ScrollBarU] myCursor
        arrowB = [[[[[Fixtur17 up_arrow: scroll extent: 17 type: StickTop]
                                viewIcon: self]
                                bkgd: BLUE]
                                idAction: scrollBarUp]
                                show];

        if(!scrollBarDown)
                scrollBarDown = [[RMGAction newFuncAction: ac_ScrollBarD] myCurs

        arrowT = [[[[[Fixtur17 down_arrow: scroll extent: 17 type: StickBottom]
                                viewIcon: self]
                                bkgd: BLUE]
                                idAction: scrollBarDown]
                                show];

/* Define action buttons and actions attach to them */

        if(!superAction)
               superAction = [[RMGAction newFuncAction: superAct] myCursor: icon

        if (!actionPlay)
                {
                actionPlay = [[RMGAction newFuncAction: ac_Play] myCursor: iconC
                }
        play = [[[[[[[[[[RMGString relative: 15: 327 extent: 70: 30
                                  superview:self
                                  bkgd: GREY_7]
                                  frameWidth: 2]
                                  frameColor: BLACK]
                                  string: " Play "]
                                  font: aFont]
                                  color: BLACK]
                                  viewIcon: self]
                                  center]
                                  centerV]
                                  idAction: actionPlay]
                                  show];

        [self buttonFace: play color1: GREY_2 color2: GREY_15];

        if (!actionStop)
                {
                actionStop = [[RMGAction newFuncAction: ac_Stop] myCursor: iconC
                }
        stop = [[[[[[[[[[RMGString relative: 15: 297 extent: 70: 30
                                  superview:self
                                  bkgd: GREY_7]
                                  frameWidth: 2]
                                  frameColor: BLACK]
                                  string: " Stop "]
                                  font: aFont]
                                  color: BLACK]
                                  viewIcon: self]
                                  center]
                                  centerV]
                                  idAction: actionStop]
                                  show];

        [self buttonFace: stop color1: GREY_2 color2: GREY_15];


        if (!actionStill)
                {
                actionStill = [[RMGAction newFuncAction: ac_Still] myCursor: ico
                }
        still = [[[[[[[[[[RMGString relative: 15: 267 extent: 70: 30
                                  superview:self
                                  bkgd: GREY_7]
                                  frameWidth: 2]
                                  frameColor: BLACK]
                                  string: " Still "]
                                  font: aFont]
                                  color: BLACK]
                                  viewIcon: self]
                                  center]
                                  centerV]
                                  idAction: actionStill]
                                  show];

        [self buttonFace: still color1: GREY_2 color2: GREY_15];
```

```
        if (!actionFwd)
                        {
                        actionFwd = [[RMGAction newFuncAction: ac_Fwd] myCursor: i
                        }
        fwd = [[[[[[[[[[[RMGString relative: 15: 217 extent: 70: 30
                                    superview: self
                                    bkgd: GREY_7]
                                    frameWidth: 2]
                                    frameColor: BLACK]
                                    string: " Fwd "]
                                    font: aFont]
                                    color: BLACK]
                                    viewIcon: self]
                                    center]
                                    centerV]
                                    idAction: actionFwd]
                                    show];
        [self buttonFace: fwd color1: GREY_2 color2: GREY_15];

        if (!actionBwd)
                        {
                        actionBwd = [[RMGAction newFuncAction: ac_Bwd] myCursor: i
                        }
        bwd = [[[[[[[[[[[RMGString relative: 15: 187 extent: 70: 30
                                    superview: self
                                    bkgd: GREY_7]
                                    frameWidth: 2]
                                    frameColor: BLACK]
                                    string: " Bwd "]
                                    font: aFont]
                                    color: BLACK]
                                    viewIcon: self]
                                    center]
                                    centerV]
                                    idAction: actionBwd]
                                    show];
        [self buttonFace: bwd color1: GREY_2 color2: GREY_15];

        if (!actionInpText)
                            {
                            actionInpText = [[RMGAction newFuncAction: ac_Text] my
                            }
        inpText = [[[[[[[[[[[RMGString relative: 15: 137 extent: 70: 30
                                        superview: self
                                        bkgd: GREY_7]
                                        frameWidth: 2]
                                        frameColor: BLACK]
                                        string: " Text "]
                                        font: aFont]
                                        color: BLACK]
                                        viewIcon: anEnvir]
                                        center]
                                        centerV]
                                        idAction: actionInpText]
                                        show];
        [self buttonFace: inpText color1: GREY_2 color2: GREY_15];

/* Print warning window to ensure that the video is ON            */
        [self viewError: 4 into: " VIDEO is ON "];

        return self;
  }

- extraNewIn: anEnvir
  {
        menuTree = [VideoMTree getIt];
        [self showAll];

        viewIcon = [[Ghost stringGhost: "Video" In: anEnvir]
                            viewIcon:self];

/* Define all the IconModels for the icons used in the buttons            */
        modelPlay = [[IconModel readFile: "/usr/RMG/DATA/ICONS/Play.icon"]
                                hotSpot: 0: 0];
        modelStill = [[IconModel readFile: "/usr/RMG/DATA/ICONS/Still.icon"]
                                 hotSpot: 0: 0];
        modelStop = [[IconModel readFile: "/usr/RMG/DATA/ICONS/Stop.icon"]
                                hotSpot: 0: 0];
        modelFwd = [[IconModel readFile: "/usr/RMG/DATA/ICONS/Fwd.icon"]
                               hotSpot: 0: 0];
        modelBwd = [[IconModel readFile: "/usr/RMG/DATA/ICONS/Bwd.icon"]
                               hotSpot: 0: 0];

        return self;
  }

- delayInit
  {
        static BOOL beenhere = FALSE;

        if(!beenhere)
```

```
		if(self == Video)
			{
			[VideoMTree delayInit];
			bezhere = TRUE;
			}
			else [Video delayInit];

/* Define the two cursors used in the application
	modelCursor = [[IconModel readFile: "/usr/RMG/DATA/ICONS/SYBICON/barMous
				hotSpot: 0: 0];
	iconCursor = [[SysIcon newIn: NULL]
				icon: modelCursor
				mask: modelCursor];

	modelArrowCu  = [[IconModel readFile: "/usr/RMG/DATA/ICONS/SYBICON/dir4ba
	arrowCursor = [[SysIcon newIn: NULL]
				icon: modelArrowCur
				mask: modelCursor];

	return self;
  }

/*********************************************
 * Method used to shrink the application to an icon *
 *********************************************/
- iconify
{
	[self iconize];
	return self;
}

/*****************************************************************************
 * Method to read into the device file, the informations coming from the video
 *****************************************************************************
- readDevice
{
	char data;

	int i;
	i = 0;

	datanum = 0;
	data ='0';
	strcpy(resultat,"");

	while((data!='\r'))
	{
		read(fp,&data,1);
		if((data >= 48) && (data <= 57) && (i == 0)) // if data is a num
			{datanum = (datanum*10)-(48-data);}
		if((((data < 48) || (data > 57)) && (data != '\r')) || ((data >=
			{			// if data letter & not CR, or a number
			 resultat[i]=data;
			 resultat[i+1]='\0';
			 i+=1;
			}

	}
	return self;
}

/**************************************************************
 * Method to move the bar within the scrollBar to scrollpoint *
 **************************************************************/
- moveBar
{
	[[bar move:0:scrollpoint]show];
	return self;
}

/***********************************************************************
 * Method that updates the position of the bar within the scrollBar,*
 *  when the video is in play mode                                  *
 ***********************************************************************/
- update
{
	int diff;

	if((pl == 1) && (pa == 0) && (st == 0)) // update scroll-bar only in play
	     {
	     write(fp,FRAMEREQ,3);
	     [self readDevice];

	     if(datanum != 0)
			{
			diff = datanum - framenum;

			scrollpoint = scrollpoint - diff;
			[self testScrlPt];
			[self moveBar];
			framenum = datanum;
			}
```

```
            if(datanum >= outputval) // if bottom end of working area reached
                        {
                        write(fp,STILL,3); // switch to still mode (only possi
                        [self videoTest];

                        [self stopRun];

                        pl = 0; // reset all modes to off so that can't go ove
                        pa = 0;
                        forw = 0;
                        back = 0;
                        st = 1;
                        }
            if(datanum < inputval) // if starting point of working area reached
                      {
                      write(fp, STILL, 3);
                      [self videoTest];

                      [self stopRun];

                      pl = 0;
                      pa = 0;
                      forw = 0;
                      back = 0;
                      st = 1;
                      }
            }
        return self;
}

/***************************************************************
 * Method that places the application in the active collection *
 ***************************************************************/
- startRun
{
        [superview installActive:self];
        return self;
}

/*******************************************************************
 * Method that removes the application from the active collection  *
 *******************************************************************/
- stopRun
{
        [superview deleteActive:self];
        return self;
}

/*******************************************************************************
 * Method that designes a shaded button for aView, using aColor and bColor as sh
 *******************************************************************************
-buttonFace: aView color1: (int)aColor color2: (int)bColor
{
        register int i;

        for (i=0; i < BUTTON_LINE_NO; i++)
        {
                [[[RMGLine superview: aView color: aColor]
                        relP1: 0: i length: [aView in_width]: 0]
                        show];
                [[[RMGLine superview: aView color: aColor]
                        relP1: [aView in_width]-1-i: 0 length: 0: [aView in_h
                        show];
        }

        for (i=0; i < BUTTON_LINE_NO; i++)
        {
                [[[RMGLine superview: aView color: bColor]
                        relP1: 0: [aView in_height]-i length: [aView in_width
                        show];
                [[[[RMGLine superview: aView color: bColor]
                        relP1: i: i
                        length: 0: [aView in_height]]
                        idAction: superAction]
                        show];
        }
        return self;
}

/*******************************************************************************
 * Method that initialise the scaling of the scrollBar from aLowScale to aLowSca
 *******************************************************************************
- initScale: aView low: (int)aLowScale high: (int)aHighScale // no indexed var.
{
        int temp;

        temp = aLowScale;

        str1 = [[[[[ModStr] font: aFont superview: aView color: BLACK]
                                string: "": "-"]
                                dataOrigin: 0: 346]
                                setIValue: temp]
                                setRJust: 5]
```

```
                        show];

        temp += 50;
        str2 = [[[[[ModStrI font: aFont superview: aView color: BLACK]
                            string: "": "-"]
                            dataOrigin: 0: 296]
                            setIValue: temp]
                            setRJust: 5]
                            show];
        temp += 50;
        str3 = [[[[[ModStrI font: aFont superview: aView color: BLACK]
                            string: "": "-"]
                            dataOrigin: 0: 246]
                            setIValue: temp]
                            setRJust: 5]
                            show];
        temp += 50;
        str4 = [[[[[ModStrI font: aFont superview: aView color: BLACK]
                            string: "": "-"]
                            dataOrigin: 0: 196]
                            setIValue: temp]
                            setRJust: 5]
                            show];
        temp += 50;
        str5 = [[[[[ModStrI font: aFont superview: aView color: BLACK]
                            string: "": "-"]
                            dataOrigin: 0: 146]
                            setIValue: temp]
                            setRJust: 5]
                            show];
        temp += 50;
        str6 = [[[[[ModStrI font: aFont superview: aView color: BLACK]
                            string: "": "-"]
                            dataOrigin: 0: 96]
                            setIValue: temp]
                            setRJust: 5]
                            show];
        temp += 50;
        str7 = [[[[[ModStrI font: aFont superview: aView color: BLACK]
                            string: "": "-"]
                            dataOrigin: 0: 46]
                            setIValue: temp]
                            setRJust: 5]
                            show];
        temp += 40;
        str8 = [[[[[[ModStrI font: aFont superview: aView color: BLACK]
                            string: "": "-"]
                            dataOrigin: 0: 15]
                            setIValue: temp]
                            setRJust: 5]
                            idAction: actionSelf]
                            show];

        scb = aHighScale;
        return self;
}

/*****************************************************************************
 * Method that rescales the scrollBar in aView, from aLowScale to aHighScale *
 *****************************************************************************/
- scale: aView low: (int)aLowScale high: (int)aHighScale
{
        int temp;

        temp = aLowScale;

        [[str1 setIValue: temp] redraw];

        temp += 50;
        [[str2 setIValue: temp] redraw];

        temp += 50;
        [[str3 setIValue: temp] redraw];

        temp += 50;
        [[str4 setIValue: temp] redraw];

        temp += 50;
        [[str5 setIValue: temp] redraw];

        temp += 50;
        [[str6 setIValue: temp] redraw];

        temp += 50;
        [[str7 setIValue: temp] redraw];

        temp += 40;
        [[str8 setIValue: temp] redraw];

        scb = aHighScale;
        return self;
}
```

```
/*************************************************************************
 * Method that test the place of the bar in the scrollBar and eventually updates
 *          and place of bar in scrollBar
 *************************************************************************
- testScrlPnt
{
        if (scrollpoint < ([scroll viewLow]+17)) // if bar at end of scroll-bar
                {
                if (framenum < ENDDISK) // if disk not at end
                        {
                        [self scale: scrollScale low: scb high: (scb + 340)]; //
                        scrollpoint = ([scroll viewLow] + 358); // reset bar to
                        }
                        else
                        {
                        scrollpoint = ([scroll viewLow] + 17);
                        }
                }
                else
                {
                if (scrollpoint > ([scroll viewLow] + 358)) // if bar at beginni
                        {
                        if (framenum > 0) // if disk not at beginning
                                {
                                scrollpoint = ([scroll viewLow] + 17); // I don't
                                [self scale: scrollScale low: (scb - 680) high: (
                                }
                                else
                                {
                                scrollpoint = ([scroll viewLow] + 358);
                                }
                        }
                }
        return self;
}

/*************************************************************************
 * Method to transform the framenumber into a string, so that it can be send to
 *      the -SEARCH- (SE) order, though moving on the disk to the right place
 *************************************************************************
- posOnDisk
{
        int adresse, i, j;
        float adressef;
        char adrcar[13];

        adresse = framenum;

        i = 5;
        j = 0;
        adrcar[5] = 'S';
        adrcar[6] = 'E';
        adrcar[7] = '\r';
        adrcar[8] = '\0';

        while(i > j) // convert the number into a string
                {
                i -= 1;
                adressef = adresse;
                adressef = adressef / 10;
                adresse = adresse / 10;
                adressef = (adressef - adresse) * 10;
                adrcar[i] = adressef + 48;
                }

        if (length == 12) // if order is PLxxxxxSEPL\r
                        {
                        write(fp, PLAY, 3);
                        [self videoTest];
                        }

        write(fp,adrcar,8); // send SEARCH order
        [self videoTest];

        if ((length == 10) || (length == 12)) // if order is (xxxxxSEPL\r) or (P
                        {
                        write(fp, PLAY, 3);
                        [self videoTest];
                        }

        return self;
}

/*************************************************************************
 * Method that receives a number string (aStr) entered via the menu and after
 * controlling sets it as the starting point of the working area
 *************************************************************************
- input: (char *) aStr
{
        int oldin;
```

```
        oldin = inputval;

        if (pl == 0) // changed only if not in play mode
                {
                inputval = atoi(aStr); // conversion into integer

                if (inputval > outputval)
                                {
                                inputval = oldin;
                                }
                                else
                                {
                                framenum = inputval;
                                }
                }
        return self;
}

/*************************************************************************
 * Method that receives a number string (aStr) entered via the menu and after
 * controlling sets it as the starting point of the working area
 *************************************************************************
- output: (char *) aStr
{
        int oldout;

        oldout = outputval;

        if (pl == 0)
                {
                outputval = atoi(aStr);

                if (outputval < inputval)
                                {
                                outputval = oldout;
                                }
                }
        return self;
}

/*************************************************************************
 * Method that saves the parameters of the working area (for the moment only beh
 * and end of working area) and the text related to it in two files aStr.vcn and
 * aStr.vct
 *************************************************************************
- saveFile: (char *) aStr
{
        int i;
        FILE *fd;
        int erreur;

        char str[13];
        char fileDotVCT[1024] ;
        char fileDotVCN[1024];

        strcpy(fileDotVCN,"./"); // will save the files in current directory
        strcpy(fileDotVCT,"./");

        i = 0;
        erreur = 0;

        strcat(fileDotVCT,aStr); // create the file name as ./aStr.vct
        strcat(fileDotVCT,".vct");

        if (textOn != 0) // if text window exists
                {
                if ((fd = fopen(fileDotVCT, "w")) != NULL)
                                {
                                i = fileno(fd); // get the filenumber
                                [ed saveFileIn: i]; // save text window
                                fclose(fd);
                                }
                                else // opening error
                                {
                                [self viewError: 1 into: fileDotVCT];
                                }
                }

        strcat(fileDotVCN,aStr); // create the file name ./aStr.vcn
        strcat(fileDotVCN,".vcn");

        if ((fd = fopen(fileDotVCN,"w")) != NULL)
                {
                if (fwrite(&inputval, sizeof inputval, 1, fd) == 1) // if fi
                                {
                                if (fwrite(&outputval, sizeof outputval, 1, fd) !=
                                                {
                                                [self viewError: 2 into: fileDotVCN]; //
                                                erreur = 1;
                                                }
                                                else
                                                {
```

```
					//printf(" OK ");
					}
				}
				else
				{
				[self viewError: 2 into: fileDotVCN]; // error wri
				erreur = 1;
				}
			fclose(fd);
			}
			else // opening error
			{
			[self viewError: 1 into: fileDotVCN];
			}

	if (erreur == 1) // error writing in file, not all numbers written => de
			{
			unlink(fileDotVCN);
			}

	erreur = 0;

	return self;
}

/*************************************************************************
 * Method that loads the number and text files (aStr.vcn & aStr.vct) into the ap
 *************************************************************************
- loadFile: (char *) aStr
{
	int i, oldint;
	FILE *fd;

	char str[13];
	char fileDotVCT[1024];
	char fileDotVCN[1024];

	strcpy(fileDotVCT,"./");
	strcpy(fileDotVCN,"./");

	oldint = inputval;

	i = 0;

	if (textOn == 0) // if text window not present => create
			{
			[self textWindow: [inpText viewIcon]];
			}

	strcat(fileDotVCT,aStr);
	strcat(fileDotVCT,".vct");

	if ((fd = fopen(fileDotVCT, "r")) != NULL)
		{
		strcpy(fileDotVCT,"");
		strcpy(fileDotVCT,aStr);
		strcat(fileDotVCT,".vct");

		[ed readFile: fileDotVCT];
		fclose(fd);
		}
		else
		{
		[self viewError: 1 into: fileDotVCT];
		}

	strcat(fileDotVCN,aStr);
	strcat(fileDotVCN,".vcn");

	if ((fd = fopen(fileDotVCN, "r")) != NULL)
			{
			if (fread(&inputval, sizeof inputval, 1, fd) == 1)
				{
				if (fread(&outputval, sizeof outputval, 1, fd) == 1
					{
					//printf(" %d %d ",inputval, outputval);
					}
					else
					{
					inputval = oldint;
					[self viewError: 3 into: fileDotVCN];
					}
				}
				else
				{
				[self viewError: 3 into: fileDotVCN];
				}
			fclose(fd);
			}
			else
			{
```

```
		[self viewError: 1 into: fileDotVCN];
		}

	return self;
}

/************************************************************************
 * Method to quit the application, that frees certain variables
 ************************************************************************
- quit_app
{
	close(fp);
	if (ed != NULL)
			[ed quit_app];

	if (iconPlay == NULL)
			{
			[iconPlay free];
			[iconStop free];
			[iconStill free];
			[iconFwd free];
			[iconBwd free];
			}
	//[iconCursor free];

	[modelPlay free];
	[modelStop free];
	[modelStill free];
	[modelFwd free];
	[modelBwd free];
	[modelCursor free];
	[modelArrowCur free];

	[super quit_app];
	return NULL;
}

/************************************************************************
 * Method to test that an order to the video has been well executed ( test
 * arrival of "R" in device file )                                     *
 ************************************************************************
- videoTest
{
	int i;

	[self readDevice];


	while ((strcmp(resultat,"R")) != 0)
			{
			[self readDevice];
			}
}

/************************************************************************
 * Method called to change the string of the buttons into icons and vice-versa*
 ************************************************************************/
- changeButtons
{
	if (butt == 1) // buttons have string on them
				{
				[[play string: ""] show]; //supress all strings
				[[still string: ""] show];
				[[stop string: ""] show];
				[[fwd string: ""] show];
				[[bwd string: ""] show];

				butt = 0;
				buttonStr = 0;

				// create the icons on buttons
				iconPlay = [[[RMGIcon makeFromModel: modelPlay
							in: play
							at: 25: 4]
							idAction: superAct
							show];
				iconStill = [[[RMGIcon makeFromModel: modelStill
							in: still
							at: 25: 4]
							idAction: superAc
							show];
				iconStop = [[[RMGIcon makeFromModel: modelStop
							in: stop
							at: 25: 4]
							idAction: superAct
							show];
				iconFwd = [[[RMGIcon makeFromModel: modelFwd
							in: fwd
							at: 25: 4]
							idAction: superAct
							show];
```

```
                              iconBwd = [[[HMGIcon makeFromModel: modelBwd
                                                               in: bwd
                                                               at: 25: 4]
                                                               idAction: superActi
                                                               show];
                              [self redraw];
                              }
                              else
                              {
                              [[iconPlay erase] free]; // supress icons on buttons
                              [[iconStill erase] free];
                              [[iconStop erase] free];
                              [[iconFwd erase] free];
                              [[iconBwd erase] free];

                              // reset the strings
                              [[[[play string: " Play "]
                                        center]
                                        centerV]
                                        show];
                              [[[[still string: " Still "]
                                         center]
                                         centerV]
                                         show];
                              [[[[stop string: " Stop "]
                                        center]
                                        centerV]
                                        show];
                              [[[[fwd string: " Fwd "]
                                       center]
                                       centerV]
                                       show];
                              [[[[bwd string: " Bwd "]
                                       center]
                                       centerV]
                                       show];

                              butt = 1;
                              buttonStr = 1;
                              [self redraw];
                              }
            return self;
}

/*****************************************************************************
 * Method used to create or manage the text window in the environment (aEnv)
 *****************************************************************************
- textWindow: (id) aEnv
{
            if (textOn == 0) // if not been created
                              {
                              ed = [[DocEdit newIn: aEnv] extraNewIn: aEnv]; // crea
                              [[ed activeTerm: ed] menuTree: [DummyEdMT getIt]]; //
                                                                   // create a Dummy menu
                              [[ed move: 400: 300] show];
                              [[inpText bkgd: GREYMAGENTA] redraw];

                              textOn = 2;
                              }
                              else
                              {
                              if (textOn == 1) // if hidden
                                              {
                                              [ed show];
                                              [[inpText bkgd: GREYMAGENTA] redraw];

                                              textOn = 2;
                                              }
                                              else
                                              {
                                              [ed hide];
                                              [[inpText bkgd: GREY_7] redraw];

                                              textOn = 1;
                                              }
                              }
            return ed;
}

/*****************************************************************************
 * Method used to print an error message related to aReason and to the aFile
 *  in the application
 *****************************************************************************
- viewError: (int)aReason into: (char *)aFile
{
            id errorView, errorStickB, errorStickT;
            char errorType[1024];
            int large;

            sprite_erase();

            switch(aReason)
                              {
```

```
                case 1: strcpy(errorType," error Opening file ");
                        break;
                case 2: strcpy(errorType," error Writing file ");
                        break;
                case 3: strcpy(errorType," error Reading file ");
                        break;
                case 4: strcpy(errorType," CHECK that the ");
                        break;
                }

        large = strlen(errorType);

        errorView = [[[[[[[[[[RMGString origin: 50: 75 extent: (large + 4) *
                                superview: self bkgd: DARKBLUE]
                                frameWidth: 2]
                                frameColor: RED]
                                string: aFile]
                                font: aFont]
                                color: RED]
                                horFill: TRUE]
                                center]
                                centerV]
                                show];

        errorStickT = [[[[[[[[RMGString Type: StickTop extent: (FONT_HEIGHT(a
                                 superview: errorView bkgd: DARKBLUE
                                 string: errorType]
                                 font: aFont]
                                 color: RED]
                                 horFill: TRUE]
                                 center]
                                 centerV]
                                 show];

        errorStickB = [[[[[[[[[[RMGString Type: StickBottom extent: (FONT_HEI
                                 superview: errorView bkgd: DARKBLUE
                                 frameWidth: 1]
                                 frameColor: WHITE]
                                 string: " Press Mouse Button "]
                                 font: aFont]
                                 color: WHITE]
                                 horFill: TRUE]
                                 center]
                                 centerV]
                                 show];

        while ([[Mouse change: &changex: &changey] getButtons] == 0)
               {
               }

        [[errorStickB hide] free];
        [[errorStickT hide] free];
        [[errorView hide] free];

        [self redraw];

        sprite_show();

        return self;
}

=:
```

```
#include "objc.h"
#include "rmg.h"
#include "envir.h"
# include "stdio.h"


@requires FontMngr,RMGMenu1, MTreeAct, MTreeActOne, EntryPad;


static id onlyInstance;


= VideoMTree: EnvMTree(barchi,working,RMGVW,Collection,Primitive)

static id ac_Save(aView) id aView;
        {
        id selected, bView;
        char *aStr;

        selected = [aView pkItmUnder: sprite with: Mouse];

        if (selected)
                {
                sprite_erase();
                aStr = [[aView showAll] keyInStr];
                [aView hideAllPrev];
                }
        if ((strcmp(aStr,"")) != 0)
                        {
                        bView = [[[aView viewIcon] APPL] receiver];
                        [bView saveFile: aStr];
                        }
        sprite_show();
        return selected;
        }

static id ac_Load(aView) id aView;
        {
        id selected, bView;
        char *aStr;

        selected = [aView pkItmUnder: sprite with: Mouse];

        if (selected)
                {
                sprite_erase();
                aStr = [[aView showAll] keyInStr];
                [aView hideAllPrev];
                }
        if ((strcmp(aStr,"")) != 0)
                        {
                        bView = [[[aView viewIcon] APPL] receiver];
                        [bView loadFile: aStr];
                        }
        sprite_show();
        return selected;
        }

static id ac_Input(aView) id aView;
        {
        id selected, bView;
        char *aStr;

        selected = [aView pkItmUnder: sprite with: Mouse];

        if (selected)
                {
                sprite_erase();
                aStr = [[aView showAll] keyInStr];
                [aView hideAllPrev];
                }
        if ((strcmp(aStr,"")) != 0)
                        {
                        bView = [[[aView viewIcon] APPL] receiver];
                        [bView input: aStr];
                        }
        sprite_show();
        return selected;
        }
```

```
static id ac_Output(aView) id aView;
	{
	id selected, bView;
	char *aStr;

	selected = [aView pkItmUnder: sprite with: Mouse];

	if (selected)
			{
			sprite_erase();
			aStr = [[aView showAll] keyInStr];
		.	[aView hideAllPrev];
			}
	if ((strcmp(aStr,"")) != 0)
				{
				bView = [[[aView viewIcon] APPL] receiver];
				[bView output: aStr];
				}
	sprite_show();
	return selected;
	}

+ getIt
  {
  return onlyInstance;
  }


+ delayInit
  {
  id aFont;
  id entryPad, entryPad1, entryPad2, entryPad3;

  static BOOL beenhere = FALSE;


  if(!beenhere)
		{
		beenhere = TRUE;
		aFont = [FontMngr  sysfont];
		self = onlyInstance = [self  new];


  if(!beenhere)
		{
		beenhere = TRUE;
		aFont = [FontMngr  sysfont];
		self = onlyInstance = [self  new];

		rootMenu = [[[RMGMenu1 char_wide: 15 items: 7 font: aFont]
					bkgd: GOLD]
					APPL: self];

		entryPad = [[[EntryPad char_wide: 12 max_wid: 12 font: aFont] action
		entryPad1 = [[[EntryPad char_wide: 12 max_wid: 12 font: aFont] actio
		entryPad2 = [[[EntryPad char_wide: 5 max_wid: 5 font: aFont] action:
		entryPad3 = [[[EntryPad char_wide: 5 max_wid: 5 font: aFont] action:


		[[[rootMenu  at: 0  putStr: "!!Quit!!"]
				center]
				idAction: [MTreeAct sel:"quit_app"]];

		[[[rootMenu at: 1 putStr: "Icon"]
				center]
				idAction: [MTreeAct sel:"iconifie"]];

		[[[rootMenu at: 2 putStr: "Change Buttons"]
				center]
				idAction: [MTreeAct sel: "changeButtons"]];

		[[[rootMenu at: 3 putStr: "Load >"]
				center]
				subMENU: entryPad];

		[[[rootMenu at: 4 putStr: "Save >"]
				center]
				subMENU: entryPad1];

		[[[rootMenu at: 5 putStr: "End At >"]
				center]
				subMENU: entryPad3];

		[[[rootMenu at: 6 putStr: "Begin At >"]
				center]
				subMENU: entryPad2];
		}

  return VideoMTree;
  }


=:
```

Appendix 2 : OSI on X-25 application listing

In this appendix, we give the listing of the OSI on X-25 application. It encompasses the following classes:

- 'Osi1';
- 'Osi1MTree';
- 'OsiStack';
- 'Layer';
- 'Interface';
- 'Pipe'.

```
# include "objc.h"
# include "rmg.h"
# include "envir.h"
# include "stdio.h"

@requires RMGView,RMGAction,Osi1MTree,RMGString, RMGString, FontMngr,Fixtur17,
                Ghost, OrdCltn, OsiStack, RMGCircle, RMGEllipse, RMGLine;
```

```
id str_appAction;

static id actionStep  = NULL;
static id actionSelf  = NULL;
static id idSource    = NULL;
static id superAction = NULL;

#define BUTTON_LINE_NO     3
#define dStrOka             " Disconnection of "
#define dStrOkb             " connection for layer "
#define dStrOkc             " entities, by layer "
#define dStrOkd             " entities ? "

/*********************************************************************
* Class created for the OSI on X-25 application                      *
* By: V ronique Nachtergaele and Dominique de Paul                    *
* This is the master-class of the application which manages it all   *
*********************************************************************/

= Osi1: Envir(osi,working,RMGVW,Primitive,Collection)
  {
  id topBanner, stackColl; /* contain the address of the title bar and
                              of the collection containing the addresses
                              of the four stacks present in the application */
  id sourceStack, destStack, idStack; /* contain the addresses of the stacks
                                         involved ina connection opening or
                                         closing                           */
  id textWin, x25Net, x25str, inTextWin1, inTextWin2; /* contain the addresses
                              of the different views composing the message box
                              and of the different views composing the X-25
                              ellipse                                         */
  id aFont, aBigFont; /* contain the addresses of the two fonts used in this
                         application                                          */

  id aLine2L, aLine1L, aLine3L, aLine1R, aLine2R, aLine3R; /* contain the addres
                         of the lines placed at the bottom of the window and
                         representing the links between the stacks

  id stepButton, idButtonOk, idButtonNot; /*contain the addresses of the buttons
                         used by the user to jump from one step to another
                         or to reply to a question.

  int sourceLayer, destLayer; /* contain the addresses of the two layers
                         between which a connection has to be opened or closed *
  int choice;
  BOOL okStep;
  BOOL okDiscCon, response;
  BOOL stopDisCon;
  }

static id ac_self(aView) id aView;
      /***************************************************************
      * function attached to the application's window which enables  *
      * this window to be the first in the hierarchy of windows present*
      * on the screen.                                               *
      ***************************************************************/

        {
        if (([aView covered]) || ([aView hidden]))
           {
           [aView popToTop];
           }
        }

static id ac_StepC(aView) id aView;
        /**************************************************************
        * Enables the user to pass to the next step of the connection  *
        * opening simulation.                                         *
        **************************************************************/

         {
         id bView;
         int cont;

         [[aView bkgd: WHITE] redraw];
         bView= [aView viewIcon];
         cont = [bView openConStep];

         if ( cont == 2 )
           {
             cont = [bView openConStep];
           }
         [[aView bkgd: GREY_10] redraw];
         if ( cont == 0)
           {
           [aView hide];
           [aView freeAll];
           actionStep = NULL;
           }
          }
```

```
        }

static id ac_StepD(aView) id aView;
  /****************************************************************
  * Enables the user to pass to the next step of a disconnection     *
  * simulation.  If the concerned layer is inferior to 5, it displays*
  * a question in the message box, of type: 'Disconnection of        *
  * transport connection by layer 4 entities, between layer 5        *
  * entities ?'. After that, it creates the 'Yes' and 'No' buttons   *
  * and sets the application into the active collection in order to *
  * enable the user to reply to the question by clicking on one of   *
  * them.                                                            *
  ****************************************************************/

   {
        id bView, cView;
        int cont, i, j;
        int idCurLayer;

        char curLayChar[2];
        char curLayStr[256];
        char curLayStrB[256];

        BOOL okOther = FALSE;


       bView = [aView viewIcon];
       cView = [bView superview];

       if ( cView->okStep )
       {
           bView= [aView viewIcon];
           idCurLayer = [bView askCurLayer];
           if ( idCurLayer >= 5 )
           {
               cont = [bView discStepUp];
           }
           else
           {
               if (( idCurLayer == 0 ) && ([bView askSkip]) == FALSE))
               {
                   cont = 0;
               }

               if ((( idCurLayer == 0) && ([bView askSkip]) == TRUE)) || (idCur
                 {
                 cView = [bView superview];

                 [cView setChoice: 3];

                 cView->okStep = FALSE;

                 strcpy(curLayStr, dStrOka);
                 if (idCurLayer < 3)
                               {
                               if (([bView askSkip]) == FALSE)
                                 {
                                 if (([bView viewLeft]) < ([[bView askEntry
                                  {
                                  if (([bView askStackType]) == 3)
                                     {
                                     strcat(curLayStr, "Right ");
                                     }
                                     else
                                     {
                                     strcat(curLayStr, "Left ");
                                     }
                                  }
                                  else
                                  {
                                  if (([bView askStackType]) == 3)
                                     {
                                     strcat(curLayStr, "Left ");
                                     }
                                     else
                                     {
                                     strcat(curLayStr, "Right ");
                                     }
                                 }
                                 strcat(curLayStr, [[bView askIdNameColl] a
                                 }
                                 else
                                 {
                                 if (([[[bView askDestiStack] askEntryNode]
                                  {
                                  if (([[[bView askDestiStack] askEntryNode
                                     {
                                     strcat(curLayStr, "Right ");
                                     }
                                     else
                                     {
```

```
                                    strcat(curLayStr, "Left ");
                                    }
                                 }
                                 else
                                 {
                                 if (([[[bView askDestiStack] askEntryNode
                                    {
                                    strcat(curLayStr, "Left ");
                                    }
                                    else
                                    {
                                    strcat(curLayStr, "Right ");
                                    }
                                 }
                                strcat(curLayStr, [[bView askIdNameColl] a
                                }
                              }
                              else
                              {
                              strcat(curLayStr, [[bView askIdNameColl] at:
                              }

        strcat(curLayStr, dStrOkb);
        if (([bView askSkip]) == FALSE)
           {
           strcat(curLayStr,(itoa((idCurLayer+1), curLayChar)));
           }
           else
           {
           strcat(curLayStr,(itoa((idCurLayer+2),curLayChar)));
           }
        strcpy(curLayStrB,dStrOkc);
        if (([bView askSkip]) == FALSE)
             {
             strcat(curLayStrB,(itoa(idCurLayer, curLayChar)));
             }
             else
             {
             strcat(curLayStrB,(itoa((idCurLayer+1),curLayChar)));
             }
        strcat(curLayStrB,dStrOkd);

        [[[[[cView->inTextWin2 string: curLayStr]
                                   color: WHITE]
                                   center]
                                   centerV]
                                   show];
        [[[[[[cView->inTextWin1 string: curLayStrB]
                                   color: WHITE]
                                   center]
                                   centerV]
                                   redraw]
                                   show];

         cView->idButtonOk = [[[[[[[[[RMGString relative: 170: 70 exten
                                              superview: cView
                                              bkgd: GREY_10]
                                              frameWidth: 2]
                                              frameColor: BLACK]
                                              string: " Yes "]
                                              font: cView->aFont]
                                              color: BLACK]
                                              center]
                                              centerV]
                                              show];

           [cView buttonFace: cView->idButtonOk color1: GREY_2 color2:

           cView->idButtonNot = [[[[[[[[[RMGString relative: 170: 20 ex
                                              superview: cView
                                              bkgd: GREY_10]
                                              frameWidth: 2]
                                              frameColor: BLACK]
                                              string: " No "]
                                              font: cView->aFont]
                                              color: BLACK]
                                              center]
                                              centerV]
                                              show];

           [cView buttonFace: cView->idButtonNot color1: GREY_2 color2:

           idSource = bView;
           [cView installActive: cView];

        }

    }

  if ( cont == 0)
    { // printf(" fin de decon. ");
         [aView hide];
```

```
                [aView freeAll];
                actionStep = NULL;
                for (i = 0; i < 4; i++)
                {
                   [[cView->stackColl at: i] setWorking: FALSE];

                }
                [[cView->inTextWin2 string: ""]
                                show];
                [[[cView->inTextWin1 string: ""]
                                show]
                                redraw];
            }
        }

       }

static id superAct(aView) id aView;
    /*****************************************************************
    * This function enables to execute the action linked to the       *
    * superview of the view passed as argument.                      *
    *****************************************************************/

      {
      [aView->superview action];
      return aView;
      }

+ newIn : anEnvir
        /*************************************************************
        * Designed to create the application's window and all its    *
        * subviews.  It objectifies also the functions to be linked to *
        * these subviews and intializes some variables.            . *
        *************************************************************/

         {
         int i;
         id nameColl;

         [self delayInit];

         if (!actionSelf)
            {
            actionSelf = [RMGAction newFuncAction: ac_self];
            }

         self = [self origin:100:100 extent:1200:600 superview:anEnvir bkgd: PERI

         [[[self frameWidth: 2]
                frameColor: CYAN]
                idAction: actionSelf];

         aFont = [FontMngr lp10x20_b];
         aBigFont = [FontMngr cour12x20];
         okStep = TRUE;
         okDiscCon = FALSE;
         response = FALSE;

         topBanner=[[[[[[[[RMGString Type: StickTop
                                extent: 17
                                superview: self
                                bkgd: DARKBLUE]
                                string: " OSI on X-25"]
                                font: aFont]
                                color: WHITE]
                                horFill: TRUE]
                                center]
                                centerV]
                                idAction: moveWithBandAction]
                                show];

         if (!superAction)
            superAction = [RMGAction newFuncAction: superAct];

         x25Net = [RMGEllipse  relative: 550: 50
                              extent: 400: 500
                              superview: self
                              bkgd: BLUE];


         [[[[x25Net  xRadius: 190 yRadius: 110]
                     origin: 150: 210]
                     circleSolid: TRUE]
                     color: WHITE]
                     show];

         x25str = [[[[[[[RMGString relative:645:225 extent:100:70 superview:self
                            string:" X 25 "]
                            font:[FontMngr pica18x30]]
                            color:BLACK]
                            center]
                            centerV]
                            idAction: actionSelf]
```

```
                        show];

        stackColl = [OrdCltn new];
        nameColl  = [OrdCltn with: 8,"Physical ->","Data ->","Network ->",
                                    "Transport ->","Session ->","Presentation -
                                    "SASE ->"];

        [OsiStack new: self
                  at: 50: 200
                  type: 7
                  strings: nameColl];

        [stackColl add: [OsiStack new: self
                             at: 250: 200
                             type: 7
                             multiplex: 0: 0]];
        [stackColl add: [OsiStack new: self
                              at: 425: 200
                              type: 3
                              multiplex: 0: 0]];
        [stackColl add: [OsiStack new: self
                              at: 1025: 200
                              type: 7
                              multiplex: 0: 0]];
        [stackColl add: [OsiStack new: self
                              at: 850: 200
                              type: 3
                              multiplex: 0: 0]];

        [[stackColl at: 0] sEntryNode: [stackColl at: 1]];
        [[stackColl at: 2] sEntryNode: [stackColl at: 3]];
        [[stackColl at: 1] sEntryNode: [stackColl at: 0]];
        [[stackColl at: 3] sEntryNode: [stackColl at: 2]];

        textWin = [RMGView relative: 450: 25 extent: 700: 100 superview: self bk
        [[[textWin frameWidth: 1]
                   frameColor: CYAN]
                   show];

        inTextWin1 = [RMGString relative: 0:0 extent: 700: 50 superview: textWin
        [[[[[[inTextWin1 string: ""]
                  font: aBigFont]
                  color: WHITE]
                  center]
                  centerV]
                  show];
        inTextWin2 = [RMGString relative: 0:50 extent: 700: 50 superview: textWi
        [[[[[[inTextWin2 string: ""]
                  font: aBigFont]
                  color: WHITE]
                  center]
                  centerV]
                  show];

        for(i = 0; i < 4; i++)
              {
              [[stackColl at: i] setTextWin: inTextWin1: inTextWin2];
              }

        aLine1L = [[RMGView relative: 310: 199
                             extent: 4: -24
                             superview: self
                             bkgd: BLACK]
                             show];
        aLine3L = [[RMGView relative: 484: 175
                            extent: 4: 24
                            superview: self
                            bkgd: BLACK]
                            show];
        aLine1R = [[RMGView relative: 910: 199
                            extent: 4: -24
                            superview: self
                            bkgd: BLACK]
                             show];
        aLine3R = [[RMGView relative: 1084: 175
                            extent: 4: 24
                            superview: self
                            bkgd: BLACK]
                             show];

        return self;
        }

- extraNewIn: anEnvir
       /****************************************************************
       * Gets the address of the application's menu and creates the   *
       * Ghost instance in order to enable the user to iconize the    *
       * application.                                                 *
       ****************************************************************/

        {
        menuTree=[Osi1MTree getIt];
```

```
        [self showAll];

        viewIcon = [[Ghost stringGhost:" OSI " In: anEnvir]
                        viewIcon:self];

        return self;
        }

+ delayInit
/*********************************************************************
 * Initializes the application's menu                                *
 *********************************************************************/

 {
 static BOOL beenhere = FALSE;

 if(!beenhere)
   if(self == Osi1)
     {
     [Osi1MTree delayInit];
     beenhere = TRUE;
     }
   else [Osi1 delayInit];

 return self;
 }

- iconifie
/*********************************************************************
 * Calls the iconize method in Osi1 superclass, in order to iconize the*
 * application.                                                      *
 *********************************************************************/

 {
        [self iconize];

        return self;
 }

- startDiscStep
/*********************************************************************
 * Sets the application in the active collection if a simulation is   *
 * not currently performed and if a connection is already established *
 * This is done in order to enable the user to select the source layer*
 * from which a disconnection is initiated.                          *
 *********************************************************************/

 {
 int i;
 BOOL work = FALSE;
 BOOL conDone = FALSE;

 choice = 12;
 for (i = 0; i < 4; i++)
     {
     if (([[stackColl at: i] working]) == TRUE)
         work = TRUE;
     }

 if (work == FALSE)
    {
    i = 0;
    while ((i < 4) && (conDone == FALSE))
        {
        conDone = [[stackColl at: i] connectionDone];
        i += 1;
        }

    if (conDone == TRUE)
       {
       [self startRun];
       }
       else
       {
       [[[[[inTextWin1 string: ""]
              color: WHITE]
              center]
              centerV]
              redraw]
              show];
       [[[[inTextWin2 string: " Nothing is Connected, Deconnection impossible
                          color: RED]
                          center]
                          centerV]
                          show];
       }
    }
 return self;
 }

- startCon
/*********************************************************************
 * Sets the application into the active collection if a simulation is *
```

```
* not currently performed and if a connection is possible.  This is   *
* done in order to enable the user to select the source and destination*
* layers from which respectively a connection is initiated and         *
* responded to.                                                        *
***********************************************************************/

 {
 int i;
 BOOL work = FALSE;
 BOOL deConDone = FALSE;

 choice = 21;

 for (i = 0; i < 4; i++)
     {
     if (([[stackColl at: i] working]) == TRUE)
         work = TRUE;
     }

 if (work == FALSE)
    {
    i = 0;
    while ((i < 4) && (deConDone == FALSE))
        {
        deConDone = [[stackColl at: i] deConnectionDone];
        i += 1;
        }

    if (deConDone == TRUE)
       {
       [self startRun];
       }
       else
       {
       [[[[[[inTextWin1 string: ""]
              color: WHITE]
              center]
              centerV]
              redraw]
              show];
       [[[[[inTextWin2 string: " Nothing is Disconnected, Connection impossible
                             color: RED]
                             center]
                             centerV]
                             show];
       }
    }
 return self;
 }

- startConStep
/***********************************************************************
* Is the same as startCon but it concerns the step by step mode.      *
***********************************************************************/

 {
 int i;
 BOOL work = FALSE;
 BOOL deConDone = FALSE;

 choice = 22;

 for (i = 0; i < 4; i++)
     {
     if (([[stackColl at: i] working]) == TRUE)
         work = TRUE;
     }

 if (work == FALSE)
    {
    i = 0;
    while ((i < 4) && (deConDone == FALSE))
        {
        deConDone = [[stackColl at: i] deConnectionDone];
        i += 1;
        }

    if (deConDone == TRUE)
       {
       [self startRun];
       }
       else
       {
       [[[[[[inTextWin1 string: ""]
              color: WHITE]
              center]
              centerV]
              redraw]
              show];
       [[[[[inTextWin2 string: " Nothing is Disconnected, Connection impossible
                             color: RED]
```

```
                        center]
                        centerV]
                        show];
        }
      }
  return self;
  }

- startRun
/*********************************************************************
* Displays the string 'Chose the Source Layer' into the message box    *
* and installs the application into the active collection.             *
*********************************************************************/

  {
  [[[[[inTextWin2 string: " Choose the source LAYER "]
              color: WHITE]
              center]
              centerV]
              show];
  [[[[[[inTextWin1 string: ""]
              color: WHITE]
              center]
              centerV]
              redraw]
              show];


  [self installActive: self];

  return self;
  }

- stopRun
/*********************************************************************
* Removes the application from the active collection                  *
*********************************************************************/

  {
  [self deleteActive: self];

  return self;
  }

- update
/*********************************************************************
* Is the method executed when the application is into the active      *
* collection.                                                          *
*********************************************************************/

  {
  switch (choice)
          {
          case 11: case 12:
                  [self selectOne];
                  break;

          case 21: case 22:
                  [self selectTwo];
                  break;
          case 3:
                  [self selectChoiceD];
                  break;

          }
  return self;
  }

- selectTwo
/*********************************************************************
* Enables the user to select the source and destination layers for a  *
* connection opening. This is done by controlling the mouse buttons    *
* state and the position on the RMG cursor on the screen.              *
*********************************************************************/

  {
  int i,j;
  BOOL found = FALSE;

  if ([[Mouse change: &changex: &changey] getButtons] == 1)
              {
              i = 0;

              while ((i < 4) && (found == FALSE))
                    {
                    j = [[stackColl at: i] layerContains: crossx: crossy];
                    if (j != 0)
                         {
                         if (sourceStack == NULL)
                                              {
                                              sourceStack = [stackColl at: i];
                                              sourceLayer = j;
                                              [[[[[[inTextWin1 string: ""]
```

```
                                                        color: WHITE]
                                                        center]
                                                        centerV]
                                                        redraw]
                                                        show];
                                        [[[[inTextWin2 string: " Choose the d
                                                    color: WHITE]
                                                    center]
                                                    centerV]
                                                    show];
                                        }
                                        else
                                        {
                                        destStack = [stackColl at: i];
                                        destLayer = j;
                                        }
                          found = TRUE;
                          }
                          else
                          {
                          i += 1;
                          }
                     }
                }
 if (destStack != NULL)
                 {
                 [[inTextWin2 string: ""]
                          show];

                 [[self stopRun]
                        connect];

                 [self setChoice: 0];
                 }
 return self;
 }

- selectOne
/**********************************************************************
* Enables the user to select the source layer for the closing of a    *
* connection.                                                         *
**********************************************************************/

 {
 int i,j;
 BOOL found = FALSE;

 if ([[Mouse change: &changex: &changey] getButtons] == 1)
                {
                i = 0;

                while ((i < 4) && (found == FALSE))
                     {
                     j = [[stackColl at: i] layerContains: crossx: crossy];
                     if (j != 0)
                         {
                         sourceStack = [stackColl at: i];
                         sourceLayer = j;
                         found = TRUE;
                         }
                         else
                         {
                         i += 1;
                         }
                     }
                }
  if (sourceStack != NULL)
                    {
                    [[inTextWin2 string: ""]
                             show];

                    stopDisCon = FALSE;

                    [[self stopRun]
                           disconnect];

                    [self setChoice: 0];
                    }
  return self;
  }

- selectChoiceD
/**********************************************************************
* Is the method enabling the user to click on one of the buttons 'Yes'*
* or 'No' when he is prompted to reply a question concerning a        *
* disconnection.                                                      *
**********************************************************************/

  {
       int i, j;
       int cont;
       BOOL found = FALSE, okOther = FALSE;
```

```
	if ([[Mouse change: &changex: &changey] getButtons] == 1)
	{
		okDiscCon = [idButtonOk contains: crossx: crossy];
		response = [idButtonNot contains: crossx: crossy];
	}

	if ( okDiscCon )
	{
		[self setChoice: 0];
		okStep = TRUE;

		[idButtonOk hide];
		[idButtonOk freeAll];
		[idButtonNot hide];
		[idButtonNot freeAll];
		[[inTextWin2 string: ""]
				show];
		[self stopRun];

		cont = [idSource discStepLowOk];

		if (cont == 0)
		  {
		    for (i = 0; i < 4; i++)
		    {
			[[stackColl at: i] setWorking: FALSE];

		    }
		  [stepButton hide];
		  [stepButton freeAll];
		  actionStep = NULL;
		  response = FALSE;
		  }

		okDiscCon = FALSE;
	}

	if ( response )
	{
		[self setChoice: 0];
		okStep = TRUE;

		[idButtonOk hide];
		[idButtonOk freeAll];
		[idButtonNot hide];
		[idButtonNot freeAll];
		[[inTextWin2 string: ""]
				show];
		[[[inTextWin1  string: ""]
				redraw]
				show];

		    [self setChoice: 0];
		    okStep = TRUE;
		    for (i = 0; i < 4; i++)
		    {
			[[stackColl at: i] setWorking: FALSE];

		    }
		    [self stopRun];
		    [stepButton hide];
		    [stepButton freeAll];
		    actionStep = NULL;

		j = [idSource askCurLayer];
	//	printf(" valeur de j: %d", j);

		okOther =  [[idSource askDestiStack] testConnect];


		if ( ( j == 1 || j == 2 ) && ( okOther) && (!stopDisCon ) )
		{ //  printf("chgt de source... ");
		      stopDisCon = TRUE;
		      idStack = idSource;
		      //sourceStack = [idSource askDestiStack];
		      //sourceLayer = [sourceStack askCurLayer];
		      //[sourceStack  setDestiStack: [sourceStack askEntryNode] ];
		      //[sourceStack  discStepLowOk];
		}
		else
		{
		    actionStep = NULL;

		}

		response = FALSE;
	}

   return self;
}
```

```
- connect
/***********************************************************************
* Tests if a connection is possible between the two chosen layers and *
* if it is possible to initiate a connection from this layer level.    *
* If the connection is impossible, a warning message appears in the    *
* message box.  If a connection is possible and that the step by step *
* mode was requested, the 'Continue' button is created and the         *
* simulation is launched.  If a connection is possible and that the    *
* continuous mode was requested, the source stack is put into the      *
* active collection in oredr to enable the simulation to be carried on*
* without any user-application interactions.                          *
***********************************************************************/

 {
 int i;
 if (( sourceLayer == destLayer )
      && (( sourceLayer != 1 ) && ( sourceLayer != 5 ) && ( sourceLayer != 6 ))
      && ((( [sourceStack testConnect] == FALSE ) &&
       (( sourceLayer > 7 )  || (( sourceLayer <= 7 ) && (( [sourceStack testCo
               {
               if (((sourceStack == [stackColl at: 0]) &&
                                 (( (destStack == [stackColl at: 1]) && (d
                                 || ((destStack == [stackColl at: 2])
                                              && ( destLayer >= 3))))
                  || ((sourceStack == [stackColl at: 3]) &&
                                   ((destStack == [stackColl at: 2]) && (d
                        {
                        for(i = 0;i < 4; i++)
                           {
                           [[stackColl at: i] setAction: 1];
                           }
                        [sourceStack setDestiStack: destStack];
                        [sourceStack setWorking: TRUE];
                        if (choice == 21)
                           {
                           [self installActive: sourceStack];
                           }
                           else
                           {
                           if (!actionStep)
                              {
                              actionStep = [RMGAction newFuncAction: ac_StepC
                              }
                           stepButton = [[[[[[[[[[RMGString relative: 50: 50
                                                  superview: self
                                                  bkgd: GREY_10]
                                                  frameWidth: 2]
                                                  frameColor: BLACK]
                                                  string: " Continue "]
                                                  font: aFont]
                                                  color: BLACK]
                                                  viewIcon: sourceStack
                                                  center]
                                                  centerV]
                                                  idAction: actionStep]
                                                  show];
                           [self buttonFace: stepButton color1: GREY_2 color2

                           [sourceStack openConStep];
                           }
                        }
                  else
                  {
                  if (((sourceStack == [stackColl at: 2]) &&
                                  ((destStack == [stackColl at: 3]) &&
                                  || ((destStack == [stackColl at: 0])
                                               && (destLayer >= 3))))
                     || ((sourceStack == [stackColl at: 1]) &&
                                    ((destStack == [stackColl at: 0]) &&
                           {
                           for (i = 0; i < 4; i++)
                              {
                              [[stackColl at: i] setAction: 1];
                              }
                           [sourceStack setDestiStack: destStack];
                           [sourceStack setWorking: TRUE];
                           if (choice == 21)
                              {
                              [self installActive: sourceStack];
                              }
                              else
                              {
                              if (!actionStep)
                                 {
                                 actionStep = [RMGAction newFuncAction: ac_St
                                 }
                              stepButton = [[[[[[[[[[RMGString relative: 50:
                                                     superview: self
                                                     bkgd: GREY_10]
                                                     frameWidth: 2]
                                                     frameColor: BLACK]
```

```
                                                                string: " Continue "]
                                                                font: aFont]
                                                                color: BLACK]
                                                                viewIcon: sourceStack
                                                                center]
                                                                centerV]
                                                                idAction: actionStep]
                                                                show];
                                    [self buttonFace: stepButton color1: GREY_2 col

                                    [sourceStack openConStep];
                                    }
                                }
                        else
                        {
                        [sourceStack resetFrame];
                        [destStack resetFrame];
                        [[[[[[inTextWin1 string: ""]
                                        color: WHITE]
                                        center]
                                        centerV]
                                        redraw]
                                        show];
                        [[[[[inTextWin2 string: " These layers can not be CONNEC
                                    color: RED]
                                    center]
                                    centerV]
                                    show];
                        }
                    }
                }
                else
                {
                [sourceStack resetFrame];
                [destStack resetFrame];
                [[[[[[inTextWin1 string: ""]
                                color: WHITE]
                                center]
                                centerV]
                                redraw]
                                show];
                [[[[[inTextWin2 string: " These layers can not be CONNECTED "]
                            color: RED]
                            center]
                            centerV]
                            show];
                }

  sourceLayer = 0;
  sourceStack = NULL;

  destLayer = 0;
  destStack = NULL;

  return self;
  }

- disconnect
/*****************************************************************
* Test if a disconnection is possible from the selected layer.  If it*
* is impossible, a message appears in the message box.  If the       *
* disconnection is possible, the 'Continue' button is created and the*
* simulation is lauched.                                             *
*****************************************************************/

  {
   int i;

  if (((sourceStack == [stackColl at: 0]) && (sourceLayer > 3)) || (sourceStack
       {
       destStack = [stackColl at: 2];
       }
       else
       {
       if ((sourceStack == [stackColl at: 0]) && (sourceLayer < 4))
           {
           destStack = [stackColl at: 2];
           }
       }
  if ((( sourceStack == [stackColl at: 2]) && (sourceLayer > 3)) || (sourceStack
       {
       destStack = [stackColl at: 0];
       }
       else
       {
       if ((sourceStack == [stackColl at: 2]) && (sourceLayer < 4))
          {
          destStack = [stackColl at: 0];
          }
       }
```

```
if  ( ([sourceStack testConnect] == TRUE) &&
      ( (sourceLayer > 7)  || ( (sourceLayer <= 7) && (([sourceStack testConUp
     {
     for (i = 0; i< 4; i++)
                                   {
                                   [[stackColl at: i] setAction: 2];
                                   }
     [destStack setCurLayer: sourceLayer];
     [destStack setWorkLayer: sourceLayer];
     [sourceStack setDestiStack: destStack];
     [sourceStack setWorking: TRUE];


     if (choice == 11)
        {
        [sourceStack resetFrame];
        [self installActive: sourceStack];
        }
        else
        {
        [sourceStack resetFrame];
        if (!actionStep)
           {
           actionStep = [RMGAction newFuncAction: ac_StepD];
           }
           stepButton = [[[[[[[[[[RMGString relative: 50: 50 extent: 100: 30
                                                        superview: self
                                                       bkgd: GREY_10]
                                                       frameWidth: 2]
                                                       frameColor: BLACK]
                                                       string: " Continue "]
                                                       font: aFont]
                                                       color: BLACK]
                                                       viewIcon: sourceStack
                                                       center]
                                                       centerV]
                                                       idAction: actionStep]
                                                       show];
          [self buttonFace: stepButton color1: GREY_2 color2: GREY_15];

           [sourceStack openDiscStep];


        }
     }
     else
     {
     [sourceStack resetFrame];
     [destStack resetFrame];
     [[[[[[inTextWin1 string: ""]
                      color: WHITE]
                      center]
                      centerV]
                      redraw]
                      show];
     [[[[[inTextWin2 string: " These layers can not be DISCONNECTED "]
                 color: RED]
                 center]
                 centerV]
                 show];
     }

 sourceLayer = 0;
 sourceStack = NULL;

 destLayer = 0;
 destStack = NULL;

 return self;
 }

- setChoice: (int)aValue
/*********************************************************************
* Sets the choice instance variable to aValue                        *
*********************************************************************/

 {
 choice = aValue;

 return self;
 }

- buttonFace: aView color1: (int)aColor color2: (int)bColor
/*********************************************************************
* Creates shaded buttons on the screen.  The first parmeter is the   *
* address of the view concerned by the 'lifting'.  The two other     *
* parameters are the identifiers of the colors which will surround the*
* the considered view.
*********************************************************************/

 {
 register int i;
```

```
for (i = 0; i < BUTTON_LINE_NO; i++)
     {
     [[[RMGLine superview: aView color: aColor]
                relP1: 0: i length: [aView in_width]: 0]
                show];
     [[[RMGLine superview: aView color: aColor]
                relP1: [aView in_width]-1-i: 0 length: 0: [aView in_height]]
                show];
     }
for ( i = 0; i < BUTTON_LINE_NO; i++)
     {
     [[[RMGLine superview: aView color: bColor]
                relP1: 0: [aView in_height]-i length: [aView in_width]-i: 0]
                show];
     [[[[RMGLine superview: aView color: bColor]
                relP1: i: i length: 0: [aView in_height]]
                idAction: superAction]
                show];
     }
return self;
}

=:
```

```
#include "objc.h"
#include "rmg.h"
#include "envir.h"


@requires FontMngr,RMGMenu1,MTreeAct;


static id onlyInstance;

/*********************************************************************
* Class created for the OSI on X-25 application                      *
* By: V ronique Nachtergaele and Dominique de Paul                   *
* Its goal is to create an instance of the application's menu        *
*********************************************************************/

=Osi1MTree: EnvMTree(osi1,working,RMGVW,Collection,Primitive)


+ getIt
 /********************************************************************
 * Returns the address of the only instance of the application's menu  *
*******************************************************************/

 {
 return onlyInstance;
 }



+ delayInit
/*******************************************************************
* Creates the application's menu having 4 items.  '!! QUIT !!' quits  *
* the application. 'Icon' iconizes the application. 'Connection'      *
* starts a connection either in continuous or step by step mode (this *
* is chosen in its sub menu). 'Disconnection' enables to start a      *
* disconnection in step by step mode.                                 *
*******************************************************************/

 {
 id aFont;
 id sub_Menu1, sub_Menu2;
 static BOOL beenhere = FALSE;


 if(!beenhere)
   {
   beenhere = TRUE;
   aFont = [FontMngr  sysfont];
   self = onlyInstance = [self  new];

   sub_Menu1 = [[[RMGMenu1 char_wide: 16 items: 2 font: aFont]
                 bkgd: GOLD]
                 APPL: self];

   [[[sub_Menu1 at: 0 putStr: "Step by Step"]
               center]
               idAction: [MTreeAct sel: "startConStep"]];
   [[[sub_Menu1 at: 1 putStr: "Continuous"]
               center]
               idAction: [MTreeAct sel: "startCon"]];

   rootMenu = [[[RMGMenu1  char_wide: 15  items: 4  font:aFont]
                          bkgd: GOLD]
                          APPL:self];
   [[[[rootMenu  at:0  putStr:"!!Quit!!"]
                 center]
                 subMENU: Menu_emph]
                 idAction: quitProgAction];
   [[[rootMenu at:1 putStr:"Icon"]
               center]
               idAction: [MTreeAct sel:"iconifie"]];
   [[[[rootMenu at: 2 putStr: "Connection >"]
                     center]
                     subMENU: sub_Menu1]
                     idAction: [MTreeAct sel: "startConStep"]];
   [[[rootMenu at: 3 putStr: "Disconnection "]
                     center]
                     idAction: [MTreeAct sel: "startDiscStep"]];

   }

 return Osi1MTree;
 }

=:
```

```
#include "objc.h"
#include "rmg.h"
#include "envir.h"

@requires RMGView, RMGString, FontMngr, OrdCltn, IntArray, Layer, Interface, RMG

#define aLayerHeight     35
#define aInterfaceHeight 10
#define aWidth           125

#define oStr1a            " Opening "

#define assoStr           " association "
#define connStr           " connection "
#define byStr             " by layer "
#define betwStr           " entities, between layer "
#define entiStr           " entities "

#define oStr2            " opened by layer "

#define cStr1a            " Closing "

#define cStr2             " closed by layer "

#define dStrOk            " Disconnection between layers "

/*******************************************************************
* This class was created for the OSI on X-25 application          *
* By: V ronique Nachtergaele and Dominique de Paul                *
* It enables the creation and management of a stack within this   *
* application. It is not well explained and detailed as it is     *
* still under development.                                        *
*******************************************************************/

= OsiStack: Osi1(osi,working,RMGVW,Primitive,Collection)
{
         id clColorArray, opColorArray; /* contain the addresses of two
                        collections containing the pastel and brilliant
                        colors of the different layers composing a stack */
         id layerColl, interfaceColl, nameColl; /* contain the addresses
                        of collection containing the addresses of each layer
                         composing a stack, of each interface composing a
                         stack and the names of each layer to be displayed
                         at the left of the application's window          */
         id destiStack, entryNode, back, winText1, winText2; /' contain the
                        address of the stack with which the connection opening
```

```
                    or closing has to be made; the address of the entry
                    node stack of a stack; the addresses of various
                    views composing the application's window          */
        id aLink;
        id cllLink, phyLine; /* These three variables contain the addresses
                    of the red rectangle which is created between two
                    stacks involved in a connection opening or closing;
                    of the link between an entry node stack and the X-25
                    ellipse; of the link between a stack and its entry node

        int stackType, currentLayer, [illegible];
        int actionType;

        BOOL down, done, skip, work;
}

+ new: aView at: (int)x: (int)y type: (int)aStack strings: aColl
        /*************************************************************
        * Enables the user to create an instance of OsiStack in which *
        * some strings are displayed. The stack is placed in the view *
        * which address is passed as first argument, at location      *
        * passed as second and third arguments.  The fourth argument  *
        * specifies the number of layers needed.  The last argument   *
        * is the address of the collection containing the strings to *
        * display into the stack.                                     *
        *************************************************************/

        {
        int i;

        for (i = 0;i <= aStack;i++)
            {
            [[[[[[[RMGString relative: x: y extent: 150: 35 superview: aView bk
                        string: [aColl at: i]]
                        font: [FontMngr lp10x20_b]]
                        color: WHITE]
                        center]
                        centerV]
                        show];
            if (i == 6)
                {
                y = y + 35;
                }
                else
                {
                y = y + 45;
                }
            }
        }
+ new: aView at: (int)x: (int)y type: (int)aStack multiplex: (int)aLevel: (int)b
        /*************************************************************
        * Enables the user to create an ordinary instance of OsiStack*
        * This stack is placed in the view which address is passed as*
        * first argument, at the location identified by the second   *
        * and third arguments.  The fourth argument is the type of   *
        * stack needed, thus it specifies the number of layers       *
        * desired.  The two last arguments are designed to pass the  *
        * layer numbers between which the multiplexing is possible   *
        *************************************************************/

        {
        int i, xpos, ypos, stackHeight;

        //layerColl = [OrdCltn new];



        if (aStack == 7)
                {
                stackHeight = 8;
                }
                else
                {
                stackHeight = 3;
                }

        xpos = 0;
        ypos = 0;
        self = [self relative: x: y
                        extent: aWidth: ((stackHeight * aLayerHeight) + ((aSt
                        superview: aView
                        bkgd: BLACK];

        down = TRUE;
        skip = FALSE;
        done = FALSE;
        work = FALSE;

        layerColl = [OrdCltn new];
        interfaceColl = [OrdCltn new];

        if (aStack == 7)
                {
```

```
                stackType = 2;
                }
                else
                {
                stackType = aStack;
                }

        [self setCurLayer: 0];
        [self loadColor];

        for(i = 0; i < aStack; i++)
                {
                [layerColl add: [Layer new: self
                                        at: xpos: ypos
                                        color1: [clColorArray intAt: i]
                                        color2: [opColorArray intAt: i]
                                        height: aLayerHeight
                                        width: aWidth]];
                ypos = ypos + aLayerHeight;

                if(i < (aStack - 1))
                     {
                     [interfaceColl add: [Interface new: self
                                                    at: xpos: ypos
                                                    color: DARKBLUE
                                                    height: aInterfaceHeight
                                                    width: aWidth]];
                     ypos = ypos + aInterfaceHeight;
                     }
                }
         if (stackType == 8)
            {
            [layerColl add:[Layer new: self
                                 at: xpos: ypos
                                 color1: [clColorArray intAt: 6]
                                 color2: [opColorArray intAt: 6]
                                 height: aLayerHeight
                                 width: aWidth]];
            [interfaceColl add: [Interface new: self
                                        at: xpos: (ypos + aLayerHeight)
                                        color: DARKBLUE
                                        height: aInterfaceHeight
                                        width: aWidth]];
            }
         [self loadLayerNames];

         return self;
         }

- loadColor
          /*****************************************************
          * Builds the instance of the OrdCltn class with the     *
          * different layers colors.  The addresses of these      *
          * instances are put into clColorArray and opColorArray  *
          * instance variables.                                   *
          ******************************************************/

            {
            clColorArray = [IntArray with: 7, BRICKRED, BEIGE, LIGHTYELLOW,
                                            LIGHTGREEN, LIGHTBLUE, PINK, M
            opColorArray = [IntArray with: 7, ORANGE, BURNTORANGE, YELLOW, DA
                                            BLUE, DARKPINK, GREYMAGENTA];

            return self;
            }
- loadLayerNames
          /******************************************************
          * Builds the instance of the OrdCltn class with the names *
          * of the different layers. Its address is placed into the *
          * nameColl instance method.                                *
          ******************************************************/

            {
            nameColl = [OrdCltn with: 8,"Physical","Data","Network",
                                    "Transport","Session","Presentation"
                                    "Application's CASE", "Application's

            return self;
            }

- (int)layerContains: (int)x: (int)y
         /*******************************************************
         * Checks if the location identified by the two values passed *
         * as arguments is inside the bounds of one of the stack's    *
         * layers.  It returns an integer which has the value zero if *
         * the location is not inside the stack; the layer number     *
         * otherwise.                                                 *
         ********************************************************/

            {
            int i;
            BOOL found;

            i = 0;
```

```
                found = FALSE;

                while ((i <= (stackType - 1)) && (found == FALSE))
                        {
                        found = [[layerColl at: i] frameContains: x: y];
                        i += 1;
                        }
                if (found == FALSE)
                        {
                        i = 0;
                        }
                        else
                        {
[[layerColl at: (i-1)] frameColor: RED];
                        [self setCurLayer: i];
                        [self setWorkLayer: i];
                        if (i >= 3)
                                {
                                [entryNode setCurLayer: 3];
                                [entryNode setWorkLayer: 3];
                                }
                                else
                                {
                                [entryNode setCurLayer: i];
                                [entryNode setWorkLayer: i];
                                }
                        }

                return i;
                }

- update
        /**************************************************************
        * Is executed when the instance of the OsiStack class is in the *
        * active collection.                                          *
        **************************************************************/

                {
                switch (actionType)
                        {
                        case 1: [self openConnect];
                                break;
                        }

                return self;
                }


- openConnect
            /*******************************************************
            * Starts the simulation of a connection opening in      *
            * continuous mode.  The first phase concerns the downward *
            * progression of the opening. The second phase concerns  *
            * the upward progression of the opening.                *
            *******************************************************/

                {
                char curLayChar[2];
                char curLayStr[256];
                char curLayStrB[256];
                static int compt = 500;
                static freeLink = FALSE;
                id idCurLayer;
                id destination;
                int curInterface;


               if (compt == 500)
                {compt = 0;
                if (skip == FALSE)
                     {
                     if (down == TRUE)
                          {
                          if (currentLayer >= 1)
                               {
                               idCurLayer = [layerColl at: (currentLayer - 1)];
                               if (currentLayer <= 3)
                                    {
                                    destination = entryNode;
                                    }
                                    else
                                    {
                                    destination = destiStack;
                                    }

                              if (currentLayer == workLayer)
                                 {
                                 aLink = [[[[FMSLink alloc] origin: [self searchXOriginLin
                                                  extent: [self searchWidthLink: de
                                                  superview: [self superview]
                                                  style: [illegible]]
                                                  frameWidth: 1]
                                                  frameColor: RED]
```

```
                            show];
        }
        else
        {
        if ([idCurLayer connectionState] == FALSE)
        {
        [[[aLink move: [self searchXOriginLink:destination
                  size: [self searchWidthLink: destination
                  show];
        [aLink move: [self searchXOriginLink: destination]
        }
        }

     if ([idCurLayer connectionState] == FALSE)
         {
         [idCurLayer flicker];
         if ((currentLayer == 3) && (stackType != 3))
            {
            [[destiStack askEntryNode] flickering];
            }
         [idCurLayer inConnectWith: [destination replyConn

         strcpy(curLayStr, oStria);
         strcat(curLayStr, ([nameColl at:(currentLayer-1)]
         if ((currentLayer == 8) || (currentLayer == 7))
            {
            strcat(curLayStr,assoStr);
            }
            else
            {
            strcat(curLayStr,connStr);
            }
         strcat(curLayStr,byStr);
         if(currentLayer == 8)
            {
            itoa(7 , curLayChar);
            strcat(curLayStr,curLayChar);
            strcpy(curLayStrB," entities, for the user ");
            }
            else
            {
            itoa(currentLayer, curLayChar);
            strcat(curLayStr,curLayChar);
            if (currentLayer < 7)
               {
               strcpy(curLayStrB,betwStr);
               itoa( (currentLayer + 1),curLayChar);
               strcat(curLayStrB,curLayChar);
               strcat(curLayStrB,entiStr);
               }
               else
               {
               strcpy(curLayStrB," for Application Layer's
               }
            }


         [[[[winText2 string: curLayStr]
                      color: WHITE]
                      center]
                      centerV]
                      show];

         [[[[[winText1 string: curLayStrB]
                      color: WHITE]
                      center]
                      centerV]
                      redraw]
                      show];

         [self setCurLayer: (currentLayer - 1)];
         [destination setCurLayer: ([destination askCurLay
         }
       else { down = FALSE; freeLink = TRUE; }
     }
     else
     {
     down = FALSE;
     }
}
else
{
if (currentLayer < workLayer )
     {
     if ((currentLayer == 0)   (freeLink == TRUE ))
        {
        [aLink free];
        freeLink = FALSE;
        }

     [self setCurLayer: (currentLayer + 1 )];
     idCurLayer = [layerColl at: (currentLayer + 1)];
```

```
if (currentLayer <= 3)
     {
     destination = entryNode;
     }
     else
     {
     destination = destiStack;
     }

if ((currentLayer == 3) && (done == FALSE) && (back =
    {
    skip = TRUE;
    [[destiStack askEntryNode] back: self];
    [self installActive: [destiStack askEntryNode]];
    [self setCurLayer: (currentLayer - 1)];
    done = TRUE;
    }
    else
    {
    [destination setCurLayer: ([destination askCurLay

    if ([idCurLayer connectionState] == FALSE)
         {
         if (currentLayer < 3)
         {
           if (([self viewLeft]) < ([destination view
            {
            if (stackType == 3)
               {
               strcpy(curLayStr, " Right ");
               }
               else
               {
               strcpy(curLayStr, " Left ");
               }
            }
            else
            {
            if (stackType == 3)
               {
               strcpy(curLayStr, " Left ");
               }
               else
               {
               strcpy(curLayStr, " Right ");
               }
            }
           strcat(curLayStr,([nameColl at: (currentLa
         }
         else
         {
         strcpy(curLayStr,([nameColl at: (currentLaye
         }

         if ((currentLayer == 8) || (currentLayer ==
            {
            strcat(curLayStr,assoStr);
            }
            else
            {
            strcat(curLayStr,connStr);
            }
         strcat(curLayStr, oStr2);
         if(currentLayer == 8)
          {
          itoa(7 , curLayChar);
          strcat(curLayStr,curLayChar);
         -strcpy(curLayStrB," entities, for the user
          }
          else
          {
          itoa(currentLayer, curLayChar);
          strcat(curLayStr,curLayChar);
          if (currentLayer < 7)
             {
             strcpy(curLayStrB,betwStr);
             itoa( (currentLayer + 1),curLayChar);
             strcat(curLayStrB,curLayChar);
             strcat(curLayStrB,entiStr);
             }
             else
             {
             strcpy(curLayStrB," for Application laye
             }
          }

         [[[[winText2 string: curLayStr]
                 color: WHITE]
                 center]
                 centerV]
                 show];
         [[[[[winText1 string: curLayStrB]
```

```
                color: WHITE]
                center]
                centerV]
                redraw]
                show];

        [[idCurLayer stopRun]
                      changeColor];
        if (currentLayer == 1)
            {
                if (([self viewLeft]) <
                             ([destination viewLeft]))
                  {
                  phyLine = [self createEllLink: ([se
                                length: 176: 4];
                  }
                  else
                  {
                  phyLine = [self createEllLink: ([se
                                length: -176: 5];
                  }
            }

        if (currentLayer == 3)
            {
            if (stackType == 3)
                {
                if (([self viewLeft]) <
                             ([destination viewLeft]))
                  {
                  ellLink = [self createEllLink: ([se
                                length: 70: 5];
                  }
                  else
                  {
                  ellLink = [self createEllLink: ([se
                                length: 70: 5];
                  }
                }
                else
                {
                int lX;

                if (([self viewLeft]) <
                             ([destination viewLeft]))
                  {
                  lX = 55;

                  }
                  else
                  {
                  lX = -5;
                  }
                ellLink = [destination createEllLink:
                                length: 70: 5];
                }
            }

        [idCurLayer createPipe];

        if (((currentLayer != 7) && (stackType == 8)
               (stackType == 3) && (currentLayer != 3
               {
                 if ( currentLayer == 5 )
                    { curInterface = 6;
                    }
                 else
                    { curInterface = (currentLayer -1)
                    }
               [[interfaceList at:curInterface] create
               [illegible] Connect];

          }
    }
if ( currentLayer == [illegible]
        {
        if (back != NULL)
             {
             [back setB[illegible]: FALSE];
             [self setW[illegible]: FALSE];
             [self deleteActive: self];
             [illegible] = NULL;

                   [illegible]
```

```
                [self setAct[illegible]
                done  = FALSE;
                skip = FALSE;
                down = TRUE;
                }
            }
        }
      }
    else compt += 1;

    return self;
    }

- [illegible]
/*[illegible]*****************************************
* Start [illegible] of a connection opening in step by *
* step mode. [illegible] are nearly the same as for *
* openConnect[illegible] completely independent from*
* the active collection.                               *
*****************************[illegible]*****************/

    {
    char curLayChar[2];
    char curLayStr[256];
    char curLayStrB[256];
    static int compt = 500;
    id idCurLayer;
    id destination;

    int cont = 1;
    int curInterface;
    BOOL freeLink = FALSE;

    if (down == TRUE)
            {
            if (currentLayer >= 1)
                {
                idCurLayer = [layerColl at: (currentLayer - 1)];
                if (currentLayer <= 3)
                    {
                    destination = entryNode;
                    }
                    else
                    {
                    destination = destiStack;
                    }

               if (currentLayer == workLayer)
                  {
                  aLink = [[[[RMGView origin: [self searchXOriginLin

                                     extent: [self searchWidthLink: de
                                     superview: [self superview]
                                     bkgd: PERIWINKLE]
                                     frameWidth: 1]
                                     frameColor: RED]
                                     show];
                  }
                  else
                  {
                  [[[aLink move: [self searchXOriginLink:destination
                              size: [self searchWidthLink: destination
                              show];
                  [aLink move: [self searchXOriginLink: destination],
                  }

               if ([idCurLayer connectionState] == FALSE)
                   {
                   [idCurLayer flicker];

                   if ((currentLayer == 3) && (stackType != 3))
                       {
                       [[destiStack askEntryNode] flickering];
                       }

                   [idCurLayer inConnectWith: [destination replyConn

                   strcpy(curLayStr, oStr1a);
                   strcat(curLayStr, ([nameColl at:(currentLayer-1)]
                   if ((currentLayer == 8) || (currentLayer == 7))
                       {
                       strcat(curLayStr,assoStr);
                       }
                       else
                       {
                       strcat(curLayStr,connStr);
                       }
                   strcat(curLayStr,byStr);
                   if(currentLayer == 8)
                       {
                       itoa(7 , curLayChar);
                       strcat(curLayStr,curLayChar);
                       strcpy(curLayStrB," entities, for the user ");
                       }
```

```
                    else
                    {
                    itoa(currentLayer, curLayChar);
                    strcat(curLayStr,curLayChar);
                    if (currentLayer < 7)
                        {
                        strcpy(curLayStr3,betwStr);
                        itoa( (currentLayer + 1),curLayChar);
                        strcat(curLayStr3,curLayChar);
                        strcat(curLayStr3,entiStr);
                        }
                        else
                        {
                        strcpy(curLayStr3, " for Application layer'
                        }
                    }

                [[[[winText2 string: curLayStr]
                             color: WHITE]
                             center]
                             centerV]
                             show];
                [[[[[winText1 string: curLayStr3]
                             color: WHITE]
                             center]
                             centerV]
                             redraw]
                             show];

                [self setCurLayer: (currentLayer - 1)];
                [destination setCurLayer: ([destination askCurLay
                }
              else { down = FALSE;
                     freeLink = TRUE;
                     skip = FALSE; }
             }
             else
             {
             down = FALSE;
             skip = FALSE;
             }
        }
    if (down == FALSE)
        {
        if (currentLayer < workLayer)
            {
            if (currentLayer == 0 || freeLink )
               {
               [aLink free];
               }

            [self setCurLayer: (currentLayer + 1)];
            idCurLayer = [layerColl at: (currentLayer - 1)];

            if (currentLayer <= 3)
                 {
                 destination = entryNode;
                 }
                 else
                 {
                 destination = destiStack;
                 }

            if ((currentLayer == 3) && (skip == FALSE) && (back =
                {
                [[destiStack askEntryNode] back: self];
                cont = [[destiStack askEntryNode] openConStep];
                [self setCurLayer: (currentLayer - 1)];

                }
                else
                {
                [destination setCurLayer: ([destination askCurLay

                if ([idCurLayer connectionState] == FALSE)
                      {
                      if (currentLayer < 3)
                      {
                        if (([self viewLeft]) < ([destination view
                         {
                         if (stackType == 3)
                             {
                             strcpy(curLayStr, " Right ");
                             }
                             else
                             {
                             strcpy(curLayStr, " Left ");
                             }
                         }
                         else
                         {
                         if (stackType == 3)
```

```
        {
        strcpy(curLayStr, " Left ");
        }
        else
        {
        strcpy(curLayStr, " Right ");
        }
    }
  strcat(curLayStr,([nameColl at: (currentLa
}
else
{
strcpy(curLayStr,([nameColl at: (currentLaye
}

if ((currentLayer == 8) || (currentLayer ==
   {
   strcat(curLayStr,assoStr);
   }
   else
   {
   strcat(curLayStr,connStr);
   }
strcat(curLayStr, oStr2);
if(currentLayer == 8)
 {
 itoa(7 , curLayChar);
 strcat(curLayStr,curLayChar);
 strcpy(curLayStrB," entities, for the user
 }
 else
 {
 itoa(currentLayer, curLayChar);
 strcat(curLayStr,curLayChar);
 if (currentLayer < 7)
    {
    strcpy(curLayStrB,betwStr);
    itoa( (currentLayer + 1),curLayChar);
    strcat(curLayStrB,curLayChar);
    strcat(curLayStrB,entiStr);
    }
    else
    {
    strcpy(curLayStrB," for Application laye
    }
 }

[[[[[winText2 string: curLayStr]
        color: WHITE]
        center]
        centerV]
        show];
[[[[[[winText1 string: curLayStrB]
        color: WHITE]
        center]
        centerV]
        redraw]
        show];

[[idCurLayer stopRun]
                changeColor];

if (currentLayer == 1)
   {
      if (([self viewLeft]) <
                  ([destination viewLeft]))
         {
         phyLine = [self createEllLink: ([se
                     length: 176: 4];
         }
         else
         {
         phyLine = [self createEllLink: ([se
                     length: -176: 5];
         }
   }

if (currentLayer == 3)
   {
   if (stackType == 3)
      {
      if (([self viewLeft]) <
                  ([destination viewLeft]))
         {
         ellLink = [self createEllLink: ([se
                     length: 70: 5];
         }
         else
         {
         ellLink = [self createEllLink: ([se
                     length: 70: 5];
         }
      }
      else
```

```
                        {
                        int lX;

                        if (([self viewLeft]) <
                                  ([destination viewLeft]))
                           {
                           lX = 55;

                           }
                           else
                           {
                           lX = -5;
                           }
                        ellLink = [destination createEllLink:
                                  length: 70: 5];
                        }
                     }

                  [idCurLayer createPipe];

                  if (((currentLayer != 7) && (stackType == 8)
                       ((stackType == 3) && (currentLayer != 3
                       {
                         if ( currentLayer == 8 )
                            { curInterface = 6;
                            }
                         else
                            { curInterface = (currentLayer -1)
                            }
                       [[interfaceColl at:curInterface] create
                       }

                  [destination replyConnect];
                  }
               }
            }
         if (currentLayer >= workLayer)
            {
            if (back != NULL)
               {
               [back setSkip: TRUE];
               [self setWorking: FALSE];
               cont = 2;
               back = NULL;
               }
               else
               {
               [self setWorking: FALSE];
               cont = 0;
               }

            [self setAction: 0];
            done  = FALSE;
            skip = FALSE;
            down = TRUE;
            }
      }
      return cont;
      }

- replyConnect
         /*********************************************************
         * Makes an instance of the Layer class flicker or change *
         * its color depending on the case. It returns the address *
         * of the concerned layer.                                *
         *********************************************************/

            {
            id idCurLayer;
            int curInterface;

            idCurLayer = [layerColl at: (currentLayer - 1)];

            if ([idCurLayer flick] == FALSE)
                     {
                     [idCurLayer flicker];
                     }
                     else
                     {
                     [[idCurLayer stopRun]
                                  changeColor];

                     [idCurLayer createPipe];

                     if (((currentLayer != 7) && (stackType == 8))
                              ((stackType == 3) && (currentLayer != 3
                              {
                               if (currentLayer == 8)
                                  { curInterface = 6;
                                  }
                               else
                                  { curInterface = curren
```

```
                                        }
                                [[interfaceColl at:curInterface] create
                                }

                        }

        return idCurLayer;
        }

- (int)openDiscStep
    /*********************************************************
    * Starts the simulation of a disconnection.  If the layers  *
    * >= 5 the disconnection is performed normaly through      *
    * discStepUp.  Otherwise questions have to be asked and it  *
    * is performed through discStepLowOk.                      *
    *********************************************************/

        {
        int cont = 1;

        if ( currentLayer >= 5 )
        {
           cont = [self discStepUp];
        }
        else
        {
          cont = [self discStepLowOk];
        }

        return cont;
        }

- (int)discStepUp
    /*********************************************************
    * Simulates the disconnection of the three top layers of the*
    * ISO reference model.  The first phase concerns the       *
    * downwards progression of the disconnection.  The second  *
    * phase concerns the upward progression of the disconnection*
    *********************************************************/

        {
        char curLayChar[2];
        char curLayStr[256];
        char curLayStrB[256];
        id idCurLayer;
        id destination;
        int curInterface;

        int cont = 1;
        static BOOL freeLink = FALSE;


        if (down == TRUE)
        {
                idCurLayer = [layerColl at: (currentLayer - 1)];
                destination = destiStack;

                if (currentLayer == workLayer)
                {
                    aLink = [[[[RMGView origin: [self searchXOriginLin
                                    extent: [self searchWidthLink: de
                                    superview: [self superview]
                                    bkgd: PERIWINKLE]
                                    frameWidth: 1]
                                    frameColor: RED]
                                    show];
                }
                else
                {
                    [[[[aLink  move: [self searchXOriginLink: destinat
                              move: [self searchXOriginLink: destina
                              size: [self searchWidthLink: destination
                              show];
                    [aLink move: [self searchXOriginLink: destination]
                }

                if ([idCurLayer connectionState] == TRUE)
                {
                   [idCurLayer flicker];

                   [destination replyDisconnect];

                      strcpy(curLayStr, cStr1a);
                      strcat(curLayStr, ([nameColl at:(currentLayer-1)]
                      if ((currentLayer == 8) || (currentLayer == 7))
                         {
                         strcat(curLayStr,assoStr);
                         }
                         else
                         {
                         strcat(curLayStr,connStr);
                         }
                      strcat(curLayStr,byStr);
```

```
            if(currentLayer == 8)
               {
               itoa(7 , curLayChar);
               strcat(curLayStr,curLayChar);
               strcpy(curLayStrB," entities, for the user ");
               }
               else
               {
               itoa(currentLayer, curLayChar);
               strcat(curLayStr,curLayChar);
               if (currentLayer < 7)
                  {
                  strcpy(curLayStrB,betwStr);
                  itoa( (currentLayer + 1),curLayChar);
                  strcat(curLayStrB,curLayChar);
                  strcat(curLayStrB,entiStr);
                  }
                  else
                  {
                  strcpy(curLayStrB," for Application layer's
                  }
               }

         [[[[winText2 string: curLayStr]
                     color: WHITE]
                     center]
                     centerV]
                     show];
         [[[[[winText1 string: curLayStrB]
                         color: WHITE]
                         center]
                         centerV]
                         redraw]
                         show];
      }
      if ( currentLayer > 5 )
      {
      [self setCurLayer: (currentLayer - 1)];
      [destination setCurLayer: ([destination askCurLayer] -
      }
      else
      {
         down = FALSE;
         freeLink = TRUE;

      }
   }
   else
   {
   if (currentLayer < workLayer)
      {
         if (freeLink)
         {
         [aLink free];
         freeLink = FALSE;
         [self setCurLayer: (currentLayer - 1)];
         [destination setCurLayer: ([destination askCurLayer
         }

        [self setCurLayer: (currentLayer + 1)];
        idCurLayer = [layerColl at: (currentLayer - 1)];
        destination = destiStack;
        [destination setCurLayer: ([destination askCurLayer]

        if ([idCurLayer connectionState] == TRUE)
        {
            strcpy(curLayStr,([nameColl at: (currentLayer-1)
            if ((currentLayer == 8) || (currentLayer == 7))
              {
              strcat(curLayStr,assoStr);
              }
              else
              {
              strcat(curLayStr,connStr);
              }
                strcat(curLayStr, cStr2);
                if(currentLayer == 8)
                 {
                 itoa(7 , curLayChar);
                 strcat(curLayStr,curLayChar);
                 strcpy(curLayStrB," entities, for the user
                 }
                 else
                 {
                 itoa(currentLayer, curLayChar);
                 strcat(curLayStr,curLayChar);
                 if (currentLayer < 7)
                    {
                    strcpy(curLayStrB,betwStr);
                    itoa( (currentLayer + 1),curLayChar);
                    strcat(curLayStrB,curLayChar);
                    strcat(curLayStrB,entiStr);
```

```
                        }
                        else
                        {
                        strcpy(curLayStrB," for Application laye
                        }
                    }

                [[[[winText2 string: curLayStr]
                        color: WHITE]
                        center]
                        centerV]
                        show];
                [[[[[winText1 string: curLayStrB]
                        color: WHITE]
                        center]
                        centerV]
                        redraw]
                        show];

                [idCurLayer inConnectWith: NULL];
                [[idCurLayer stopRun]
                         changeColor];

                [idCurLayer deletePipe];

                if ( currentLayer != 7 )
                      {
                       if ( currentLayer == 8 )
                          { curInterface = 6;
                          }
                      else
                          { curInterface = currentLayer - 1;
                          }
                      [[interfaceColl at:curInterface] delete
                      }

                [destination replyDisconnect];
            }
          }
        if (currentLayer >= workLayer)
         {
            cont = 2;
            [self setCurLayer: 4];
            [destination setCurLayer: 4];
            down = TRUE;
         }

         }


    return cont;
  }


- (int)discStepLowOk
    /************************************************************
    * Simulates the disconnection of the four lower laayers of  *
    * the ISo reference model.  This is made by taking into     *
    * account the choices of the user.  The user replies to the *
    * question of type: 'Disconnection of transport connection  *
    * by layer 4 entities, between layer 5 entities ?'.  If the *
    * reply is positive the disconnection keeps on.  If the     *
    * reply is negative, the disconnection stops for the        *
    * concerned part.                                           *
    ************************************************************/

        {
        char curLayChar[2];
        char curLayStr[256];
        char curLayStrB[256];
        id idCurLayer;
        id destination;
        int curInterface;

        int cont = 1;

        if (skip == FALSE)
          {
          if (currentLayer <= 3)
                 {
                 destination = entryNode;
                 }
                 else
                 {
                 destination = destiStack;
                 }
           idCurLayer = [layerColl at: (currentLayer - 1)];

           if ([idCurLayer connectionState] == TRUE)
             {
                     if (currentLayer < 3)
                        {
```

```
        if (([self viewLeft]) < ([destination viewLe
          {
          if (stackType == 3)
             {
             strcpy(curLayStr, " Right ");
             }
             else
             {
             strcpy(curLayStr, " Left ");
             }
          }
          else
          {
          if (stackType == 3)
             {
             strcpy(curLayStr, " Left ");
             }
             else
             {
             strcpy(curLayStr, " Right ");
             }
          }
        strcat(curLayStr,([nameColl at: (currentLaye
        }
        else
        {
        strcpy(curLayStr,([nameColl at: (currentLaye
        }

        strcat(curLayStr, cStr2);
        if(currentLayer == 8)
         {
         itoa(7 , curLayChar);
         strcat(curLayStr,curLayChar);
         strcpy(curLayStrB," entities, for the user
         }
         else
         {
         itoa(currentLayer, curLayChar);
         strcat(curLayStr,curLayChar);
         if (currentLayer < 7)
            {
            strcpy(curLayStrB,betwStr);
            itoa( (currentLayer + 1),curLayChar);
            strcat(curLayStrB,curLayChar);
            strcat(curLayStrB,entiStr);
            }
            else
            {
            strcpy(curLayStrB, " for Application lay
            }
         }
    [[[[winText2 string: curLayStr]
                color: WHITE]
                center]
                centerV]
                show];
    [[[[[winText1 string: curLayStrB]
                color: WHITE]
                center]
                centerV]
                redraw]
                show];

    [idCurLayer inConnectWith: NULL];
    [[idCurLayer stopRun]
                 changeColor];

    if (currentLayer == 3)
       {
       if (ellLink != NULL)
          {
          [self freeEllLink];
          }
          else
          {
          [entryNode freeEllLink];
          }
       }

    if (currentLayer == 1)
       {
       if (phyLine != NULL)
          {
          [self freePhyLine];
          }
          else
          {
          [destination freePhyLine];
          }
       }
```

```
                        [idCurLayer deletePipe];

                        if (((currentLayer != 7) && (stackType == 8)) ||
                                ((stackType == 3) && (currentLayer != 3
                                {
                                 if ( currentLayer == 8 )
                                    { curInterface = 6;
                                    }
                                 else
                                    { curInterface = currentLayer - 1;
                                    }
                                [[interfaceColl at:curInterface] delete
                                }

                        [destination replyDiscStepLowOk];
                        if ( currentLayer == 3 )
                        {
                              if ( !skip )
                              {
                                  [[[destiStack askEntryNode] setSkip: FA
                              }
                              else
                              {
                                  skip = FALSE;
                              }

                         }
                        if ((currentLayer <= 2) && (workLayer >= 3))
                           {
                           skip = TRUE;
                           }

            }
        [self setCurLayer: (currentLayer - 1)];
        [destination setCurLayer: ([destination askCurLayer] - 1)];
        }
        else
        {
            [[[destiStack askEntryNode] setSkip: FALSE] discStepLowOk]
            [destiStack setSkip: FALSE];
             if (currentLayer == 0)
                {
                cont = 0;
                }

            skip = FALSE;
        }

    return cont;
    }

- (int)replyDiscStepLowOk
        /**************************************************************
        * Makes an instance of the Layer class change its color      *
        * following the situation.                                   *
        **************************************************************/

    {
    id idCurLayer;
    int curInterface, cont = 1;

    idCurLayer = [layerColl at: (currentLayer - 1)];

    if ([idCurLayer connectionState] == TRUE)
    {

        [idCurLayer inConnectWith: NULL];
        [[idCurLayer stopRun]
                     changeColor];

        [idCurLayer deletePipe];

        if (((currentLayer != 7) && (stackType == 8)) ||
                           ((stackType == 3) && (currentLayer != 3
        {
             if ( currentLayer == 8 )
             { curInterface = 6;
                  }
             else
             { curInterface = currentLayer - 1;
             }
             [[interfaceColl at:curInterface] deletePipe];
        }

    }

    return cont;
    }
```

```
- replyDisconnect
        /***********************************************************
        * Makes an instance of the Layer class flicker or change its  *
        * color following the situation.                              *
        ***********************************************************/

            {
            id idCurLayer;
            int curInterface;

            idCurLayer = [layerColl at: (currentLayer - 1)];

            if ([idCurLayer flick] == FALSE)
               {
               [idCurLayer flicker];
               }
               else
               {
               [[idCurLayer stopRun]
                            changeColor];

               [idCurLayer deletePipe];

               if (((currentLayer != 7) && (stackType == 8)) ||
                                        ((stackType == 3) && (currentLayer != 3
                                        {
                                         if ( currentLayer == 8)
                                             { curInterface = 6;
                                             }
                                         else
                                            { curInterface = currentLayer - 1;
                                            }
                                        [[interfaceColl at:curInterface] delete
                                        }

               }
            return idCurLayer;
            }

- askEntryNode
        /***********************************************************
        * Returns the address of the entry node, contained in the     *
        * entryNode instance variable.                                 *
        ***********************************************************/

              {
              return entryNode;
              }

- sEntryNode: aStack
        /***********************************************************
        * Stores the address passed as argument into the entryNode    *
        * instance variable.                                           *
        ***********************************************************/

              {
              entryNode = aStack;

              return self;
              }

- setCurLayer: (int)aValue
        /***********************************************************
        * Stores the number passed as argument, as the initiating layer*
        * number into the currentLayer instance variable              *
        ***********************************************************/

              {
              currentLayer = aValue;
              return self;
              }

- setWorkLayer: (int)aValue
        /***********************************************************
        * Stores the number passed as argument as the current layer *
        * number into the workLayer instance variable               *
        ***********************************************************/

              {
              workLayer = aValue;
              return self;
              }

- (int)askCurlayer
        /***********************************************************
        * Returns an integer which is the number of the initiating layer*
        * which is contained into the currentLayer instance variable  *
        ***********************************************************/

              {
              return currentLayer;
              }
```

```
- askLayerColl
        /***************************************************************
        * Returns the address of the collection containing the addresses*
        * of each instance of the Layer class composing the stack.  It *
        * is contained in the layerColl instance variable.             *
        ***************************************************************/
                {
                return layerColl;
                }


- setSkip: (BOOL)aValue
        /***************************************************************
        * Sets the skip instance variable to the boolean value passed as*
        * argument.                                                    *
        ***************************************************************/

                {
                skip = aValue;

                return self;
                }

- back: aView
        /***************************************************************
        * Sets the back instance variable to the address passed as     *
        * argument.                                                    *
        ***************************************************************/

                {
                back = aView;

                return self;
                }

- setDestiStack: aView
        /***************************************************************
        * Sets the the destiStack instance variable to the address     *
        * passed as argument.                                          *
        ***************************************************************/

                {
                destiStack = aView;

                return self;
                }

- setTextWin: aView: bView
        /***************************************************************
        * Stores the addresses passed as arguments into the winText1 and*
        * winText2 instance variable.  These arguments represent the   *
        * addresses of the views in which strings can be displayed     *
        ***************************************************************/

                {
                winText1 = aView;
                winText2 = bView;

                return self;
                }

- setAction: (int)aValueA
        /***************************************************************
        * Sets the actionType instance variable to the integer passed as *
        * argument.                                                    *
        ***************************************************************/

                {
                actionType = aValue;

                return self;
                }

- (BOOL)working
        /***************************************************************
        * Returns the value of the work instance variable.             *
        ***************************************************************/

                {
                return work;
                }

- setWorking: (BOOL)aBool
        /***************************************************************
        * Sets the work instance variable to the boolean passed as     *
        * argument.                                                    *
        ***************************************************************/

                {
                work = aBool;

                return self;
                }
```

```
- (BOOL)testConnect
        /********************************************************
        * Tests if a connection is already established at the     *
        * current layer. It returns a boolean which is TRUE if the*
        * connection is already established and FALSE otherwise.  *
        ********************************************************/

            {
            BOOL ok;

            if ([[layerColl at: (currentLayer - 1)] connectionState] == TRUE
                {
                ok = TRUE;
                }
                else ok = FALSE;
            return ok;
            }

- (BOOL)testConUp
        /**********************************************************
        * Tests if a connection is established anyware in the stack. *
        * It returns TRUE if this is the case, FALSE otherwise.      *
        **********************************************************/

            {
            BOOL ok = FALSE;
            int i;

            for (i=currentLayer ; i <= stackType -1 ; i++)
            {
            if ([[layerColl at: i] connectionState] == TRUE)
                {
                ok = TRUE;
                }
            }
            if ( stackType == 3 && ( [[layerColl at:2] connectionState] ) )
            {
              ok = TRUE;
            }

            return ok;
            }

- (int)searchWidthLink: aView
        /**********************************************************
        * Calculates the width of the red rectangle appearing between*
        * two stacks at a connection opening or closing.             *
        **********************************************************/

            {
            int xS, xD, length;

            xS = [self viewLeft];
            xD = [aView viewLeft];

            if (xS < xD)
                {
                length = (xD - (xS + aWidth));
                }
                else
                {
                length = (xS - (xD + aWidth));
                }
            return length;
            }

- (int)searchXOriginLink: aView
        /**********************************************************
        * Calculates the X coordinate where the red rectangle has to*
        * be located.                                               *
        **********************************************************/

            {
            int xS, xD, origin;

            xS = [self viewLeft];
            xD = [aView viewLeft];

            if (xS < xD)
                {
                origin = (xS + aWidth);
                }
                else
                {
                origin = (xD + aWidth);
                }
            return origin;
            }

- (int)searchYOriginLink
        /**********************************************************
```

```
        * Calculates the Y coordinate where the red rectangle has to*
        * be located.                                                *
        *************************************************************/

            {
            int origin;

            if (currentLayer <= 7)
               {
               origin = ([self viewLow]) + ((currentLayer - 1) * (aLayerHeig
               }
               else
               {
               origin = ([self viewLow]) + (((currentLayer - 2) * (aInterfac
               }
            return origin;
            }
- (BOOL)askSkip
        /*************************************************************
        * Returns the boolean stored into the skip instance variable*
        *************************************************************/

            {
            return skip;
            }

- resetFrame
        /*************************************************************
        * Resets the frame of the initiating layer to white          *
        *************************************************************/

            {
            [[layerColl at: (workLayer - 1)] frameColor: WHITE];
            return self;
            }

- (BOOL)connectionDone
        /*************************************************************
        * Tests if a connection is opened using every layer in the   *
        * stack. TRUE is returned if this is the case, FALSE         *
        * otherwise.                                                 *
        *************************************************************/

            {
            int i;
            BOOL conDone = FALSE;

            i = 0;
            while ((i < ([layerColl size])) && (conDone == FALSE))
                {
                if ((([layerColl at: i] connectionState]) == TRUE)
                   {
                   conDone = TRUE;
                   }
                i += 1;
                }
            return conDone;
            }

- (BOOL)deConnectionDone
        /*************************************************************
        * Tests if each layer inside a stack is disconnected.        *
        *************************************************************/

            {
            int i;
            BOOL deConDone = FALSE;

            i = 0;
            while ((i < ([layerColl size])) && (deConDone == FALSE))
                {
                if ((([layerColl at: i] connectionState]) == FALSE)
                   {
                   deConDone = TRUE;
                   }
                i += 1;
                }

            return deConDone;
            }
```

```
- createLink: (int)x: (int)y length: (int)lX: (int)lY
               /*********************************************************
               * Creates a white bar between an entry node stack and the*
               * X-25 ellipse.  Its place is identified by the two first*
               * arguments and its width and height by the two last ones*
               *********************************************************/

               {
               id line;

               line = [[RMGView origin: x: y extent: lX: lY superview: [self su
                              show];
               return line;
               }

- freeEllLink
          /*********************************************************
          * Hides and frees the link between an entry node stack and *
          * the X-25 ellipse.                                        *
          *********************************************************/

               {
               [[ellLink hide]
                        freeAll];
               return self;
               }

- freePhyLine
          /*********************************************************
          * Hides and frees the link between the machine stack and   *
          * the entry node stack.                                    *
          *********************************************************/
               {
               [[phyLine hide] free];
               return self;
               }

- (int)askStackType
         /*********************************************************
         * Returns an integer which is the value stored into the     *
         * stackType instance variable and which identifies the type *
         * of the stack.                                             *
         *********************************************************/

               {
               return stackType;
               }
```

```
- askIdNameColl
        /************************************************************
        * Returns the address of the collection containing the names *
        * of the different layers, contained into the nameColl       *
        * instance variable.                                         *
        *************************************************************/

                {
                return nameColl;
                }

- askDestiStack
        /*************************************************************
        * Returns the address of the destination stack of the connection*
        * opening or closing.  It is contained in the destiStack       *
        * instance variable.                                           *
        ***************************************************************/

                {
                return destiStack;
                }

- flickering
        /*************************************************************
        * Makes the current layer flicker and sets various parameters  *
        * concerning thsi instance of the Layer class.                 *
        ***************************************************************/

                {
                id idCurLayer;

                idCurLayer = [layerColl at: (currentLayer - 1)];
                if ([idCurLayer connectionState] == FALSE)
                            {
                            [idCurLayer flicker];

                            [idCurLayer inConnectWith: [entryNode replyConnec

                            [self setCurLayer: (currentLayer - 1)];
                            [entryNode setCurLayer: ([entryNode askCurLayer]
                            }
                return self;
                }
=:
```

```
#include "objc.h"
#include "rmg.h"
#include "envir.h"


@requires RMGView, Pipe;

/**********************************************************************
*  Class created for the OSI on X-25 application                     *
*  By: V ronique Nachtergaele and Dominique de Paul                  *
*  This class implements methods to create and manage the different  *
*       layers composing a stack                                     *
**********************************************************************/

= Layer: OsiStack(osi,working,RMGVW,Primitive,Collection)
{
        id conWith; /* contains the address of the other instance of the
                       Layer class with which this layer is connected */
        id aPipe;   /* contains the address of the white bar representing
                       an opened connection                           */
        int closeColor, openColor; /* contains the pastel and brilliant
                                      colors identifiers of a Layer
                                      instance                        */
        BOOL flickering; /*  contains TRUE if the Layer instance flickers
                             FALSE otherwise                          */
}

+ new: aView at: (int)x: (int)y color1: (int)aColor1
       color2: (int)aColor2 height: (int)aHeight width: (int)aWidth
       /***************************************************************
       * Creates the instance of Layer in the view which address is   *
       * passed as first argument.  The second and third arguments    *
       * identify the location on the screen.  Fourth and fifth       *
       * arguments identify the pastel and brilliant color of the layer *
       * The two last arguments are the height and width of the layer *
       ***************************************************************/

        {
        self = [self relative: x: y
                     extent: aWidth: aHeight
                     superview: aView
                     bkgd: aColor1];
        [[self frameWidth: 1]
              frameColor: WHITE];

        closeColor = aColor1;
        openColor  = aColor2;
        flickering = FALSE;

        [self showAll];

        return self;
        }

- flicker
       /***************************************************************
       * Installs the Layer instance in the active collection in      *
       * to make it flicker                                           *
       ***************************************************************/

        {
        if (flickering == FALSE)
           {
           [self installActive: self];
           flickering = TRUE;
           }

        return self;
        }

- (BOOL) flick
       /***************************************************************
       * Returns the value of the flickering instance variable        *
       ***************************************************************/

        {
        return flickering;
        }
```

```
- update
        /***********************************************************
        * This method is executed when the layer is in the active   *
        * collection.It enables the layer to flicker.               *
        ***********************************************************/

         {
          static int i = 10;
if (([self frameColor]) == RED)
    {
    [self frameColor: WHITE];
    }
          if (i == 10)
          {i = 0;
          if([self hidden])
              {
                     [self show];
              }
          else
              {
                     [self hide];
              }
          }
          else i+=1;

         return self;
         }

- stopRun
        /*************************************************************
        * Removes the Layer instance from the active collection. Thus  *
        * stopping the flickering motion.                              *
        *************************************************************/

         {
         [self deleteActive: self];
         flickering = FALSE;

         return self;
         }

- changeColor
        /*************************************************************
        * Changes the color of the Layer instance from pastel to      *
        * brilliant or vice versa depending on the case.              *
        *************************************************************/

         {
         if([self connectionState])
                 {
                 [[self bkgd: closeColor] show];
                 }
                 else
                 {
                 [[self bkgd: openColor] show];
                 }

         return self;
         }

- (BOOL)connectionState
        /*************************************************************
        * This layer tests the state of a layer to see if it is used *
        * in an opened connection. It returns TRUE if this is the case*
        * FALSE otherwise.                                             *
        *************************************************************/

         {
         BOOL state = FALSE;

         if([self bkgd] == openColor)
                 {
                 state = TRUE;
                 }
                 else
                 {
                 state = FALSE;
                 }

         return state;
         }

- inConnectWith: aLayer
        /*************************************************************
        * Stores into the conWith instance variable the address of the *
```

```
        * Layer instance to which the current one is connected.  This  *
        * address is passed as argument.                               *
        ****************************************************************/

         {
         conWith = aLayer;

         return self;
         }

- inConnectWith
        /****************************************************************
        * Returns the address contained in the conWith instance variable*
        ****************************************************************/

         {
         return conWith;
         }

- createPipe
        /****************************************************************
        * Creates the white bar in the middle of the layer when this    *
        * layer is concerned by an opened connection                    *
        ****************************************************************/

         {
         int middle;

         middle = (([self in_width]) / 2) - 5;
         aPipe = [Pipe new: self
                       at: middle: 0
                       color: WHITE
                       height: [self viewHeight]
                       width: 10];
         return self;
         }

- deletePipe
        /****************************************************************
        * Deletes the white bar situated in the middle of a layer when *
        * when it is concerned by an opened connection.                 *
        ****************************************************************/

         {
         [aPipe free];
         [self redraw];
         return self;
         }

=:
```

```
#include "objc.h"
#include "rmg.h"
#include "envir.h"


@requires RMGView, Pipe;

/*******************************************************************
* Class created for the OSI on X-25 application                    *
* By: V ronique Nachtergaele and Dominique de Paul                 *
* It manages the creation of an interface between two layers of an OSI*
* stack.                                                           *
*******************************************************************/

= Interface: OsiStack(osi,working,RMGVW,Primitive,Collection)
{
  id aPipe; /* contains the address of the white bar (the pipe) which
               is created in the middle of an interface              */
}

+ new: aView at: (int)x: (int)y color: (int)aColor height: (int)aHeight width: (
        /**************************************************************
        * Enables the creation of an instance of Interface in the view *
        * which address is passed as first argument.  The location is  *
        * identified by the second and third arguments.  It will be    *
        * in the color identified by the fourth argument and the two   *
        * last arguments are the height and the width of the interface *
        **************************************************************/

         {
         self = [self relative: x: y
                         extent: aWidth: aHeight
                         superview: aView
                         bkgd: aColor];

         [self showAll];

         return self;
         }

- createPipe
        /**************************************************************
        * Creates the Pipe instance in the middle of the interface when *
        * it is concerned by a opened connection.                      *
        **************************************************************/

         {
         int middle;

         middle = ([self in_width] / 2) - 5;
         aPipe = [Pipe new: self
                      at: middle: 0
                      color: WHITE
                      height: [self viewHeight]
                      width: 10];
         return self;
         }

- deletePipe
        /**************************************************************
        * Deletes the Pipe instance when [illegible]
        * an interface.
        **************************************************************/

         {
         [aPipe free];
         [self redraw];
         return self;
         }

=:
```

```
#include "objc.h"
#include "rmg.h"
#include "envir.h"


@requires RMGView;

/************************************************************************
* Class created for the OSI on X-25 application                         *
* By: V ronique Nachtergaele and Dominique de Paul                      *
* It manages the creation of the white bar which is created in the     *
* middle of an interface or a layer when they are concerned by an      *
* opened connection                                                     *
************************************************************************/

= Pipe: Envir(osi,working,RMGVW,Primitive,Collection)
  {
  }

+ new: aView at: (int)x: (int)y color: (int)aColor height: (int)aHeight width: (
        /**************************************************************
        * This method enables the creation of an instance of Pipe      *
        * in the view which address is passed as first argument.       *
        * The location is identified by the second and third arguments *
        * Its color is identified by the fourth argument and the two   *
        * last ones represent its height and width.                    *
        **************************************************************/

         {
         int i;

         self = [self relative: x: y
                        extent: aWidth: 1
                        superview: aView
                        bkgd: aColor];
         [self showAll];

         for (i = 2; i <= aHeight; i++)
             {
             [self size: aWidth: i];
             [self show];
             }

         [self showAll];

         return self;
         }

- modifyHeight: (int)aValue
        /**************************************************************
        * Modifies the overall height of the Pipe instance and sets it *
        * to the height passed as argument.                            *
        **************************************************************/

         {
         [[self size: 10: aValue]
               redraw];
         return self;
         }

- modifyHeight2: (int)aValue
        /**************************************************************
        * Modifies the overall height of the Pipe instance and sets it *
        * to the integer passed as argument.  It changes also the place*
        * of the Pipe instance on the Y axis and sets it  to the value *
        * computed from the difference of the current height of the    *
        * pipe and the new one passed as argument.                     *
        **************************************************************/
       {
       int temp;

       temp = [self viewHeight] - aValue;
       [[[self moveBy: 0: temp]
               size: 10: aValue]
               redraw]];
       return self;
       }
=:
```

Appendix 3 : Example of mainClass.m file

In this appendix, we give an example of mainClass.m file.

```
#include "objc.h"
#include "rmg.h"
#include "envir.h"

#define RMG_CLASSES SysIcon,RMGLine,RMGVLine,RMGHLine,\
                RMGRect,RMGAction1,RMGActionS,\
                ScrlBar,XY_indic,JoyStick,RMGZaxis,\
                Fixtur11,Fixtur17,Fixtur21,RMGHist,\
                RMGXYPlot,RMGGrid,RMGCirBox,Dial,RMGBall,RMGBall2,\
                RMGPattern,RMGStrip,RMGAxis,\
                IconMngr,IconSModel,\
                StringModel,DiamondModel,CircleModel,EllipModel,RectModel,\
                MTreeActOne,MTreeActPad,MTreeActGrp,Menu1_1,Menu1_2,Menu1_3

#define PPI_CLASSES  ObjGraph,Set,Dictionary,Point, Dictionary,SortCltn,Assoc,\
                     AsciiFiler,Stack,String,Sequence,IdArray,IntArray,BytArray

#define ENVIR RMGModStrD,RMGModStrI,RMGModStrE,RMGModStrS,RMGSpacStr,\
              CFunction,Envir,Resource,FileApp,\
              DirBrowser,FileMTree,DirMTree

#define COLOR_MAP ClrMapEdit,ClrMTree

#define IMAGES     IconEdit,IconMTree

#define CLIP_BOARD IconClpBrd,ICBMTree

#define MOREIMAGES SunRaster,PatrnEdit

#define DOCVIEW    DocEdit,DocViewer, DocVMTree,DocEMTree

#define CLOCK      AnalogClock, AClkMTree ,DigiClock, DClkMTree

#define TERMINAL      FileSBwsr,FSMTree
#define BROWSER ClsBrowser,ClsBSMTree,\
                MsgBrowser,MsgBSMTree,\
                ClsTree2,ClsMTree,InsVXtract,InsBrowser,InsBSMTree, \
                Mess,MessMTree,PtrBrowser, PtrBSMTree

#define VALUES    Voltmeter, VoltMTree

#define DEPENDENTS ScrlStripE,ScrlSMTree,ScrlStrip,\
                   ListBox,ListMTree
#define JOHN RMGXBar, RMGNumLab, BarMeter,VertBarMet, BarXMTree

#define VIP       Calc,warning,pr4,NetPanel,MMEdit, RMGHpib,RMGGpio,\
                  PanelMTree, Thumbwheel,ThumbMTree
```

```
#define DIP_PROG ProgEdit,ProgMTree, ProgEdit2,\
                 Expr, Statement,Program,\
                 Int,SymTable, CharString

#define SCIENCE SubParticle, PartPhysics,Pendulum,PartPMTree, \
                MoleView,MoleMTree

#define WAVE_CAP RMGAttrib,RMGGpic,WaveCapture,WaveMTree

#define Simulation RMGMarker,\
                   DiffModel, SemiModel,Diffusion ,DiffMTree,Lattice

#define PROGMATRIX ProgEdit1,ProgMMTree,ProgEdit2,ProgEdit3

#define NETWORK SrcDict,DiaModel,PanelModel,ArcModel,NetToken,\
                PanelDiag,ConnectDiag,NetMTree,RMGShape1,\
                FuncModel,ReturnModel,StopModel,MessModel,\
                ICModel,ICModel1,\
                GenModel,UNetModel,NullModel,\
                ICModel,LifeModel,LifeDiag

#define PTY Pty,PtyApp,PAMTree

#define PHASOR          Phasor,PhasorMTree,Circuit,CircMTree
#define SMITH           Smith, SmithMTree

#define ICDESIGN        IcDesign,ICDMTree,IcNetwork,IcNetwMTree,IcNode,IcConnect
                        IcSticks,IcStMTree,IcLayout,IcLtMTree,IcMessArea,IcMesMT
                        IcBox,IcDesrules,IcNetworkArea,IcStickArea,IcLayoutArea,
                        IcMaskArea,IcMask,IcDesrArea,IcElmntArea,IcElmntMTree,\
                        IcElmnt,IcLine



/***************************************************************
*  Classes of COLOS Member FUNDP Namur BELGIUM                 *
***************************************************************/

#define DCO Ftam, Ftametat

/***************************************************************
*  Classes of COLOS Member FUNDP Namur 2 BELGIUM               *
***************************************************************/

#define OSI1 Pipe, Layer, Interface, OsiStack, Osi1, Osi1MTree

/***************************************************************
*  Classes of COLOS Member Ecole Centrale de Lyon              *
***************************************************************/

#define TROTEK  ShowCol, TrotekMTree, Trotek

#define ECLTOOLS EclModStrI, EclVMTree, EclView, EclIcon, \
                 EclStrip, EclButton, EclPlot

#define PONTMONO PontMono, PoMoMTree, PoMoCalc, PoMoDes


#define TD TDlines,TDlinesMTree,RMGActionPar,TDtestMTree,TDwheels,TDspring,TDfun

#define POT Potentiel,PotMTree

#define AMPLI GeneBF,GBFMTree,XGeneBF,XGBFMTree,MaskMTree,Masque,\
              ModStrD,Scope,OscMTree,Ampli,AmpMTree,XAmpli,InvAmpl, \
              Electron, EGMTree

/***************************************************************
*  Classes of COLOS Member IPN (Kiel).                         *
***************************************************************/

#define LEGS LEGS_Dummy, \
            Graphix, Buffer, ModStrD, ModStrI, ModStrS, LegsAct0, \
            PointDoub, RMGNewLine, LineView, RegionView, \
            BounceBall, BouncPartic, FieldPartic, \
            Hello, HelloMTree, Latte, LatteMTree

#define FIXMENU FM_Dummy, FMView, \
                ButtonView, ButtonEnvir, Controller, FixMAct1, \
                FixMAct2, FixMAct3, UpdateItem, UpdToggle,  \
                FixMenuItem, ToggleItem, FixMenu, FixMMTree, Toolkit, \
                ColosAppls

#define ELSTAT  ES_Dummy, \
                IconMTAct3, IconMTAct4, ObjMTAct5, IconMTAct6, \
                ObjMTAct61, IconMTAct62, IconMTAct64, IconMTAct7, \
                ObjMTAct71, BallMTAct72, MTreeActItm, MTActItmOne, \
                ObjMTActItm, ModelAct8, BallMTAct9, Interact, InteractRel,\
                Field, BallWorld, ParticWorld, FieldWorld, ElStatWorld, \
                BallMngr, ParticMngr, FldPartMngr, ElStPartMgr, CouIconFM,\
                GraIconFM, CoulombFM, GravityFM, worldFM, FieldFM, LawApp,\
                FileMngr, ElStat, ElStatMT, BallMT, BallGroupMT, AllBallsMT,\
                FileMT, LineMT, InfLawMTAct, HelpMTAct, HelpMT, FieldMT, \
                InfoViewMT, ProgEditMT, RegionMT
```

```
#define CELLSYSTEM CS_Dummy, \
                   FMMTAct0, FMMTAct1, FMMTAct2, Cell, Connection, \
                   ConRecord, CellAgg, CellView, CellSystem, SystemRep, \
                   LinearRep,SystemView, SysEditView, SysViewMT, \
                   ACellSystem, ACellSysFM, CMapCltn, CellViewMT, \
                   LinPnts, CSFileMngr, InfoMngr, LinRects, LinViews, \
                   LinLines, NetRep, NetLines,  NetPnts, NetRects, \
                   NetViews, CellNet, CNetFMMT, CellNetFM, ActCellStrp, \
                   ActCellMT, MTActGrpOne, CSInfLanAct, CSHelpMTAct, \
                   CSHelpMT, DiffRateMT, DiffNodeMT


/***************************************************************
*  Classes of COLOS Member MILANO                               *
***************************************************************/

#define SIMAUTOMI FAutArco,NModel,FAutNodo,Automa,SimulAutoma,FAutMTree

#define NDAUTOMI NdAmArco,NdNmodel,NdAmNodo,NdAutoma,SimulAutNd,NdAmMTree

#define TURING  TgArco, Label, TgNodo, Turing, TuringWorks, TuringMTree

#define AUTOMI Automata, AutMTree

#define NLNET NlNet, NeuralNet, NlNodo, NlArco, NlNetMTree

#define NEURBIN NeurBin, NeurBMTree

#define NEURPROB NeurProb, NeurPMTree

#define NEURCONT NeurCont, NeurCMTree

#define BIO Nodo,Arco,NumReal,Bio,BioMTree

#define MODELS NeurModels, ModelMTree



#define TRAINING coucheN



#define Crystal  Error

#if LINK
#define INK       Aout,InkAppl,InkMain,InkObjc,InkSym,OrdColl,OrdCollt
#endif

@classes(
    RMG_CLASSES,

#ifdef STAGE1
        ENVIR,
#endif

#ifdef STAGE2
        COLOR_MAP,
#endif

#ifdef STAGE3
      CLOCK,
#endif

#ifdef STAGE5
        DOCVIEW,
#endif
#ifdef STAGE5A
      BROWSER,
#endif

#ifdef STAGE6
      IMAGES,
      CLIP_BOARD,
#endif
#ifdef STAGE6A
      MOREIMAGES,
#endif

#ifdef STAGE7
        SCIENCE,
#endif

#ifdef STAGE[illegible]
        VIF,
#endif

#ifdef STAGE[illegible]
        VALLEE,
        DEFENDENTE,JOHN,
```

```
#endif
#ifdef STAGE11
        USR_PROG,
#endif
#ifdef STAGE11A
        PROGMATRIX,
#endif
#ifdef STAGE12
        Simulation,
#endif

#ifdef STAGE14
        NETWORK,
#endif

#ifdef STAGE15
        PTY,
#endif

#ifdef STAGE18
        TRAINING,
#endif

#ifdef STAGEED
        PHASOR, SMITH,
#endif

#ifdef STAGEDE
        ICDESIGN,
#endif

#ifdef STAGELY
        TROTEK, ECLTOOLS, PONTMONO,
#endif

#ifdef STAGEMU
        TD,
#endif

#ifdef STAGEPA
        AMPLI, POT,
#endif

#ifdef STAGEKI
        LEGS, FIXMENU, ELSTAT, CELLSYSTEM,
#endif

#ifdef STAGEMI
       SIMAUTOMI, NDAUTOMI, TURING, AUTOMI,
       NLNET, NEURBIN, NEURPROB, NEURCONT, BIO, MODELS,
#endif

#ifdef STAGENA
       DCD,
#endif

#ifdef STAGENA2
       OSI1,
#endif

        PPI_CLASSES

#if CRYSTAL
        ,Crystal
#endif
        ,  INK
       )

@messages()
```

Appendix 4 : Example of individual makefile

In this appendix, we give an example of individual makefile.

```
.PRECIOUS: osi.a


COMPILE=objcc3.3
OPT =    -q -O -N -nRetain +ffpa -I../../.. \
                                 -I../../../../CP \
                                 -I../../../../INCLUDE \
                                 -I../../../METERS \
                                 -I../../../SCIENCE \
                                 -I../INCLUDE \
                                 -I../CP
OPTISL =  -I/usr/local/lib/ink

osi.a : osi.a(Osi1.o) osi.a(Osi1MTree.o) \
        osi.a(OsiStack.o) osi.a(Layer.o) osi.a(Interface.o) osi.a(Pipe.o)
        objcc3.3 $(OPT) -c  $(?:.o=.m)
        cp [CP]_* ../../../CP
        ar ruv osi.a $?;
        /bin/rm -f $?


osi.out:; cd ../../../; make osi.out
.SUFFIXES :
.SUFFIXES : .m  .a
.SUFFIXES : .m  .o
.SUFFIXES : .m  .r

.m.o: $(COMPILE) $(OPT)  $(OPTISL)  -c  $< ;cp C_$* ../../../CP ;cp P_* ../../..

.m.a: ;

.m.r: ; $(COMPILE) $(OPT)   -C $<
        /usr/local/lib/ink3.3/inkstamp <$*.c > $*I.c
        cc -c $*I.c
        mv $*I.o $*.r
        /bin/rm -f $*I.c
        /bin/rm -f $*.c

clean :; /bin/rm *.o C_* P_* CellSystemAppl

dotRFiles =  .r


DOTRFILES : Osi1.r Osi1MTree.r OsiStack.r Layer.r Interface.r Pipe.r ; cp [CP]_*

CellSystemAppl : $(dotRFiles)
        ld -dr  -N $(dotRFiles) -o CellSystemAppl
```

Appendix 5 : Example of the environment's makefile

In this appendix, we give an example of the environment's makefile.

```
COMPILE=objcc3.3

.PRECIOUS: envir.a osi.out\
        CLRMAP/clrmap.a CLOCK/clock.a DOCVIEW/docview.a\
        BROWSER/browser.a ICONS/icon.a ICONS/icon1.a SCIENCE/science.a\
        PANEL/panel.a METERS/meters.a USRPROG/prog.a SEMI/semi.a \
        NETS/network.a NETS/net1.a SHAPE/shape.a PTY/pty.a TRAINING/training.a \
        COLOS/NAMUR/OSI1/osi.a  COLOS/NAMUR/DCO/corbygy.a

OPT = -q  -N  -I../INCLUDE -I../CP  -I/usr/local/lib/objc3.3
OPT1 = -q -N  -I./CP/ -I../INCLUDE -I../CP/
OPTISL = -I/usr/local/lib/ink3.3
OPTALL1 = -q -N -I../CP/ -ICP -I../INCLUDE  $(OPTISL)

LIB =  -ll -lm  -lmalloc   -lBSD  -ldvio
LIBRMG =  ../SRC/RMG.a ../LIB/rmg.a
LIBUSR = USRPROG/PROGMODEL/ProgModel.a
LIBISL = /usr/local/lib/objc3.3/hpisl.a /usr/local/lib/ink3.3/ink.a

all:    osi.out

#  ************** STAGE1 **************** uses P_working

envir.a : envir.a(Ghost.o)        envir.a(EntryPad.o) \
        envir.a(RMGModStr.o)    envir.a(RMGModStrD.o) \
        envir.a(RMGModStrI.o)   envir.a(RMGModStrE.o) \
        envir.a(RMGModStrS.o) \
        envir.a(MTreeAct.o)     envir.a(MTreeActBar.o) \
        envir.a(MTreeActPad.o) envir.a(MTreeActGrp.o) \
        envir.a(MTreeActOne.o) \
        envir.a(Envir.o)        envir.a(EnvMTree.o)\
        envir.a(NullMTree.o)\
        envir.a(DirBrowser.o)   envir.a(DirMTree.o) \
        envir.a(FileApp.o)      envir.a(FileMTree.o)
        objcc3.3 $(OPT)   -c  $(?:.o=.m)
        ar ruv envir.a $?
        /bin/rm -f $?

# ************ stage 2 **************  uses P_colormap
CLRMAP/clrmap.a : CLRMAP/*.m ;( cd CLRMAP; make clrmap.a )

# ************ stage 3 **************  uses P_clock
CLOCK/clock.a: CLOCK/*.m ; (cd CLOCK; make clock.a )

# ************ stage 5 ************** uses P_docviewer
DOCVIEW/docview.a: DOCVIEW/*.m ; ( cd DOCVIEW; make docview.a)

# ************ stage 5A ************** uses P_browser
BROWSER/browser.a: BROWSER/*.m ; (cd BROWSER; make browser.a)

# ************ stage 6 ************** uses P_icons
ICONS/icon.a: ICONS/*.m; (cd ICONS; make icon.a )
```

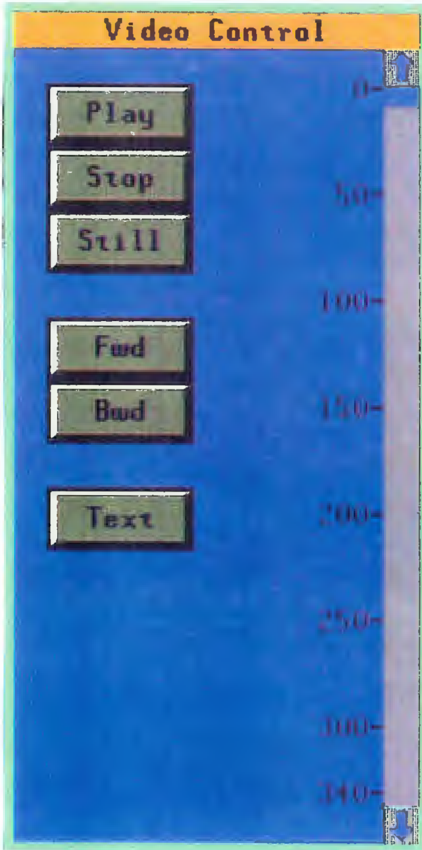
```
# ************ stage 6A ************** uses P_icons
ICONS/icon1.a: ICONS/*.m; (cd ICONS; make icon.a icon1.a )

# ************ stage 7 ************** uses P_science
SCIENCE/science.a:SCIENCE/*.m ; (cd SCIENCE; make science.a )

# ************ stage 8 ************** uses P_diagram
PANEL/panel.a: PANEL/*.m ; (cd PANEL; make panel.a  )

# ************ stage 10 **************  uses P_indicator and p-NumLab
METERS/meters.a:METERS/*.m; (cd METERS; make meters.a )

# ************ stage 11 **************  uses P_userProg
USRPROG/prog.a: USRPROG/*.m ;(cd USRPROG; make prog.a )

# ************ stage 12 **************  uses P_diff  and P_simul
SEMI/semi.a : SEMI/*.m;(cd SEMI; make semi.a  )

# ************ stage 14 **************  uses P_network
NETS/network.a:NETS/*.m; (cd NETS; make network.a )
NETS/net1.a:NETS/*.m; (cd NETS; make network.a )

SHAPE/shape.a:SHAPE/*.m; (cd SHAPE; make shape.a )

# *************** stage 15 ***************************
PTY/pty.a:PTY/*.m; (cd PTY; make pty.a )

# ****************stage 18 **********  uses P_training
TRAINING/training.a: TRAINING/*.m; (cd TRAINING; make training.a)

# **************** stage ED (Heriot-Watt, Edinburgh) *******
#COLOS/EDINBURGH/PHASOR/phasor.a: COLOS/EDINBURGH/PHASOR/*.m; (cd COLOS/EDINBURG
#COLOS/EDINBURGH/SMITH/smith.a: COLOS/EDINBURGH/SMITH/*.m; (cd COLOS/EDINBURGH/S

# **************** stage DE (Delft) *******
#COLOS/DELFT/ICDESIGN/icdesign.a: COLOS/DELFT/ICDESIGN/*.m; (cd COLOS/DELFT/ICDE

# **************** stage LY (Lyon) *******
#COLOS/LYON/TROTEK/trotek.a: COLOS/LYON/TROTEK/*.m; (cd COLOS/LYON/TROTEK; make
#COLOS/LYON/ECLTOOLS/ecltools.a: COLOS/LYON/ECLTOOLS/*.m; (cd COLOS/LYON/ECLTOOL
#COLOS/LYON/PONTMONO/pontmono.a: COLOS/LYON/PONTMONO/*.m; (cd COLOS/LYON/PONTMON

# **************** stage MU (Murcia) *******
#COLOS/MURCIA/TD/td.a: COLOS/MURCIA/TD/*.m; (cd COLOS/MURCIA/TD; make td.a)

# **************** stage PA (Paris) *******
#COLOS/PARIS/AMPLI/ampli.a: COLOS/PARIS/AMPLI/*.m; (cd COLOS/PARIS/AMPLI; make a
#COLOS/PARIS/POT/pot.a: COLOS/PARIS/POT/*.m; (cd COLOS/PARIS/POT; make pot.a)

# **************** stage IP (KIEL) *******
#COLOS/KIEL/LEGS/legs.a: COLOS/KIEL/LEGS/*.m; (cd COLOS/KIEL/LEGS; make legs.a)
#COLOS/KIEL/FIXMENU/fixmenu.a: COLOS/KIEL/FIXMENU/*.m; (cd COLOS/KIEL/FIXMENU; m
#COLOS/KIEL/ELSTAT/elstat.a: COLOS/KIEL/ELSTAT/*.m; (cd COLOS/KIEL/ELSTAT; make
#COLOS/KIEL/CELLSYSTEM/cellsystem.a: COLOS/KIEL/CELLSYSTEM/*.m; (cd COLOS/KIEL/C

# ************* stage MI (Milan) **************************
#COLOS/MILAN/SIMAUTOMI/simautomi.a:COLOS/MILAN/SIMAUTOMI/*.m ; (cd COLOS/MILAN/S
#COLOS/MILAN/NDAUTOMI/ndautomi.a:COLOS/MILAN/NDAUTOMI/*.m ; (cd COLOS/MILAN/NDAU
#COLOS/MILAN/TURING/turing.a: COLOS/MILAN/TURING/*.m ; (cd COLOS/MILAN/TURING;
#COLOS/MILAN/AUTOMI/automata.a: COLOS/MILAN/AUTOMI/*.m; (cd COLOS/MILAN/AUTOMI;
#COLOS/MILAN/NLNET/neural.a: COLOS/MILAN/NLNET/*.m; (cd COLOS/MILAN/NLNET; make
#COLOS/MILAN/NEURBIN/neurbin.a: COLOS/MILAN/NEURBIN/*.m; (cd COLOS/MILAN/NEURBIN
#COLOS/MILAN/NEURPROB/neurprob.a: COLOS/MILAN/NEURPROB/*.m; (cd COLOS/MILAN/NEUR
#COLOS/MILAN/NEURCONT/neurcont.a: COLOS/MILAN/NEURCONT/*.m; (cd COLOS/MILAN/NEUR
#COLOS/MILAN/BIO/bio.a: COLOS/MILAN/BIO/*.m; (cd COLOS/MILAN/BIO; make bio.a)
#COLOS/MILAN/MODELS/model.a: COLOS/MILAN/MODELS/*.m; (cd COLOS/MILAN/MODELS; mak

# ************* stage NA2 (Namur) **************************
COLOS/NAMUR/OSI1/osi.a:COLOS/NAMUR/OSI1/*.m ; (cd COLOS/NAMUR/OSI1; make osi.a)
COLOS/NAMUR/DCO/corbugy.a:COLOS/NAMUR/DCO/*.m ; (cd COLOS/NAMUR/DCO; make corbug
# COLOS/NAMUR/couche.a:COLOS/NAMUR/*.m ; (cd COLOS/NAMUR; make couche.a)


OPTS = $(NORMAL)

STAGE1    = envir.a
STAGE2    = CLRMAP/clrmap.a
STAGE3    = CLOCK/clock.a
STAGE5    = DOCVIEW/docview.a
STAGE5A   = BROWSER/browser.a
STAGE6    = ICONS/icon.a
STAGE6A    = ICONS/icon1.a
STAGE7    = SCIENCE/science.a
STAGE8    = PANEL/panel.a
```

```
STAGE10   = METERS/meters.a
STAGE11   = USRPROG/prog.a  USRPROG/PROGMODEL/ProgModel.a
STAGE12   = SEMI/semi.a
STAGE14   = NETS/network.a SHAPE/shape.a
STAGE15   = PTY/pty.a
STAGE18   = TRAINING/training.a
STAGEED   = COLOS/EDINBURGH/PHASOR/phasor.a COLOS/EDINBURGH/SMITH/smith.a
STAGEDE   = COLOS/DELFT/ICDESIGN/icdesign.a
STAGELY   = COLOS/LYON/TROTEK/trotek.a COLOS/LYON/ECLTOOLS/ecltools.a COLOS/LYON/
STAGEMU   = COLOS/MURCIA/TD/td.a
STAGEPA   = COLOS/PARIS/AMPLI/ampli.a COLOS/PARIS/POT/pot.a
STAGEKI   = COLOS/KIEL/LEGS/legs.a COLOS/KIEL/FIXMENU/fixmenu.a COLOS/KIEL/ELSTAT
STAGEMI   = COLOS/MILAN/SIMAUTOMI/simautomi.a \
            COLOS/MILAN/NDAUTOMI/ndautomi.a \
            COLOS/MILAN/TURING/turing.a \
            COLOS/MILAN/AUTOMI/automata.a \
            COLOS/MILAN/NLNET/neural.a \
            COLOS/MILAN/NEURBIN/neurbin.a \
            COLOS/MILAN/NEURPROB/neurprob.a \
            COLOS/MILAN/NEURCONT/neurcont.a \
            COLOS/MILAN/BIO/bio.a \
            COLOS/MILAN/MODELS/model.a
STAGENA2  = COLOS/NAMUR/OSI1/osi.a
STAGENA   = COLOS/NAMUR/DCO/corbugy.a


ALL = $(STAGE1) \
      $(STAGE2)\
      $(STAGE3) \
      $(STAGE5) \
      $(STAGE5A) \
      $(STAGE6) \
      $(STAGE6A) \
      $(STAGE7)  \
      $(STAGE8) \
      $(STAGE10) \
      $(STAGE11) \
      $(STAGE12) \
      $(STAGE14) \
      $(STAGE15) \
      $(STAGE18) \
      $(STAGENA2) \
      $(STAGENA)


ALLSTAGE = -DSTAGE1 \
           -DSTAGE2 \
           -DSTAGE3  \
           -DSTAGE5 \
           -DSTAGE5A \
           -DSTAGE6 \
           -DSTAGE6A \
           -DSTAGE7 \
           -DSTAGE8 \
           -DSTAGE10 \
           -DSTAGE11 \
           -DSTAGE12 \
           -DSTAGE14\
           -DSTAGE15\
    -DSTAGE18\
           -DSTAGENA2 \
           -DSTAGENA


#ALLSTAGE is used to switch mainClass.m

ALLLIB =  $(ALL)  $(LIBRMG) $(LIBISL) $(LIB)

osi.out : $(ALL) CFunction.o Resource.o  main.o mainClass.o ;
        objcc3.3 $(OPT1) $(OPTISL) CFunction.o Resource.o   main.o mainClass.o $

env : ;
        objcc3.3 $(OPT1)  $(OPTISL) CFunction.o Resource.o main.o mainClass.o $(

main.o: main.m ; objcc3.3 $(OPTALL1)  -c  main.m
mainClass.o: mainClass.m CP/[CP]_*
         objcc3.3 $(OPTALL1) $(ALLSTAGE) -c -ccOpt:Nt60000 -asOpt:Ns10000  mainC

.SUFFIXES :
.SUFFIXES : .m  .a
.SUFFIXES : .m  .o
.SUFFIXES : .m  .r
.m.o: : $(COMPILE) $(OPT)  $(OPTISL)  -c  $<

.m.a: :
.m.r: : $(COMPILE) $(OPT)   -C $<
        /usr/local/lib/ink3.3/inkstamp <$*.c > $*I.c
        cc -c $*I.c
        mv $*I.o $*.r
```

```
        /bin/rm -f $*I.c
        /bin/rm -f $*.c

clean :; touch *.m */*.m; /bin/rm  [CP]_* */[CP]_*

cp :; touch mainClass.m

E : osi.out; osi.out
Z :; /bin/rm core
t :; /bin/rm osi.out; make osi.out

allClean:; touch *.m */*.m
```

Appendix 6 : Screen copy of the Video and OSI on X-25 applications

We give in this appendix a screen copy of the Video application and a screen copy of the OSI on X-25 application.


Video Control
Play
Stop
Still
Fwd
Bwd
Text
0
50
100
150
200
250
300
340

OSI on X-25
SASE ->
CASE ->
Presentation ->
Session ->
Transport ->
Network ->
Data ->
Physical ->
X 25

Appendix 7 : RMG directory structure

We give here the RMG directory structure (from [FAZARINC, 89]). This will enable the reader to find easily the directories we talk about in this work.

Hewlett Packard

RMG/

Software Organization June 1989

CP/
C_
P_

CP1/
C_ P_
Collection
Object
Primitive

DATA/
map.ras
trio.ras

ENVIR/
env.out
makefile
mainClass.m
Envir.m
FileApp.m
main.m
C_ P_ *.m
envir.a
RMGModStr*

INCLUDE/
rmg.h
envir.h
show.h
*.h

LIB/
rmg.a

SRC/
C_ P_
*.m
RMG.a

rebuild

readme
*.tar.Z

/lib
c22@ ccom2@
cpp lib*.a

/usr/local
bin
objcc3.3
tint2
lib
crystal
objc3.3 ink3.3

ICONS/
*.icon
Frac*.icon
SYSICON/

MOLE/
*.icon
*.dat

NETWORK/
*.nets
*.nets,a

ODATA/
*.map

PANEL/
*.icon
*.panel
*.m

PATTERN/
*.pat

SEMI/
*.icon

USRPRGM/
*.prgm
loop*

BROWSER/
browser.a
CPm

CLOCK/
clock.a
CPm

CLRMAP/
clrmap.a
CPm

CONFIG/
A.menu C.menu
B.spec E.config
cpDirs srcDirs
*.help DEFAULTS/

DOCVIEW/
docview.a
CPm

ICONS/
icon.a
icon1.a
CPm

METERS/
meters.a
CPm
*.h

NETS/
*.h
network.a
CPm

PANEL/
panel.a
CPm
prf.m

PTY/
*.h
pty.a
CPm

SCIENCE/
science.a
CPm

SHAPE/
shape.h
shape.a
CPm

TRAINING/
training.a
Program Templates

USRPROG/
prog.a
CPm

PROGMODEL/
*.m.o.h
support

CP/
C_ P_

Appendix 8 : Example of A.menu file

We give here an example of the A.menu file.

```
51 12 10
AnalogClock
Automata
BarMeter
Calc
CellNet
Circuit
ClrMapEdit
ClsBrowser
ClsTree2
coucheN
Diffusion
DigiClock
DocEdit
DocViewer
Electron
ElStat
Ftam
IconClpBrd
IconEdit
InsBrowser
Latte
LifeDiag
ListBox
Mess
MoleView
MsgBrowser
NetPanel
NeuralNet
NeurModels
Osil
PanelDiag
PartPhysics
PatrnEdit
Pendulum
Phasor
Potentiel
ProgEdit2
PtrBrowser
PtyApp
ScrlStripE
SunRaster
TDfunction
TDiso
TDlines
TDspring
TDwave
TDwheels
Toolkit
Trotek
VertBarMet
Voltmeter
/* 34 items;
   maximum 12 characters;
   display 10 items on screen */
/* note the names above must correspond to class names */
```