A Beginners' Guide to Visual Prolog

Version 7.2

Thomas W. de Boer

Version of this book: 1.1

Preface

There is a beautiful programming language out there. It is called Prolog.

Originally the language Prolog was created by Calmeraur it was popularized by famous books by Clocksin & Mellish and by Ivan Bratko and brought to the masses by software producer Borland who sold Turbo Prolog for MsDos systems. But the masses didn't buy it and Borland stopped selling Turbo Prolog.

That was long ago. After a few years it became apparent that the makers of Turbo Prolog had returned to Denmark and started the company PDC: the Prolog Development Company. They put years of effort into their dialect of the language and now there is Visual Prolog. It is object-oriented, it has a Graphical User Interface, it has a Integrated Development Environment.

And it still has the charm and elegance of the language they call Prolog.

Formerly the great problem in learning Prolog was that you had to master the very different thinking of a declarative programming language. Once you've mastered it, it is a great way to write computer programs, so the effort was and is worthwhile.

But nowadays it is not only mastering a declarative language, there are these other aspects of a modern programming language: object-orientation, graphical user interface, et cetera. This book is an effort to tell you all about it.

This book is an introduction. It is meant for people who know little about programming. It is not for people that know nothing. You should know the basics about computers and that it is possible to program them and that for a program you use a programming language. But nowadays these things seem to bee widely known - even among computer nitwits. What when you know more? Then you should look for other sources.

- When you are programmer and know about other languages like Visual Basic or C##, read the book Prolog for Tyros by Eduardo Costa.
- When you know about other languages and are curious about Prolog, read the articles by Thomas Linder Puls and Sabu Francis at the Visual Prolog web site at www.pdc.dk.
- When you have experience in Prolog programming, look for details about Visual Prolog, object orientation in the advanced tutorials at the same web site.

You will find the publications at the website: www.pdc.dk -> Solutions -> Visual Prolog -> Tutorials. At that site you will also find a Wiki on programming in Visual Prolog

There is a lot available about Visual Prolog. But for a beginner it is very hard to find his way through all the material. This book tries to bring structure in the apparent chaos for the beginner. I edited this book by borrowing and rewriting texts from other people (with their consent) and by filling in some gaps with old material that I wrote years ago. It all comes together in this introductory text. I have tried to keep the content as low profile as possible, so even beginners can use it.

But if you think that some things are not understandable, please let me know. You can reach me at t.w.de.boer@gmail.com.

Groningen, Summer 2009. Thomas W. De Boer

Contents

Introduction

This is a book on how to write computer programs in Visual Prolog. Visual Prolog, or VIP for short, is an Object-Oriented Programming Language (OOPL for short) that can be used to create programs that run under MsWindows. This means that the programs that you write should support a GUI, a Graphical User Interface.

Writing this book gives immediately a funny didactic problem: you will need to learn about these three subjects (programming in VIP, programming the GUI and object-oriented programming) at the same time. But it is good practice to write about one thing at a time. I shall try to solve this problem by writing incrementally. I introduce one subject a little, just enough to understand the basics. Then I switch to another subject, again to give you just enough understanding for the chapters that follow.

This sounds rather vague and it is. Let me elaborate on what I am going to do.

1. I start with explaining what you see when you start Visual Prolog. What you see is the Integrated Development Environment, the IDE. This is the place where you will be programming in Visual Prolog. In the first few chapters you will get to grips with the basics of the IDE. You will do a lot of things that will make you feel like Harry Potter: you enter some magic formulae and suddenly it appears that you wrote a working program.
2. Then I should like to extend your knowledge to the basics of the Graphical User Interface (GUI) as that is important in programming under MsWindows. But when you program the GUI, it is unavoidable that you will need some knowledge of Prolog. So first I pay attention to explaining the basics of Programming in Prolog.
3. Then we go back to Visual Prolog as an Object Oriented Programming language. In the next chapters I tell you the basics of OO programming and how it is implemented in Visual Prolog. In this chapter you will find stuff about class declaration, class interface and class implementation.
4. Different from other Prolog dialects, Visual Prolog is a strongly typed language. In chapter 9 we take a look at the declarative side of VIP.
5. The last few chapters are on special topics that are important in Prolog like recursion, and lists. Also we take a look at well-known data structures as stacks, queues and trees. Finally there is a chapter on how to decently handle input and output.

By then you are ready to explore Prolog and VIP by yourself. There are a few very good books available, there are several articles on various subjects on the PDC site. Besides, as this book is for the very beginners, by then you are no longer a “beginner”. So it is time to close the book.

I wish you a lot of fun in reading and trying this book. Please let me know when you encounter trouble, I like to improve this book as much as I can. You can reach me at t.w.de.boer@rug.nl. Please indicate in the subject line of your message “Visual Prolog”.

Acknowledgments
I have to thank Eduardo Costa, Sabu Francis and Thomas Linder Puls for their generosity to allow me to freely use texts they have written. I extensively used (and maybe abused) their writings. I tried to be precise and I indicate in every chapter where the text is originally from. Also I want to thank Yi Ding. He did a great job in translating version 1.0 of this book into Chinese and indicated a lot of mistakes and typing errors. Thanks also to Dieter Lutolf for sending a lot of corrections. When I missed

something, please let me know as I want to give everybody the credits he or she deserves. I hope it will be clear that any mistakes are purely mine.

Chapter 1 The Integrated Development Environment[1]

In this chapter I introduce the Integrated Development Environment of Visual Prolog. Programming nowadays is not only done by typing code in an editor. Programming has become very complex - that's why the developers of programming languages offer more than a programming language. They provide you with programming aids. The Integrated Development Environment (IDE for short) is such an aid. When you create a Visual Prolog computer program, you will use the IDE. The purpose of this chapter is to introduce to you feel the Integrated Development Environment (IDE) of Visual Prolog (VIP).

1.1 The Integrated Development Environment IDE

When you start Visual Prolog, what is really started is the Integrated Development Environment, the IDE. It looks more or less like figure 1.1[2]

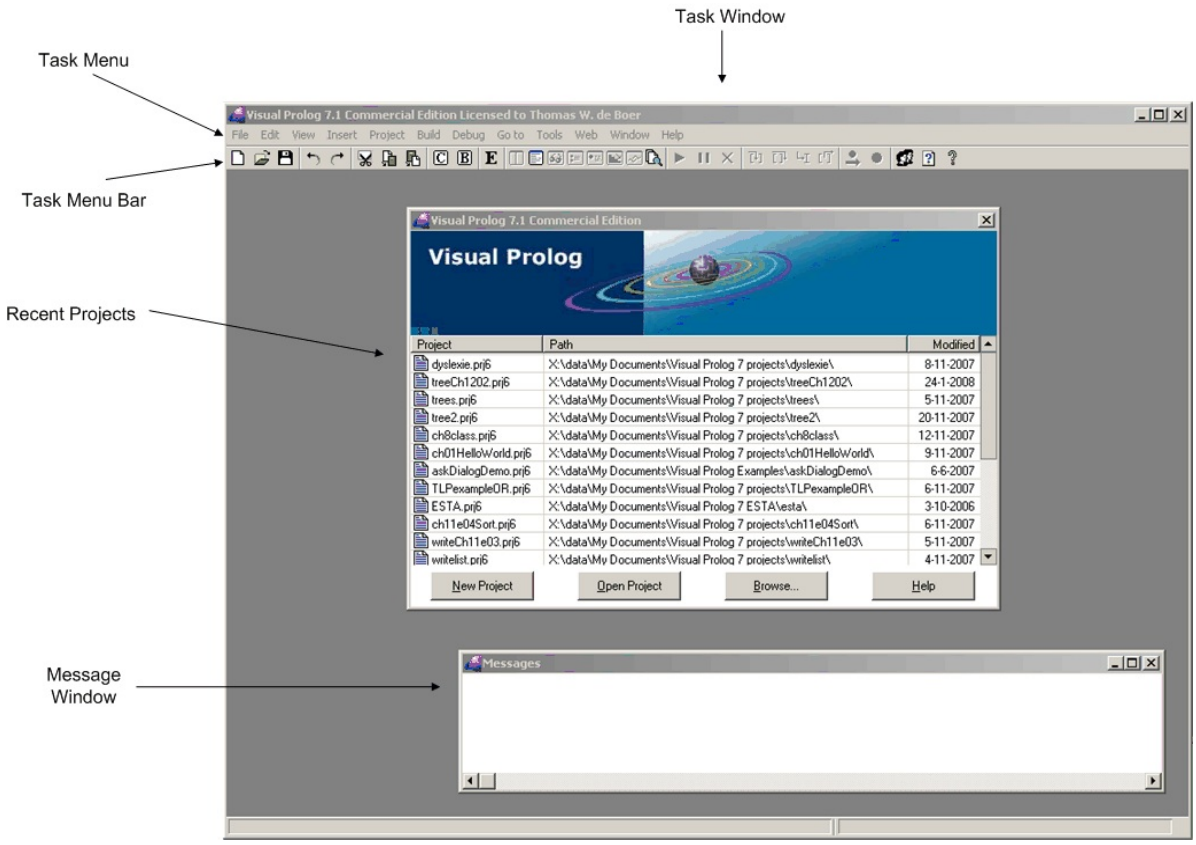


Figure 1.1 The Integrated Development Environment of Visual Prolog

[1] This chapter is a slightly rewritten version of the first chapter of Prolog For Tyros by Eduardo Costa

[2] The screenshots are from Visual Prolog Commercial Edition Version 7.1

The IDE shows the usual parts of a Windows program. In the rest of this book we will refer to several parts of the IDE. These parts are marked in figure 1.1 with the names that are used within Visual Prolog. They are:

- The Task Window. In the IDE the Task Window is the window that is shown when you start a program. You could also call it the "Main Window". This window usually shows the menu options "File, Edit, View, ..., Windows, Help" on top.
- The Task Menu. It is the menu that will be familiar to you as it contains options like "File", "Edit", and the other well-known Windows menu options. This menu will also be referred to as "Main Menu".
- The Task Menu Bar. It contains icons for the most used options in the Task Menu.
- The Project Overview. It lists the names of the projects that have been processed in the IDE. When you start Visual Prolog for the first time, there probably will be no recent projects.
- The Messages Window. This window is typical for Visual Prolog and the IDE. Here you will find the messages from the IDE and later on you are going to write short messages in the Messages Window yourself.

A lot more can be said about the IDE, but this is enough for the moment. Now it is time to do something with the IDE. You are going to create a simple program. As programming is a project, I shall use the words program and (programming) project interchangeably.

1.2 Creating a project in VIP

Let us create an empty project, to which you will add functionality later on.

Step 1. Create a new project

There are two ways to create a project. When the Project Overview is still open, you can click <New Project> in the Project Overview. The other way (when the Project Overview is closed) is to click the option *Project/New* from the Task Menu, as shown in figure 1.2.

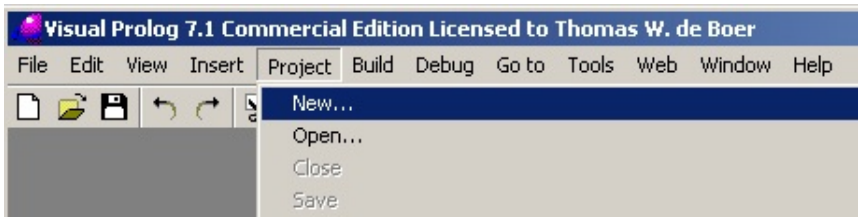


Figure 1.2 Task menu: Project/New

When you click the option, the IDE opens a new window so you can enter the Project Settings. As this window is meant for a dialog with the user (that is you), a window like this one is called a Dialog Window or shortly a Dialog. At the same time this Dialog acts like a form, because you will fill in some of its fields. That's why a window like this is also called a Form. Both Forms and Dialogs can be used as synonyms, they are more or less the same. You will learn about Forms and Dialogs in later chapters.

Now, fill in the *Project Settings* dialog as shown in figure 1.3.

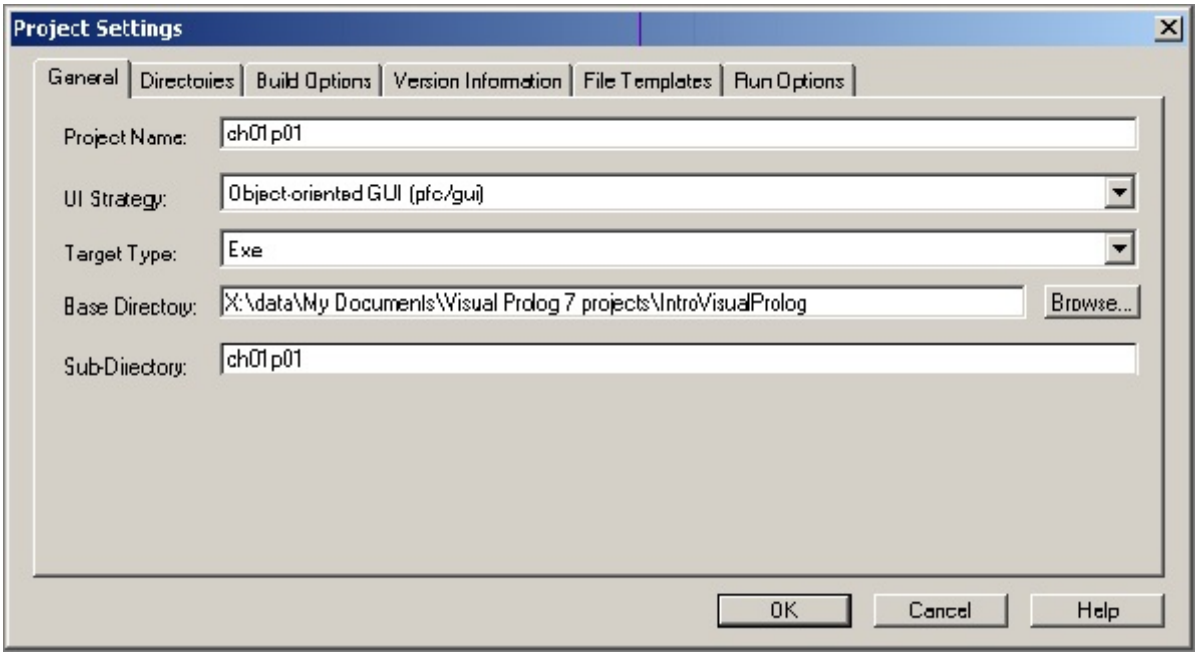


Figure 1.3 Project Settings

You only have to fill in the Project Name as most of the fields are already filled by the IDE, but it is wise to take a closer look at the content.

- The first field <Project Name> will contain the project name. For the Project Name I chose “ch01p01". Of course you can choose another name, but I suggest that you use the same name.
- The next field is the UI Strategy. It indicates what kind of user interface you will use. The IDE suggests that you will use an Object-oriented Graphical User Interface (GUI) . It means that you will use the “standard” windows interfacing. That’s OK.
- The Target Type is the kind of program the IDE should produce. “Exe” is fine, as it means that the IDE wil produce a stand-alone Windows Program. The other option is DLL. If that means nothing to you, forget about it for now.
- The Base Directory is the folder where you want your programs to reside. The base directory is not the definite folder where this program “ch01p01" will reside; that folder is specified in the next field. The Base Directory is the folder where your program folders are located.
- In the Sub-Directory you give the name of the folder where this program “ch01p01" should be stored. This folder will be created, if necessary, in the Base Directory. When you typed the name of the project, the IDE assumed that the name of this sub-directory would be the same. But you can change this, if you want. For now we suggest that you keep the names of project and sub-directory the same. By the way, you understand by now that we use the words “folder” and “directory” interchangeably.

There are more project settings. You see them when you click the other Tabs (<Directories>, <Build Options> et cetera) but in this chapter we only need to fill in the fields in the <General> tab. When you are satisfied with the project settings, press the <OK>-button and you will get the Project Tree. The Project Tree is an overview of the files of your programming project. It is set up like the directories in Windows Explorer, in fact it shows shows a part of the directory structure on your hard disk.

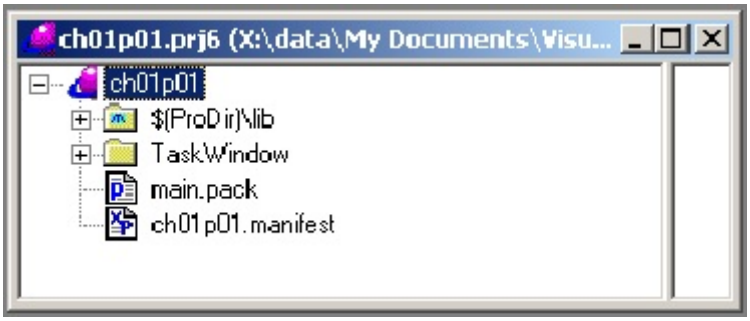


Figure 1.4 The Project Tree

The Project Tree is also called the Project Directory. It shows the set of folders and modules in which the IDE keeps track of your program. When you created the new project ch01p01, a directory was created somewhere on your hard disk that is resembled in the Project Tree. The root name of the Project Tree is “ch01p01". It is the name of the project that you entered in the project settings. It is not the name of the sub-directory that you indicated in the project settings. In fact you don’t see the name of the directory in which the program is stored on your hard disk. The IDE only copes with the name of the project. In this case it is “ch01p01". In the root directory of the Project Tree are two folders, named “$(ProDir)\Lib” and “TaskWindow”. They were created by the IDE, you can skip these for the moment, we will come back to them later. In the project directory there is also a file, named “main.pack". This file contains the basic part of your program. There is also a file named “”ch01p01.manifest”. information about your project. This file is created when an option is checked in the project settings. You can find this optionunder the tab “Build Options” when you create a new project. A Win32 side-by-side assembly contains a collection of resources—a group of DLLs, windows classes, COM servers, type libraries, or interfaces—that are always provided together with applications. These are described in the assembly Manifest files. For now you may take a look in this file, but I use to leave this one to the professionals.

You should be aware that there is not yet something like a program. To create the program of this project, we use the generating power of the IDE. Based on what you declared about the project, the IDE generates the Prolog code that is necessary. Although you only declared the name of the project and maybe the directory where it should reside, the IDE knows that it is a Windows program so it generates the code to produce a Windows program with a Main Window (or Task Window) and the menu items File, Edit, et cetera that you know from other Windows applications.

So now it is time for step 2: Build, compile and execute the program

To generate the Prolog code, choose the option *Build/Build* from the IDE Task Menu, as shown in figure 1.5. When you click this option, the IDE generates the Prolog Code. It generates several messages that you see in the Messages Window. Building the project takes several seconds. If the IDE asks you if you want to add something, choose<Add> or <Add All>. Watching the messages in the Messages Window gives you an idea of how much code is necessary to create an simple Windows program. The IDE generates the code, compiles it and finally links the parts of the program. When all is done, the message “Project has been built” is shown.

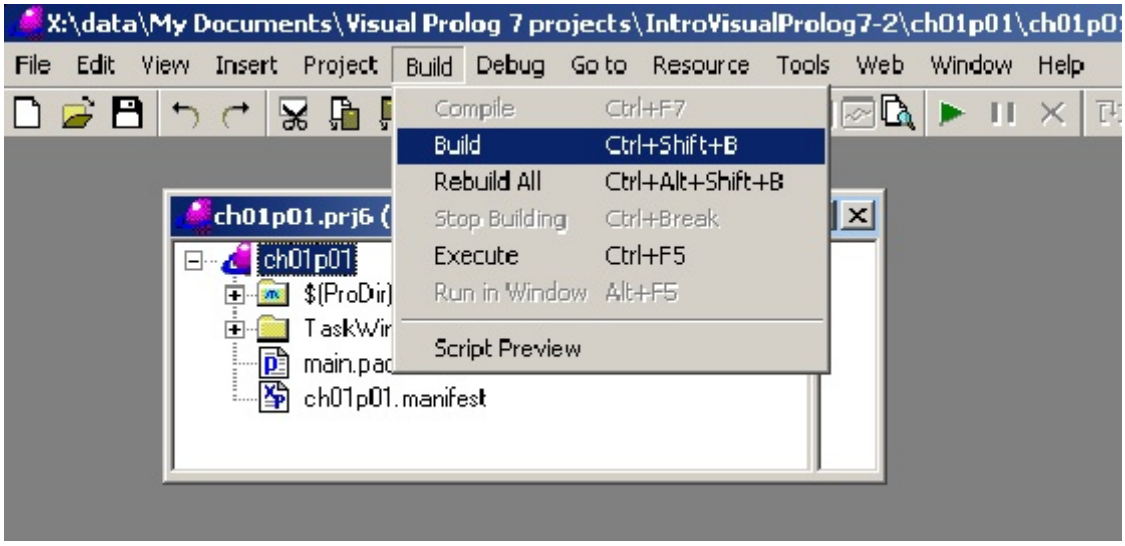


Figure 1.5 Build the project

When you have built the project (that is you made the IDE generate the Prolog source code), take a look at the Project Tree. It has grown. Building the project means generating code that is placed in several files. Take a look at figure 1.6. The IDE has added among others the file "ch01p01.pro". This is the file where you will find the kernel Prolog code of your program. The other files also contain Prolog code. To see what is inside, simply doubleclick on the name, but please don't change the code.

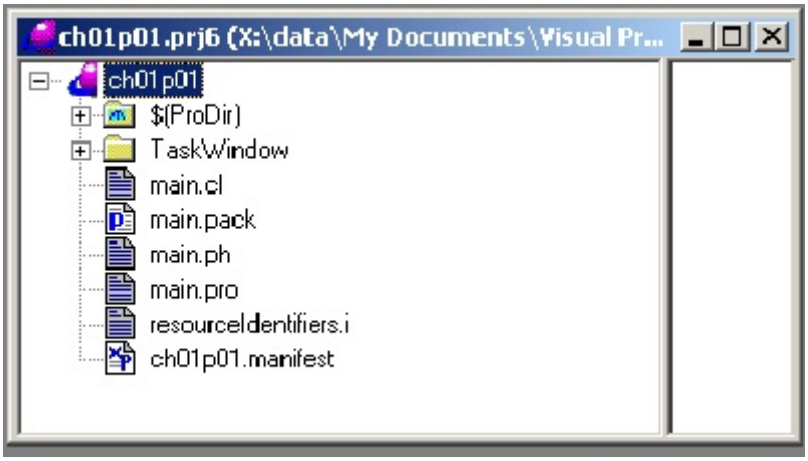


Figure 1.6 The Project Tree after the project has been built

Now that you have built the program, you are ready to execute it. To execute it, choose *Build/Execute* from the Task Menu, and a window like the one in figure 1.7 will pop up on the screen.

You see that the IDE created a "normal" Windows-looking program with the usual menu-options File, Edit, Window and Help. When you click one of these, nothing happens. That is because you did not specify what should happen. Only a few options really work in this program. The Minimize and Maximize buttons in the upper right corner and the menu option File\Exit. These options work because they are standard options that are the same in every Windows application. For the IDE it was not hard to guess what code you wanted here. One last remark concerns the Messages window. It is typically for Visual Prolog. You don't have to bother about it yet. By the Way, your program

ch01p01 is a separate program from the IDE. When you close ch01p01, the IDE is still open. When you close the IDE, the program ch01p01 is still running.

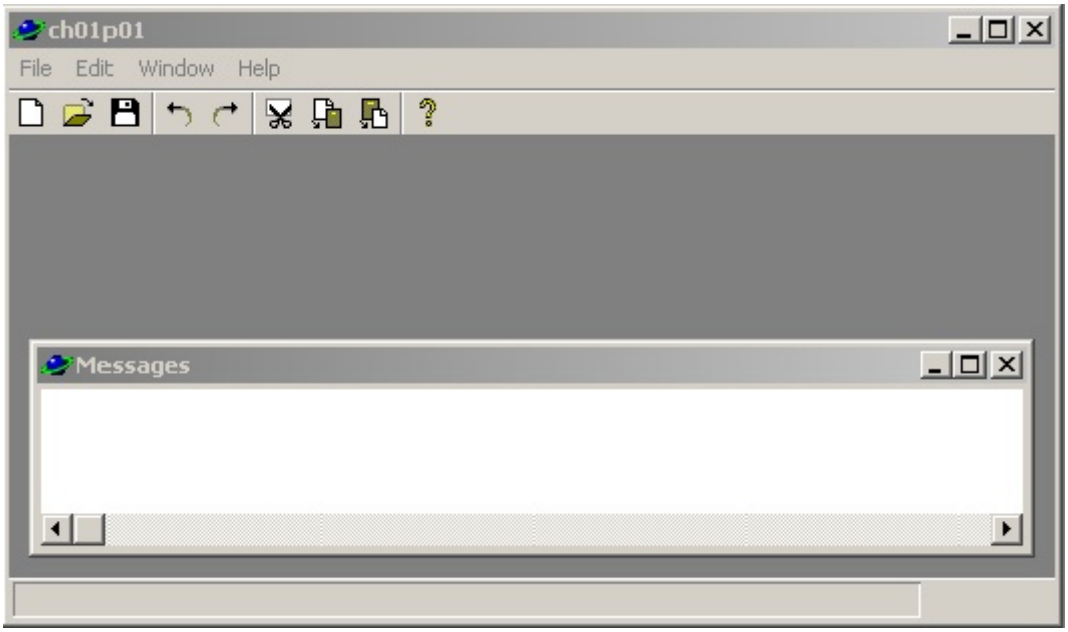


Figure 1.7 Execution of program ch01p01

Congratulations. You made your first Visual Prolog 7 program. Or maybe better said, you arranged things and the IDE of Visual Prolog generated the code for the program so you could execute it. This is the way you will learn to use the IDE. You specify what you want, the code is generated, occasionally you change the code or add something to it. This is the way to create computer programs with an advanced tool like Visual Prolog and its Integrated Development Environment. This way of programming seems excessive and laborious. The reason for it is, that programming in an object oriented programming language environment using a graphic user interface is quite complex. It is very difficult to keep track of every detail. That's why the IDE does a lot of bookkeeping for you. It makes life easier.

It may be interesting to inspect at this moment the files on your hard disk with Windows Explorer. When you inspect the directory where you put ch01p01, you will see that the IDE created not only the files that you see in the Project Tree, but a lot more. Especially note the two folders "Exe" and "Bin". In "Exe" you will find the executable files, that are the programs that run on their own. Here you will find among others the file "ch01p01.exe", that is the executable code of your program. When you want to export the program to another computer, you will have to copy the .exe file and the other files from this directory. In the directory "Bin" you will find other files that the IDE needs to generate your program; they are not needed for export. When you want to export the complete project to another computer, you have to copy the complete project tree.

The Project Tree gives you some idea of the efforts that are to be made to create a Windows program. Also you can see that your program project consists of several files. This is typical for an Object Oriented Language like Visual Prolog.

Chapter 2 Forms[3]

The program you created in chapter 1 is a simple program. You can start it, minimize and maximize the Task Window and you can exit it. That's all. A real program should at least be able to communicate a little bit more with the user. In a Windows environment communication is, among others, done in special windows. There are several kinds of communication windows. The ones most used are called Forms and Dialogs. Because in VIP Forms and Dialogs look and are very much the same, I will not make a difference between them. In this chapter, you will add a form to the empty project that you have created in chapter 1. So please, when project "ch01p01" is not open, open it. To open a project use the option *Project/Open* from the Task Menu or choose project "ch01p01" from the Project Overview that is shown when you start Visual Prolog.

2.1 Create a form

A project may become very complex and will consist of a lot of files. Therefore it is wise to think beforehand about the folders in the Project Tree where you want to put the various parts or modules of your program. In this case we will not bother too much about this, we will do that in later chapters. Now we simply place the new form in the root of your project. Go to the Project Tree and highlight "ch01p01" by clicking on it. With "ch01p01" highlighted you are ready to create a form and it will be placed in the root directory of "ch01p01". To create a form, choose the option *File/New in Existing Package* from the Task Menu, as in figure 2.1.

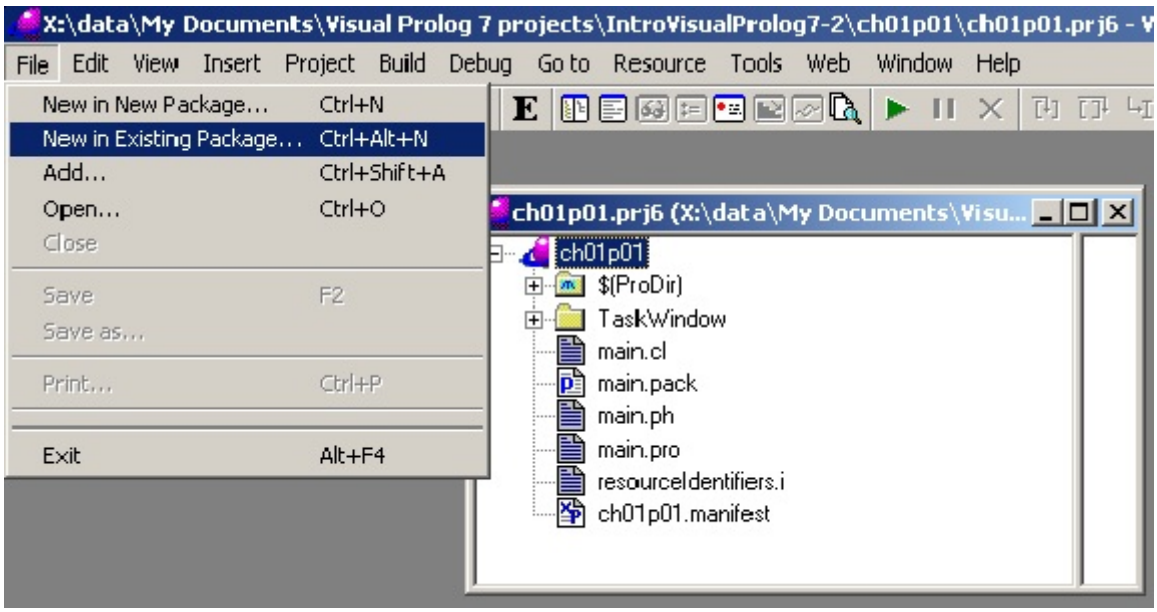


Figure 2.1 *File/New in Existing Package*

The IDE opens a window that offers the possibility to create several project items, see figure 2.2. The name of the window is "Create Project Item" (that is what you want to do) and it is a form with two panes. The left pane shows the types of items that you can create. Take a moment to review the

[3]This chapter is a rewritten version of the second chapter of Prolog For Tyros by Eduardo Costa

possible items. You can create a Package, a Class, A Dialog, A Form, just to mention a few. In the left pane you choose the type of item that you want to create. The right pane offers some options for the item.

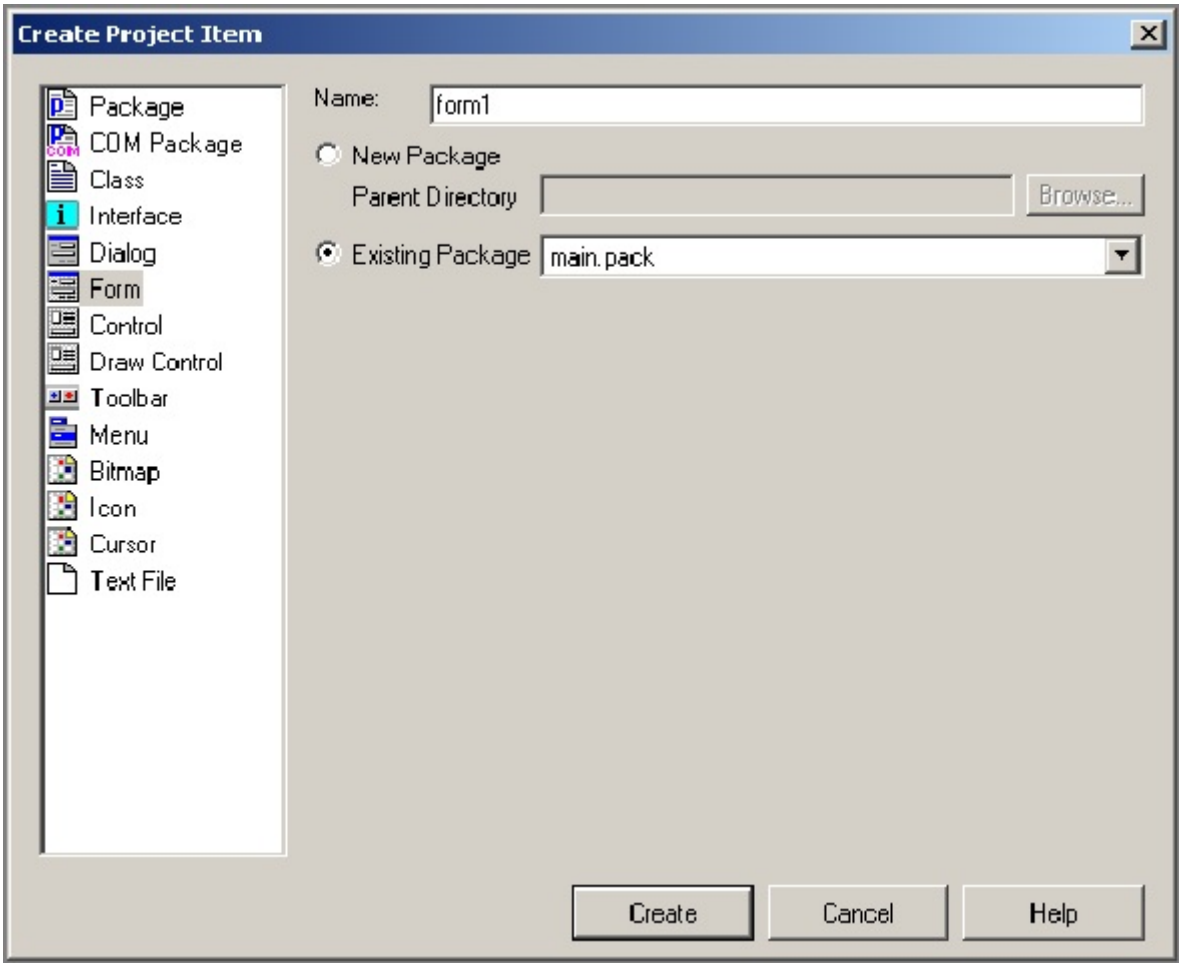


Figure 2.2 Create Project Item - a form

Here we want to create a form, so in the left pane you highlight *Form*. In the right pane the IDE opens the fields for a few things it needs to know about this form. A window like the one in Figure 2.2 with the name “Create Project Item”, is called a dialog window, in this case it is the Create Project Item dialog. The IDE uses dialogs to hear from you what you want and where to put it.

For the new form name I chose “form1". That is a very bad name as it says nothing about the form. In general you should think of meaningful names. But I think that here it will do. Type the name as shown in figure 2.2. Be sure that you highlighted “ch01p01" in the Project Tree to place the form in the project root. If necessary you can click on “Existing package” and accept the name “main.pack” that the IDE suggests. When your dialog looks like the one in figure 2.2, everything is complete and you click the <Create> button to make the IDE create a new form.

When you click <Create> the IDE opens four small windows that work together to support you in creating the new form. These windows are titled “form1" (that is the name we chose for the new form), “Layout”, “Controls”, and “Properties”. To understand what is happening here, you should

know something about the basics behind a Windows program. A Windows program consists mainly of windows (this will not surprise you :-)) in which the user can do something. E.g. in a form you can fill in your name in a field, you can click on a radio button, you can choose from a dropdown menu, you can click an <OK>-button. These “things” with which the user works are called “controls”. A window, like the form we are creating, can be filled with many controls. The available types of controls are shown in the window with title “Controls”. You are free to place a control anywhere on a form. To do that, you need a prototype of the new form. That prototype is the window with title “form1". You use this window to design and edit the new form, I call it the Form Edit Window. In creating the form you can use the options in the window “Layout” to change the layout, e.g. to justify a group of buttons. Every control in a form has a series of properties. E.g. a button has a place on the form and certain dimensions. The properties of a control are shown in the window “Properties”. Try it! Click on the <OK>button in the edit window and watch the content of the properties window change. Click anywhere on the form in the edit window (but not on a button!) and you see the general properties of the new form. Change the size of the Form Edit Window by pulling a side and watch the dimensions in the properties window change. The four windows work together, but you are free to place them anywhere on your screen.

Let’s take a closer look at the four windows.

The window titled “Controls” (figure 2.3) is in fact a toolbar. It shows icons for every control that can be put on the form. On the top row you see from left to right icons for a button, a check box, a radio button, a static text field and an edit field. When you move the cursor over the icons, little text boxes show their meaning. To add a control to the form, you click the icon in the Controls Toolbar and then click in the forms editor at the place where you want the control to be. When you click an icon, the cursor changes to indicate the type of control that you have chosen.

Figure 2-3 Controls Toolbar

The window titled “Layout” (figure 2.4) is also a toolbar. The icons represent different ways to adjust the layout of the controls on the form. On the top row you see icons for left, center and right justifying. When you select a group of controls on the form and then click one of these icons, the positions of the controls are justified. When you move the cursor over an icon, its meaning is displayed. Especial handy are the icons in the bottom row: they are for equalizing the spacing between controls and for equalizing the dimensions.

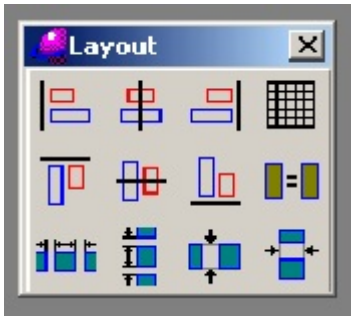


Figure 2.4 Layout Toolbar

The window titled “form1" (figure 2.5) is the Form Edit Window. It is a prototype in which already the three buttons <OK>, <Cancel> and <Help> are put by the IDE.

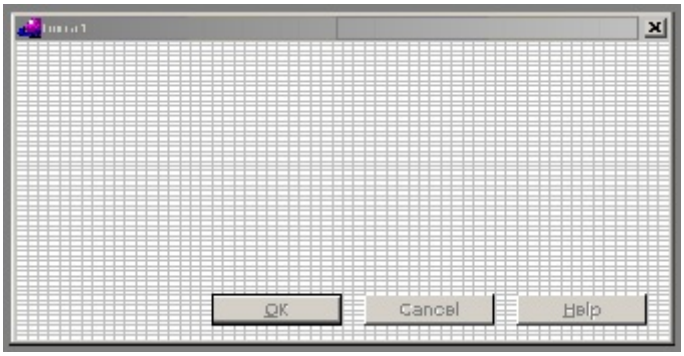


Figure 2.5 Form Edit Window

The fourth window titled “Properties” contains the properties of the selected item on the form. When there is no item selected, the properties of the form are shown. When you don’t see the properties of the form, click anywhere in the form (but not on the buttons) and the properties will look like they are in figure 2.6. The properties tell you the title that is shown in the title bar, the coordinates X and Y where the form will be placed on the screen, the Width and Height of the form and many more characteristics of this form. Also you can set properties by changing the value True into False. E.g. for (not) showing the title bar, a closebox, et cetera. Feel free to experiment a little with controls and properties, you cannot do much harm. Please note that the name in the Title field is the name that appears in the Title Bar of the form. It is not the name of the form, that is “form1". Feel free to change the title of the form.

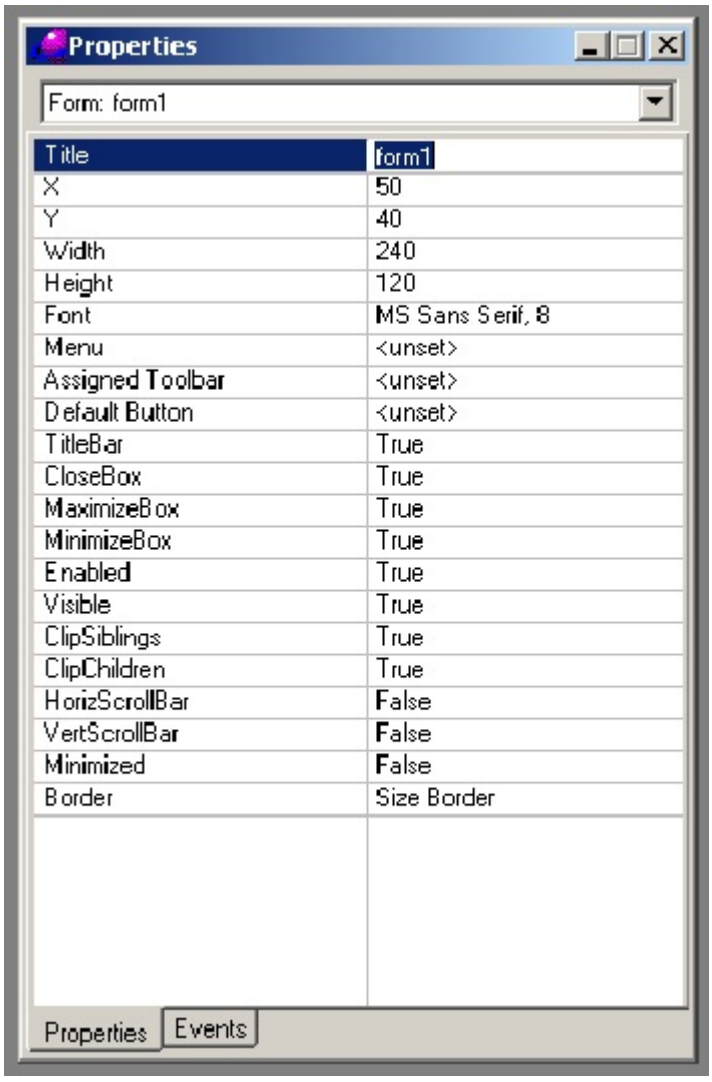


Figure 2.6 Properties window

The IDE has already placed three buttons (<OK>, <Cancel>, <Help>) as these are very common in Windows-windows. When you want to change the size of the form, click and hold the mouse at lower right corner and drag it, as you do when you resize any Windows-window. We call this a prototype as it shows the general characteristics of a form. It has a name on top, room to work in and three often-used buttons. This is the way the IDE works. Whenever you create something in the IDE, the IDE comes up with a kind of prototype that you can use to modify. At this moment the form only contains three buttons. Normally you would enter other controls, but for now we are satisfied with the form as it is. In the background the IDE has already created the necessary files for this form. So when you accept (as we do) the form without changes, you can close the form editro by clicking the cross button in the upper right corner. If you change something, the IDE will ask you if it should save the changes that you made.

When you now look at the project tree, you see that there are four files added. They all have the name “form1”, but they differ in their extensions. See figure 2.7.

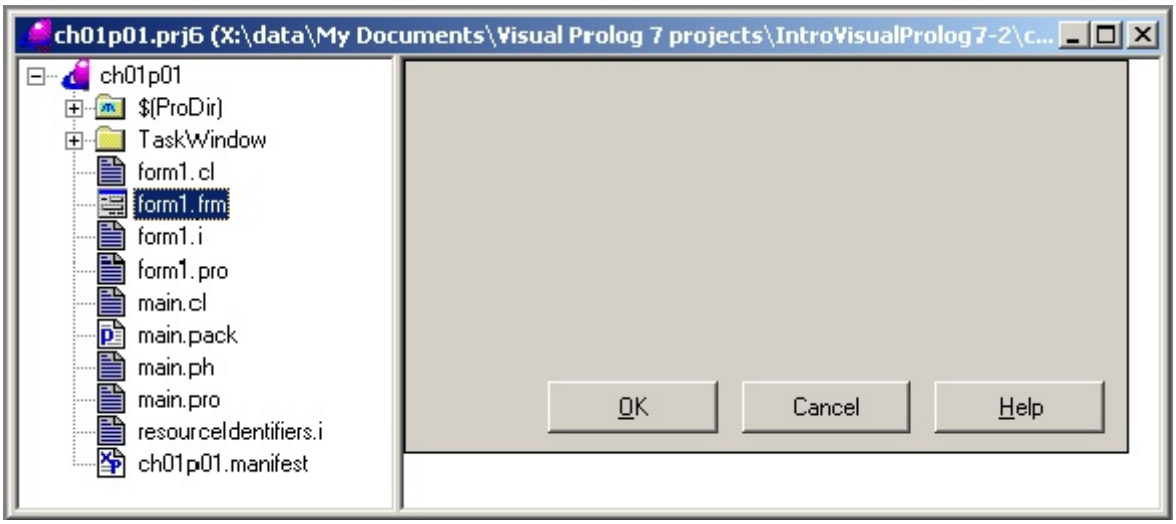


Figure 2.7 The files for “form1"in the Project Tree

In these four files the information about the form is stored.

- In “form1.frm” the prototype that you created in the editor is stored. Highlight “form1.frm” in the Project Tree and you will see the prototype in the right pane. When you double click on the name “form1.frm”, the form editor opens again.
- In “form1.cl” the class characterisitcs of the form are stored. Visual Prolog is an Object Oriented Programming Language. That implies among other that the programm is split up in so-called classes. Each class can be thought of as a module that contains a part of the program. More about that later. For now think of this file as the container of the general characteristics of your form. When you higlight “form1.cl”, you will see the predicates of this form. Don’t bother too much about that now.
- In “form1.i” the interface of the form is stored. Because the program consists of several classes, each class must know how to communicate with other classes. This is specified in this file.
- In “form1.pro” you will find the Prolog code for the form. When you double click on “form1.pro” the IDE opens a text editor with the code. Take a look but plwease don’t change anything.

In this case, we accepted the prototype as it was created by the IDE. Take care that when you change something (and you will do that probably always), the file “form1.frm” is changed, but the other files are not changed automatically. To do that, that is to generate the necessary Prolog program code you choose the option Build form the menu Build. This is comparable with what we did in chapter 1. There you specified the properties of a program (giving it a name and a base directory) and then ordered the IDE to generate the code. In this case you change a form and now we make the IDE generate the Prolog code. When you click *Build/Build* the IDE wil update the files: “form1.cl”, “form1.i”and “form1.pro”. You may want to open the file “form1.pro” to see the code the IDE has generated. Simply double click on the name. An editor opens with the Prolog code of the form in it. You can scroll to see the code, but please do not change the code. The IDE keeps everything about the form “form1" in the four files mentioned. The file “form1.frm”contains the design prototype, the other files contain the Prolog code. If you want to change something in “form1", you open the design file and change whatever you want. Then close and save the design and generate the code again. This way of working will come back very often.

2.2 Enable the Task Menu option.

By now you have a program, you have a form in it, but it still doesn’t work. When you now compile and execute the program, nothing happens with the form. The reason is that we did not yet specify when the form should be opened. In a Windows program it takes a so-called event to make something happen. There are many events possible, but the one you are probably most familiar with, is the click with the left mouse button. A click with that button may open or close a window or it may activate a menu option. Your program needs to know when to open the form that you created. You are free to choose any event that you like, but here I suggest that you choose the event that the user clicks with the left mouse button on the menu option *File/New* in your program.

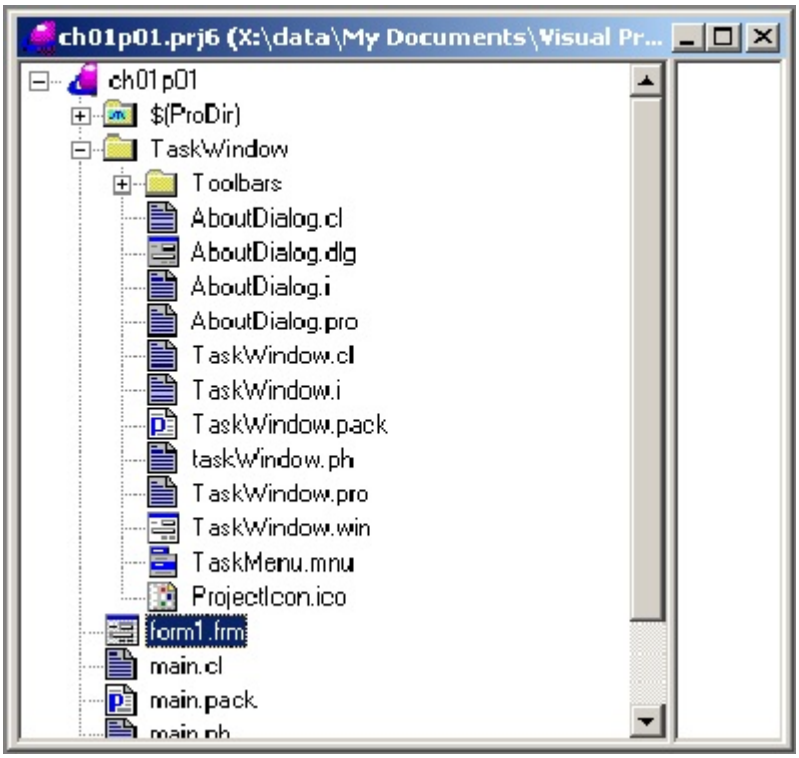


Figure 2.8 Contents of folder TaskWindow

When you ran the empty application in chapter 1, you certainly took notice of the fact that the *File/New* option was disabled. It was grey, that means in Windows that the option cannot be used. If we want to use it, it must be enabled. To find the place to enable File/New, go to the Project Tree. In the Project Tree you see a folder icon labeled “TaskWindow”. This folder contains everything that is needed to create the Task Window (or Main Window) when you start the program. Please open the folder “TaskWindow”. You open it by clicking on the plus-sign in front of the name or by double

clicking on the name. In the folder you find typical parts of the Task Window of the program, that is also called the Main Window. See figure 2.8. There are several files, and maybe you already see a familiar pattern. E.g. there are four files with the name “AboutDialog”, but with different extensions. The file “AboutDialog.dlg” contains the design of the window that pops up when you click the option *Help/About* when you run your program. Double click on “AboutDialog.dlg” and the IDE opens a dialog editor that looks very similar to the one that we used to create “form1". The other files contain the Prolog code for the AboutDialog window.

For now we are interested in the menu bar of the program because that is where the option File/New resides. So you click on “TaskMenu.mnu”. This is a kind of prototype of the Task Menu. You can compare it to “form1.frm” that is a prototype of a form. If you want to change something in the main menu, use the file “TaskMenu.mnu”.

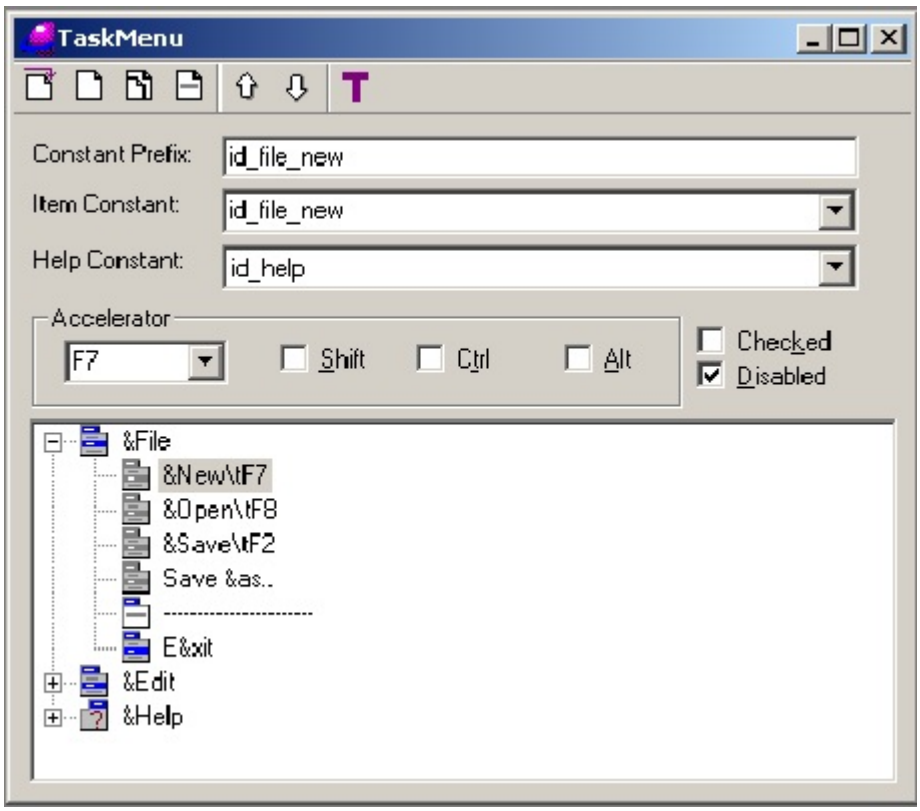


Figure 2.9 Editor for the TaskMenu

You want to enable an option in the menu, so you double click on the name TaskMenu.mnu. The IDE opens the TaskMenu Editor. It is shown in figure 2.10. Here you can enable, disable, append and remove menu options, define short cuts et cetera. As you are interested in the option File/New of the menu, you open the &File option in the lower part of the form by clicking on the plus-sign. Subsequently single click on &New/tF7 to select it. The IDE shows the characteristics of this menu option in the upper part of the form. There you find the checkbox “Disabled”. Uncheck this box and the menu option will be enabled. Save the new menu options and close the TaskMenu Dialog.

2.3 In CodeExpert, add code to Project Tree item

The menu option *File/New* is enabled now. When you run the program you will see that it is no longer grayed. But when you click on it, nothing happens. That is bcause we have not yet added code to the menu option. Now is the time to add the necessary Prolog code to this option. To add code to the *File/New* option, go back to the project tree and look for the file "TaskWindow.win". In this file you manage and control the Prolog code that is linked to the various parts of the Task Window. The code is eventually placed in the file "TaskWindow.pro" You could go directly to "TaskWindow.pro" and enter code. But for now it is better not to do so. Do not place the code directly. It is better to use the part of the IDE that is called the Code Expert. The Code Expert keeps track of the places where code is inserted. When the program grows, this Code Expert will guide you. Trust him.

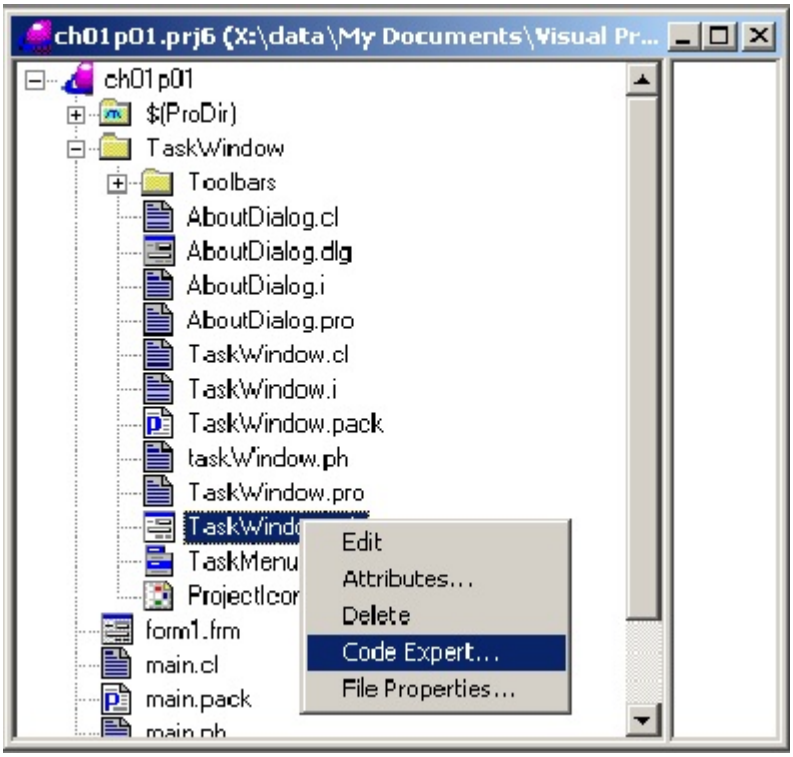


Figure 2.10 Open the Code Expert

Click on "TaskWindow.win" in the Project Tree with the *right* button of the mouse. That opens a floating menu. In this menu choose the option *Code Expert* (figure 2.10)

The IDE now opens the file "TaskWindow.win" in the "Dialog and Window Expert" and you see a list of the many options that you have in the Task Window of your program. Here you can specify many things. You see the Dialog and Window Expert in figure 2.11.

To add code to the menu option *File/New*, you first have to find it. It is hidden in the folder "Menu". Open it and you see the subfolder "TaskMenu". Open that one too and you see the standard menus that the IDE placed in your program. The options File, Edit and Help are represented here by id_file, id_edit and id_help. Open the id_file folder and single click id_file_new .

At the bottom of the dialog a button appears labeled "Add". When you single click this button, the Code Expert adds to your program some standard Prolog code. This code will be executed when the option *File/New* is clicked by the user. You don't see this code yet, but you will see that the Code Expert adds the name of something to "id_File_New". It added "onFileNew" and you may think of it as the name of a procedure that will be executed when you click the option *File/New*. Or in more technical words: onFileNew is excuted when the event happens that you click the option *FIle/New*.

As a sign that something was added to your program, the label on the <Add>-button changes into “Delete”. You can delete the code, if you want to, but please don’t delete it now. In the program the menu option *File/New* is now connected to the procedure with the name “onFileNew”. To see the added code, double click on “id_file_New -> onFileNew”. The Code Expert opens an editor with the Prolog code in “TaskWindow.pro” at the place where it put the code. The window is a text editor, the cursor is placed at the following piece of code:

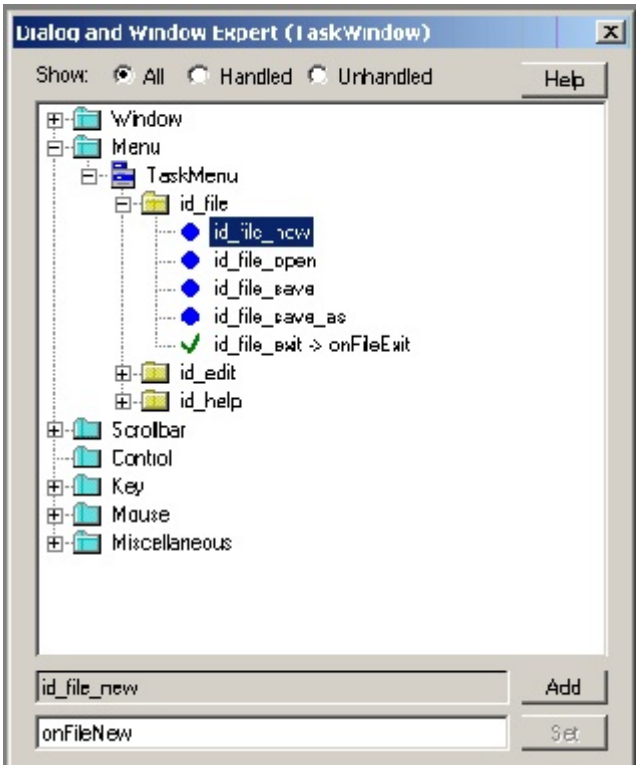


Figure2.11 Dialog and Window Expert (TaskWindow.win)

```
predicates
   onFileNew : window::menuItemListener.
clauses
   onFileNew(_Source, _MenuTag).
```

Now it is time to build the application. Close the editor with “TaskWindow.pro”, click on Build/Build in the Task Menu. When the IDE asks you if you want to insert a certain pack, simply say “yes”. When the message “Project has been built” appears in the Messages Window go to the Project Tree, right click on “TaskWindow.win” and use the Code Expert to go to the inserted code for “onFileNew”. This may seem Too Much Work (and to be honest, it is) but it makes you feel at home in the IDE and it is wise to build a project frequently.

In the editor you modify the snippet to:

```
clauses
        onFileNew(Source, _MenuTag) :- NewForm= form1::new(Source), NewForm:show().
```

Please take care to type the modification exactly as it is shown here. Take care of upper and lower case letters and of the underscores, commas and full stops. Did you notice that you have to remove the underscore in front of “Source”?

Close the editor and save the changes. Then build the program again by choosing the option *Build/Build* from the Task Menu, as in figure 1.4. After that, execute the program and you will see that, whenever you choose the *File/New* option, a new form is created. When you click *File/New* more than once, the forms are placed on top of each other. To see the underlying form, you have to drag the top one.

2.4 What is happening in the background?

At this time you may not be too interested in what is happening in the background. Yet I want to explain a few details. You will need to know them as the IDE sometimes may seem to do strange

things. What is happening in the background is twofold. When you click the <Add>-button, the IDE places two pieces of code in "TaskWindow.pro". Please open that file and try to find the two pieces but take care not to change anything. One piece of code you already know. It is the code that you changed in the previous section. You know by now how to get there with the help of the Code Expert.

When it was inserted it looked like:

```
clauses
        onFileNew(_Source, _MenuTag).
```

You changed it. The second piece of code is put in the part of "TaskWindow.pro" where you should only come as a visitor, but never come to edit. It is at the end of the code, it starts with the warning (in blue):

```
% This code is maintained automatically, do not update it manually. 13:14:32-10.7.2007
```

The %-sign means that to the compiler this is only a comment. A comment is text that is meant for a human reader. It is not executed, in fact the compiler simply skips it. Below this line in the code the IDE takes care of the code. Here it placed a line of code (you have to scroll almost to the end of the file to find it) that connects clicking the menu-option *File/New* with the piece of code that you changed. The line of code looks like:

```
addMenuItemListener(resourceIdentifiers::id_file_new, onFileNew).
```

In VIP this is called "adding a listener" as this line of code adds a so-called listener to the click on the menu item. You may think of a listener as a little demon that sits back and waits until you click the option *File/New*. At that moment the demon directs the program control to the clauses in the program that you changed (the clauses that start with "onFileNew") and this code is executed.

In the previous section you changed the code that was inserted by the IDE. What happens when you don't change that code? In that case the code acts as a "do-nothing" instruction, so you don't have to bother about it.

In this way the IDE controls which menu items are activated (by unchecking "disabled" and inserting a listener) and which are not and next to this it controls which activated items do something (by executing the code that you enter) and which items do nothing. And, maybe more important, when you want to change the program, the IDE helps you to find the relevant code. You can imagine that in a large program it is quite a job to do this kind of bookkeeping.

But there is a price for this service. When you want to change the program code of the Task Window, you should never do it on your own; always use the Code Expert, otherwise things may become corrupt. Especially you should take care when you <delete> code that was inserted by the IDE. E.g. when you want to delete the code for "onFileNew" you open the "TaskWindow.win" in the Dialog and Window Expert (as you did in the previous section) , look for "id_file_new -> onFileNew", highlight it and then click on <delete>. What happens is that the IDE deletes the line of code for the listener, but it does NOT delete the program code that you changed. The reason is that you probably have changed that code. Therefore it is difficult to decide what exactly should be deleted. A second reason is that by deleting, the IDE could destroy code that you want to keep to use it later or

elsewhere. That's why you must delete that code yourself by hand. Things may go wrong in this case. There are two possibilities

1. You delete the listener but not your own code
   When you build the project in this case you get a Warning that there are unused predicates in your program. The warning points at your own code, that is not used anymore as there is no listener to activate it. It is a warning because the remaining code doesn't harm the program, it is simply not used.
2. You delete your own code but not the listener
   When you build the project in this case you get an Error that there is an unused identifier (say: variable) in your program. The listener points to an identifier variable (onFileNew) that it needs to react to clicking *File/New*. But you deleted that identifier so it is not found in the program code so the listener cannot direct the program control to it. That's an error, you should repair it by entering your own code or by removing (via the IDE) the listener.

2.5 A mouse event

Clicking on *File/New* is an event, but events are not restricted to clicking menu options. In this section you are going to add another mouse functionality to the program. More specific: you are going to extend your program so that it will react when you click the left mouse button anywhere in the form. Clicking the mouse button generates a so-called mouse event.

When you click a mouse button, in fact two events take place. You push the button down and after that release it. These events are known as MouseDown and MouseUp. Windows is capable to detect both events as different events. Here we will use MouseDown and neglect MouseUp. The place where we want to make MouseDown trigger a reaction from the program is in the form "form1.frm". That means that we must insert the needed program code in the form.

Go to the project tree and click with the right mouse button on "form1.frm". The IDE opens a floating menu. From this menu you choose the option "Edit". The IDE opens the form editor that you already know. Another way to open this editor is to double click on "form1.frm".

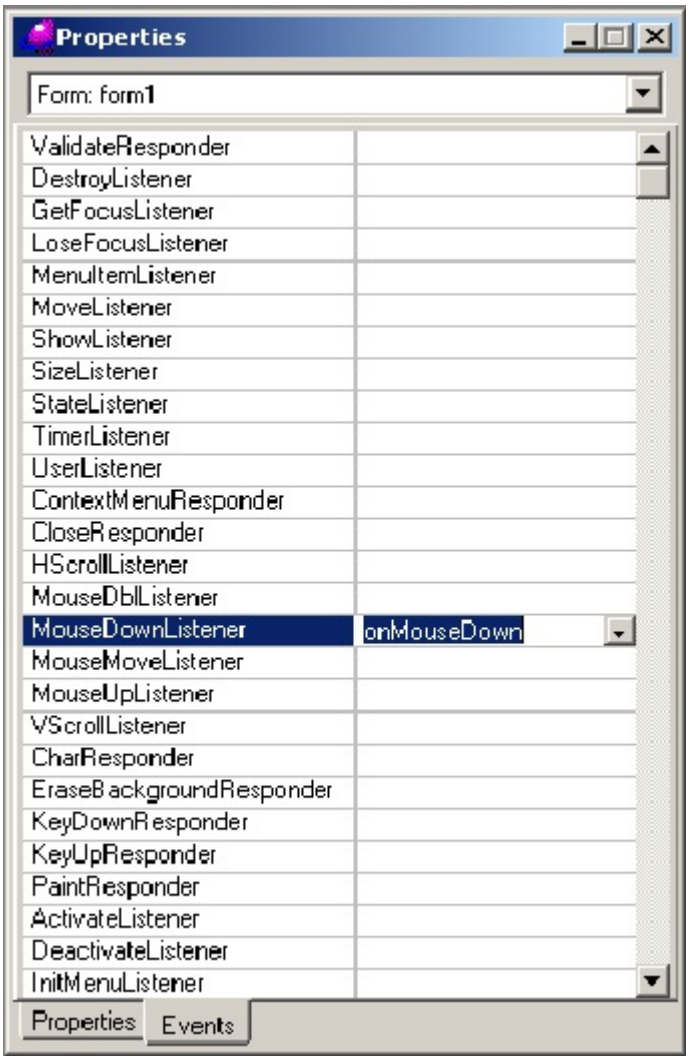


Figure 2.13 Events for "form1"

We want the form to react to the event MouseDown. So we need to find the event and connect it with the appropriate program code. If necessary click somewhere in the form (but not on the buttons) to bring up the properties of the form. In the properties window there are two tabs at the bottom. One says “Properties”, the other says “Events”. Click on “Events” and you see a list of possible events. Look for the event MouseDown, it is in the middle of the list. Click in the field to the right of the text “MouseDown”. The field changes into a listbox. Click on the arrow to reveal the options. There is only one, named “onMouseDown”. Choose this option. Then double click on “onMouseDown”. As a result the IDE generates some standard code for this event and opens opens the code editor. In the code you see the code that reacts to the event MouseDown. The cursor is placed next to it. The code looks like:

```
predicates
   onMouseDown : drawWindow::mouseDownListener.
clauses
   onMouseDown(_Source, _Point, _ShiftControlAlt, _Button).
```

The IDE understood that you want something to happen at MouseDown, so it inserted code for that event. This is comparable to what happened in the previous section when you inserted code for File/New. You may think of the inserted piece of code as the header of a procedure with the name “onMouseDown”. So far so good, but of course the IDE has no idea what exactly must happen at MouseDown, so you have to insert that code yourself. You should replace the piece of code

```
clauses
        onMouseDown(_Source, _Point, _ShiftControlAlt, _Button).
```

with the following code:

```
clauses
        onMouseDown(Source, Point, _ShiftControlAlt, _Button) :-
           Window= Source:getVPIWindow(),
           Point= pnt(X, Y),
           vpi::drawText(Window, X, Y, "Hello, World!").
```

Please take care of every dot, comma, underscore, lower and upper fonts as Prolog is, just like any other programming language, very sensitive for typos. Especially take care to remove the underscores in front of “Source” and “Point”.

When you have inserted the code and saved the file, build the program, and execute it. In the program choose the option *File/New* to create a new form. Whenever you click at any point within the form, the program will write a famous greeting expression. The code in onMouseDown takes care of the event MouseDown. That’s why we call onMouseDown an event handler. Windows programming is mostly a matter of taking care of events by defining and coding the right event handlers.

Please take some time to consider what you did and saw. In the previous section you added code to File/New, here you added code to MouseDown. The place and names are different, but the pattern is the same.

- You open a prototype (TaskMenu.mnu or form1.frm)

- You select an event (File/New or MouseDown)
- You <Add> the code for the event
- You go to the inserted code to change it according to what you want to happen.

Let's draw some conclusions. There is a pattern in the way we are programming. When a program should show some action, you first think of an appropriate event. Then you think of the user action that will invoke this event. It could be a click on a button or a click on a menu option or a click anywhere. You create and enable the necessary item (e.g. a button) in a form or (e.g. an option) in a menu (or in whatever is appropriate). Next you use the IDE to insert a listener (or a so-called responder, that is another kind of listener) and some standard code and finally you change and/or insert the necessary code. Until now the code looks probably like some magical words to you, but the pattern should be clear.

Chapter 3 Simple user interfacing

In this chapter I will elaborate a little bit more on user interfacing. There are several ways to communicate with the user and you will need to know about it. At the same time it will give you experience in using the IDE.

Programs under Windows consist for a great part of code that is used for communication with the outside world and the user is an important part of it. So it is useful to know about how to communicate with the user. Besides it is an easy way to get to grips with VIP and the IDE. In this section we will show the user some messages and ask for input.

Writing a computer program is a lot of fun (and often quite frustrating :-)). But it is not a goal in itself. There is a user out there - you will have to talk and listen to that user. In this chapter I introduce some simple ways to do that. You will learn to create messages and create ways for the user to enter input. At the same time you will become more and more at home in the IDE and you will see patterns in programming.

As I like to go slowly, because I think you are a beginning programmer, I will be repeating things; telling the same story again (and again). If I go too slowly for you, you can skip parts of this chapter

When you use a Windows program, you are familiar with several popup windows that tell you things you need to know. In this section I introduce standard ways to talk to the user like notes, error messages and message boxes. They are available in VIP, so you can easily use them. “Available in VIP” means that there is a part in the IDE where you can find the code. In your program you will have to tell the IDE that, when it builds the program, it has to get that code and use it in your program. The code that you need is in a special so-called “class”. Why it is called a class is not important at this moment, I will come to that in a later chapter. For now you can think of a class as a set of procedures that have been written and can be used whenever you need them. Procedures are little programs that can be used from within your program. I deal with procedures in the next section. The class we will be using in this chapter is called “vpiCommonDialogs”. It contains over 30 procedures for communication with the user. If you want you can take a look in the helpfile: Go to the index and look in the index for “vpiCommonDialogs”.

3.1 About procedures

When you start programming, you will realize after some time that the code that you are writing must have been written a thousand times before by other people. Well, maybe not exactly the same code, but surely many times before people wrote code to open a form, to enter some input, to say “Hello” to the world. It would be nice when you could use these pieces of already written code in your program. Reusing existing code is an efficient way of writing programs.

There are several ways to implement reusing code. One way is to have snippets of code at hand and copy-and-paste them in your program when necessary. Another way is to include in your program a library of pieces of code that perform the needed tasks. In this chapter we will use the second way.

You add a library with pieces of code and refer in your program to the code that you need when necessary.

Such a piece of code is called a procedure. It is like a little program that performs a task and that can be used whenever you need it. A procedure has a name and you activate it by naming the procedure in your code. This is called “calling a procedure”. Let me give an example. Suppose you have to do some arithmetic in your program, say finding the sum of two numbers. Your program could look like this (this is not Prolog or any other program code, it is only an example!):

```
Number1 = 2
Number2 = 3
Answer = Number1 + Number2
```

In this code we give the variables Number1 and Number2 a value and then add the two variables and give the resulting value to Answer. The variable Answer will get the value 5. Now suppose that there is a procedure, called “sum”, that performs additions of two numbers. Now your program could look like:

```
Number1 = 2
Number2 = 3
sum(Number1, Number2, Answer)
```

In this code we give the variables Number1 and Number2 a value and then call the procedure “sum”. After the name “sum” there is a list of variables between brackets. These are called the arguments of the procedure. In this case there are three arguments. The first two (Number1 and Number2) are used to tell the procedure which numbers it should add. The third argument is the variable in which the procedure returns the result. In this case the variable Answer will have the value 5 after the procedure has been called. Of course this is a silly example because there are better ways to perform an addition but I hope you get the idea.

Groups of procedures are kept together, especially when they are related in some way. Groups of procedures are brought together in so-called classes. You may think of a class as a set of procedures that you can call from inside your program. In this chapter you will meet the class “vpiCommonDialogs” that contains procedures for (as the name says) common dialogs for communication with the user. Visual Prolog offers many of these classes. They are made available when you install VIP and the IDE uses them in compiling and linking. You don’t have to indicate where these classes can be found, the IDE knows where to look.

One final remark. In Prolog there is a shorthand notation for procedures. For the procedure “sum” in the example you can write “sum/3" indicating that the procedure “sum” uses three arguments. The name and the number of arguments are a unique identification of a procedure, so everybody will know exactly which procedure you mean. This is necessary because in the libraries that come with Prolog sometimes the same name is used for different procedures. These procedures then have the same name but a different number of arguments.

3.2 Writing messages.

Messages can be generated anywhere in your program. Here I will generate them by using the Taskmenu. Of course this is not the way you will use them later on, but for the moment it is a convenient way. Please start a new project, and give it a name, e.g. “Ch03p01".

The simplest way to tell the user something is to write a “note”. When you do, the user sees a small window with a short text in it like the one you see in figure 3.1

Figure 3.1 a note

To generate a message like that one, we first have to create an event. In this case, and in the rest of this chapter, the event will be a mouse click on a menu option in the Taskmenu. So first we will create a new option in the Taskmenu, then we will add the standard code and finally we will change the standard code to generate the message. This pattern will be repeated throughout this chapter.

Figure 3.2 the icon toolbar in the Task Menu Editor

To create a new menu option, go to the folder TaskWindow in the Project Tree, select TaskMenu.mnu and open it in the Menu Editor by double clicking on it. Just below the title bar you see seven icons, representing options in this editor (see figure 3.2). When you put the cursor on one of them, you will get the name. From left to right they are:

- New first Item
  Inserts a new menu item before the first item. In the menu in your program it appears left of the (former) first item or above it.
- New item
  inserts a new menu option under the selected item
- New sub item
  inserts a new menu option in a sub menu below the selected item
- New separator
  inserts a dotted line below the selected item
- Shift Up
  shifts the highlighted item one place up
- Shift Down
  shifts the highlighted item one place down
- Test
  shows the menu in the Task Window.

To insert a new menu option, highlight an existing option and click on one of the icons. Which icon depends on what you want to do. When you want to insert a new item, click the <New Item> icon. A small edit field is opened below the highlighted option and you can type the name of the new item. In the name of a new menu option, you can use two special symbols. The symbol & in the name, e.g. in &File, makes the next character ‘F’ a shortcut key for the option. You can place & anywhere in the name, the next character becomes the shortcut. Take care that you don’t use a shortcut that already exists. When you add \tF5 to the name, the function key F5 becomes a shortcut.

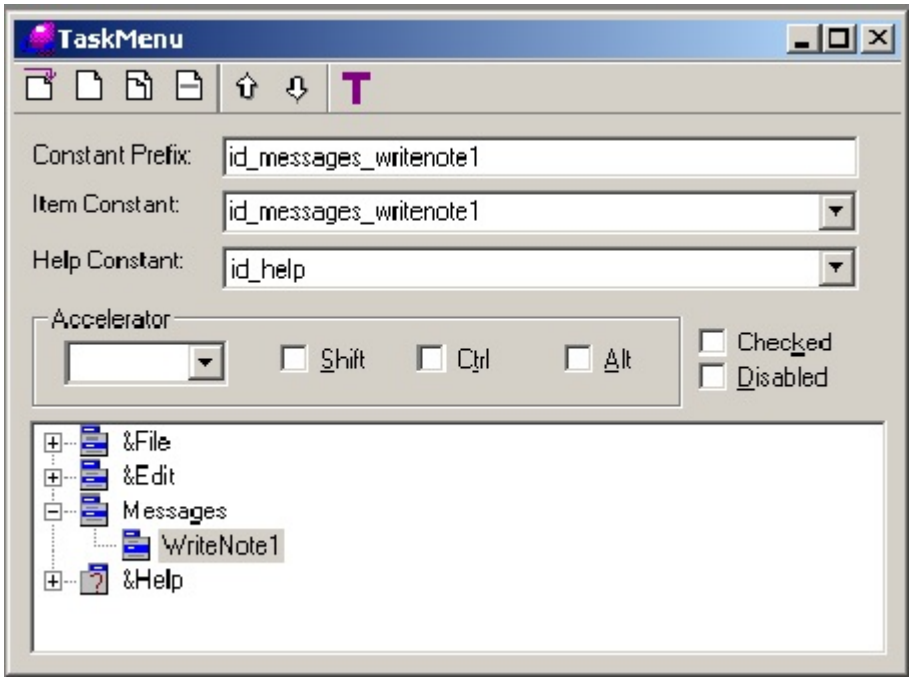


Figure 3.3 Task Menu with options added

For now we want to add a few menu options to generate messages. It seems appropriate to put these new options in a separate menu item. According to Windows tradition the options File and Edit should stay on the left side of the menu bar and Help should stay on the right side. So our new options should be placed between Edit and Help. To achieve that, highlight "&Edit" and click <New Item> on the toolbar. The IDE opens a small edit window where you can enter the name of the new menu option. Let us call the option "Messages". The editor should start to look like the one in figure 3.3. If you want you can add an ampersand (&) somewhere in the name. If you do so, e.g. you enter "&Messages", then Windows will allow the user to use the letter after the ampersand as a shortcut. Compare &File or &Edit. You can close the edit field with <Enter> or with a mouse click anywhere.

Below "Messages" we want to create a submenu. Highlight "Messages" and click <New Subitem> on the toolbar. Below the option Messages the editor opens an edit box where you can enter the name of the new option. Let's call this option WriteNote1. See figure 3.3. If you want you can check the new menu by clicking the option "T" on the toolbar. The Taskmenu changes into your new menu. Click on the Project Window to return to the original Taskmenu. Close the menu editor by clicking on the "x" in the upper right corner and don't forget to save the changes you made.

We now have a new menu option, so it is time to generate some code. Build the project and when the project is built, select TaskWindow.win, right click and choose the Code Expert. In the Code Expert open the folder Menu, open TaskMenu, open id_messages and <Add> the code for id_messages_writenote1. See figure 3.4

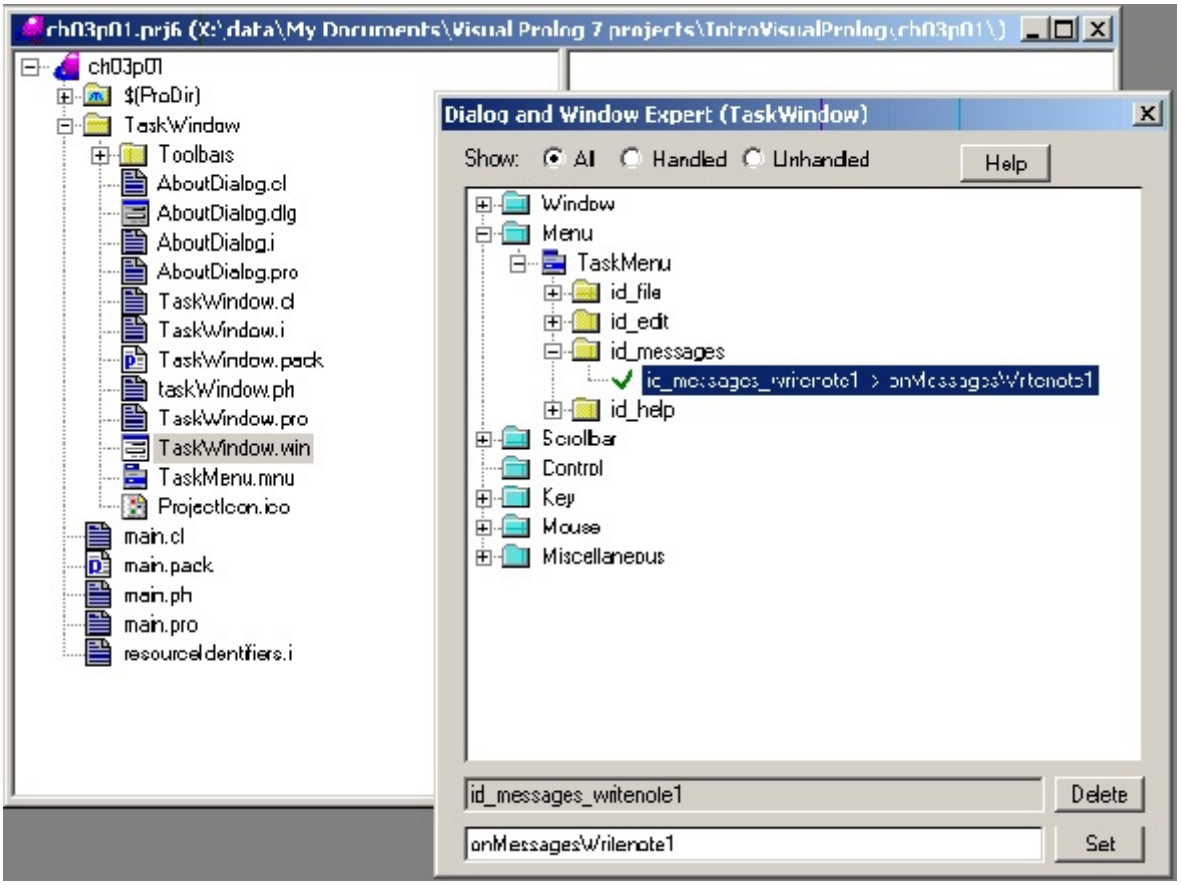


Figure 3.4 Adding code to a menu option

Then double click on

id_messages_writenote1 -> onMessagesWriteNote1

and the IDE opens the editor where you can enter the code to produce a message to the user.

In the editor you see the code that is inserted by the IDE. It looks like:

```
predicates
   onMessagesWritenote1 : window::menuItemListener.
clauses
   onMessagesWritenote1(_Source, _MenuTag).
```

Change it into:

```
predicates
   onMessagesWritenote1 : window::menuItemListener.
clauses
   onMessagesWritenote1(_Source, _MenuTag) :-
      vpiCommonDialogs::note("This is a message").
```

The last line tells Prolog to use the procedure with the name “note” from the collection in class “vpiCommonDialogs”. Between brackets is the string that should be displayed on the screen. Close the editor, save the changes and build and execute the project. In the Task Menu the new menu option appears, click it and next click on WriteNote1 and the message should appear. Click on <OK> to close the message.

When the message window appears, you will notice the word “note” in the title bar. If necessary you can add a different title. Let’s add another item in the submenu under Messages and call it WriteNote2. Add the code for this item and change the clauses into:

```
clauses
   onMessagesWritenote2(_Source, _MenuTag) :-
      vpiCommonDialogs::note("This is a title", "This is a message").
```

Save the changes, build the project and execute it. When you choose the option WriteNote2, you see that the first argument has become the title and the second one is the message body. So note/1 lets you send a (short) message to the user in a window with a preset title, note/2 allows you also to edit the title of the window.

When you look back at the code, you may think that it is cumbersome to type the name of the class “vpiCommonDialogs” every time you want to use a procedure from that class. It is and luckily there is a quicker way to make the procedures in a class available. Go to the Project Tree, go to the folder TaskWindow and double click on TaskWindow.pro. This is the file where the IDE keeps the Prolog code for the TaskWindow. Normally you will approach the code via the Code Expert, but by opening the file TaskWindow.pro, you get directly access to the code. Please be careful not to damage the code.

At the beginning of the file TaskWindow.pro you find these lines of code:

```
implement taskWindow
   inherits applicationWindow
   open core, vpiDomains
```

Pay attention to the third line that says “open core, vpiDomains”. This line tells the compiler that when it encounters a procedure name (like “note”) that it doesn’t see in your code, it should look in the classes “core” and “vpiDomains”. We know that we are going to use the procedure “note”. We know that this procedure is in class “vpiCommonDialogs”. So we can tell the compiler to look in class “vpiCommonDialogs” by adding that name to this line. Change the line into:

```
      open core, vpiDomains, vpiCommonDialogs
```

When you add the name vpiCommonDialogs to this line, you can remove it from the other lines. Change those lines into

```
      clauses
         onMessagesWritenote1(_Source, _MenuTag) :-
            note("This is a message").
```

And

```
      clauses
         onMessagesWritenote2(_Source, _MenuTag) :-
            note("This is a title", "This is a message").
```

respectively. You will find the lines when you scroll down to the end of TaskWindow.pro, above the code that is maintained by the IDE. Please make sure that you don't change that code! As an alternative you can of course also find the lines of code with the help of the Code Expert.

When the changes have been made, close the editor, save the changes and build/execute the project. You will see the messages appear as before.

You will have noticed that you can use the procedure "note" with one and with two arguments. This may seem confusing, but to Prolog it is clear: the procedure note/1 is different from note/2 because the number of arguments differ. The number of arguments is clear from the declaration or from the calling lines.

Another way to send the user a message is by way of the procedure "error", that is also contained in vpiCommonDialogs. The use is identical to the use of "note". There are two variants of "error". Error/1 takes as an argument the message text, error/2 allows you to also add a title for the message window. I suggest that you add two more options to the menu, WriteError1 and WriteError2, and that you add with the help of the Code Expert the following lines of code at the appropriate places.

```
predicates
   onMessagesWriteerror1 : window::menuItemListener.
clauses
   onMessagesWriteerror1(_Source, _MenuTag) :-
      error("This is an error message").

predicates
   onMessagesWriteerror2 : window::menuItemListener.
clauses
   onMessagesWriteerror2(_Source, _MenuTag) :-
      error("The title", "The error message").
```

Save the changes and execute the project and you will see the error message boxes pop up. Notice that there is a small difference with note/1 and note/2. The message window generated by note has and "I"-icon, the error message window has a white cross in a red circle to indicate an error.

The procedures "note" and "error" are for short messages. Next to them there is the procedure "messagebox/6". This one allows you to control the layout of the message window and also to keep track of the reaction of the user. To see how this procedure works, add another menu item, called WriteMessageBox, and add the following code:

```
predicates
   onMessagesWritemessagebox : window::menuItemListener.
clauses
   onMessagesWritemessagebox(_Source, _MenuTag) :-
      Answer = messageBox("TitleString", "Message line",
         mesbox_iconExclamation,
         mesbox_buttonsYesNo,
         mesbox_defaultSecond,
```

```
        mesbox_suspendApplication),
    note("You pressed button number ...", tostring(Answer)).
```

Please take care to enter everything just as it is shown. This may seem to be a lot of work, but it is worth the trouble. The procedure “messagebox” has six arguments, that allow you to control almost everything. The first argument (“Titlestring”) contains the text for the Title Bar of the message. The second argument (”Message line”) contains the text of the message. The other arguments are for controlling the icon, the buttons, the default button and the suspending.

The icon

There are four icons available: Information, Question, Error and Exclamation. You already know the Information icon from “note” and the Error icon from “error”. Here you choose one of the four by one of the names:

- mesbox_iconinformation shows the information icon. If you want, you can also use the number 0 as an argument;
- mesbox_iconquestion shows a question mark. Alternative is the number 1;
- mesbox_iconerror shows the error icon (number = 2);
- mesbox_iconexclamation shows an exclamation mark (number = 3)

These four names are called constants. They are predefined within Visual Prolog, that is they represent a value that is constant within the Visual Prolog environment. Instead of the names you can also use the mentioned integers (in fact these integers are the values that the constants represent), but your program will be better understandable when you use the names. You should place the icon name after the Title string and the Message string as the third argument.

The buttons

At the fourth place, you enter an indicator of the button(s) that should appear in the message window. The available names (and their equivalent numbers) are

- mesbox_buttonsok (= 0): shows only an <OK>-button so this looks like the note-message;
- mesbox_buttonsokcancel (= 1): shows two buttons, <OK> and <Cancel>;
- mesbox_buttonsyesno (= 2): shows two buttons labeled <Yes> and <No>
- mesbox_buttonsyesnocancel (= 3): shows three buttons labeled <Yes>, <No>, <Cancel>
- mesbox_buttonsretrycancel (= 4): shows two buttons labeled <Retry>, <Cancel>;
- mesbox_buttonsabortretryignore (= 5): shows three buttons, <Abort>, <Retry>, <Ignore>.

The default button

At the fifth place, that is the fifth argument, you can indicate which button will be the default button. When the message window appears, the user is supposed to close it by clicking a button. Instead she can also hit the <Enter> key. In that case it is like she clicked the default button.

- mesbox_defaultfirst (= 0) means that the first mentioned button (in the sequence above) is default
- mesbox_defaultsecond (= 1) the second mentioned button (in the sequence above) is default.
- mesbox_defaultthird (= 2) the third mentioned button (in the sequence above) is default.

You will understand that it is bad programming practice to indicate mesbox_defaultthird when there are only two buttons.

Suspending
The last position is to indicate whether the user can work in another application while the message is being shown.

- mesbox_suspendapplication (= 0) means that the user must react to the message before she can continue with your program. But she can work in other programs while the message is shown.
- mesbox_suspendsystem (= 1) means that the user cannot work with any application while the message is shown.

A last remark concerns the variable Answer. When you call the procedure “messagebox”, it not only puts a message on the screen, it also accepts the button that the user clicks and returns it. In other programming languages this is called a function. The number of the clicked button is returned and put in variable Answer. In the last line of the code this variable is used in a note for the user. Because Answer is a number and the procedure “note” expects a string, we have to transform Answer into a string with the function “tostring”. Buttons are numbered from left to right starting with the number 1. At this point I suggest that you start playing with the possibilities of the procedure messagebox. Change the icon and the buttons. It will give you a feeling for using these procedures. They are very useful tools.

3.3 Getting the user response

With messagebox you could receive input given by the user. Messagebox is one possibility but there are other ones. Their procedure names are ask/2, ask/3, getString/3, listSelect/5 and getFileName/6. To show how they work, I shall use options in the TaskMenu to trigger them and as they are meant to receive input from the user I shall add some code to show the input that was received. Just like I did with messagebox/6 where you saw the answer that the user clicked.

First let’s take a look at ask/2 and ask/3. Their use looks like the use of “note” and “error”. As triggering events for ask/2 and ask/3 add two menu items in the submenu below Messages in the Task Menu. I called the options ask2 and ask3. Generate the code and change it into:

```
predicates
   onMessagesAsk2 : window::menuItemListener.
clauses
   onMessagesAsk2(_Source, _MenuTag) :-
      ButtonPressed = ask("This is the question", ["Button0", "Button1", "Button2"]),
      note("You pressed button number ...", tostring(ButtonPressed)).
```

Build and run the program. A window should pop up that contains the question and three buttons as shown in figure 3.5.. The user can click one of the buttons and the number of the clicked button is returned to your program.

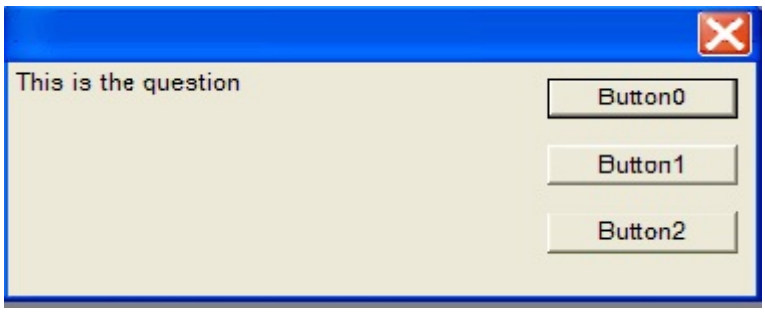


Figure 3.5 the pop-up window created by onMessagesAsk()

Please take a closer look at the piece of code that says:

["Button0", "Button1", "Button2"]
It is the second argument in the procedure call. In Prolog this is called a list. A list consists of a number of elements between square brackets separated by comma's. In this case the list contains the labels for the buttons. A label for a button is a string, so the names are put between apostrophes. I specified three button, but you are free to specify one or two buttons. You must specify the name of at least one button. Specifying four or more buttons is of no use, three is the maximum. You are free to choose any string for a label. Internally the buttons are numbered in the order as they appear in the list. The first button gets number 0 (zero). Take care, this is different from what happens with the procedure messagebox.

The procedure ask/3 is almost the same as ask/2. The only difference is that you also give the messagewindow a title. I presume that you can insert the necessary code without problems, so I only give the code:

```
predicates
   onMessagesAsk3 : window::menuItemListener.
clauses
   onMessagesAsk3(_Source, _MenuTag) :-
      ButtonPressed = ask("A convenient Title", "This is the question", ["Button0", "Button1"]),
      note("You pressed button number ...", tostring(ButtonPressed)).
```

You should know a little bit more about listening to the user as with ask/2 and ask/3. A user is supposed to click one of the available buttons. But sometimes she doesn't. She may hit the <Enter> key, she may click the <close>button () in the top right corner or she may hit the <Escape> key. With ask/2 and ask/3 hitting <Enter> is equivalent to clicking the first button, because the first button is the default with ask/2 and ask/3. When the user clicks the <close> button, this is equivalent to using <Enter>. When the user hits the <Escape> key, Prolog accepts this as if she clicked the last button. I don't bother about it here, but when you write a program, your response to the user input should take this into account.

Ask/2 and ask/3 are very useful when the question for the user has only a few multiple choice answers. Things become different when this is not the case, e.g. when you want to ask for a person's name. Then you ask for a string, but you don't know what it will look like. For input of a string there is the procedure getString/3. Calling this procedure looks like ask/2 and ask/3. This is what the code looks like. In my program the code is connected to the menu option GetString in the submenu of Messages in the TaskMenu.

```
predicates
   onMessagesGetstring : window::menuItemListener.
clauses
   onMessagesGetstring(_Source, _MenuTag) :-
      AnswerString = getString("Title", "Question", "Preset Answer"), !,
      note("Your answer is ...", AnswerString).
   onMessagesGetstring(_Source, _MenuTag) :-
      note("You clicked the <Cancel> button").
```

GetString/3 takes three strings as an input: a Title, a Question and a Suggested Answer. GetString/3 uses these to show a window with Title, Question, an edit box filled with Suggested Answer and two buttons labeled <OK> and <Cancel>. The user can accept the Suggested Answer or change it. The window closes when the user clicks <OK> or <Cancel>. Getstring/3 returns the string that the user entered. There are two possibilities. When the user clicks the <OK> button, the answer string is returned. When the user clicks the <Cancel> button, no string is returned. That's why the code contains two lines for the answer. It is too early to explain how this works in Prolog, this will be dealt with later. By the way, when you don't have a suggested answer, simply use the empty string " ". When the user closes the window in another way, the results are as follows. When she uses <Enter>, it is like she hits <OK>. When she uses the <close> button or <Escape> the result is like <Cancel>. This is different from how things are handled with ask/2 and ask/3.

Another way to input strings is listSelect/5. You can use this procedure when there are a restricted number of possible answers. An example is when the user must enter the country where she lives. With listSelect/5 you show the user the list of possible answers and she selects one of them. When you use this procedure you enter a title, a list of choices and the index of the preset selection. The results are returned in arguments four and five, these should be variables. Take a look at the code, that in my program is connected to the menu option Listselect.

```
predicates
   onMessagesListselect : window::menuItemListener.
clauses
   onMessagesListselect(_Source, _MenuTag) :-
      b_true = listSelect("Title", ["Choice1", "Choice2", "Choice3"],0,SelectedString, SelectedIndex),
      !,
      note("Your selection is ...", SelectedString),
      note(" ... with index ...",tostring(SelectedIndex)).
   onMessagesListselect(_Source, _MenuTag).
```

The IO-pattern of listSelect/5 is (i, i, i, o, o), the first three arguments are input, the last two arguments are output. As an input you give as the first argument the title of the window. As the second argument the list of possible choices are put together in a list of strings ["Choice1", "Choice2", "Choice3"]. The first one ("Choice1") has index 0 (zero), the next one ("Choice2") has index 1 (one) and so on. The third argument indicates which of the possible choices should be shown as the preset answer. The argument 0 after the list indicates that in the window the first item should be the preset answer. If another answer from the list should be the preset answer, you should give its index. If you don't have a preset answer, you should use -1. The variables SelectedString and SelectedIndex receive the answer of the user. That is to say, when the user finishes with <OK>. When she uses <Cancel> or the <close> button, no string is returned and the returned index is -1.

A last way of communication with the user is the familiar question to select a file. In VIP there is a useful procedure getFileName/6 to do this. It may seem complex, so if you want to skip it for now, that is OK. In my program I connected the code to the menu option GetFileName and inserted the following code in the program

```
predicates
   onMessagesGetfilename : window::menuItemListener.
```

```
clauses
   onMessagesGetfilename(_Source, _MenuTag) :-
      SelectedFile = getFileName("*.txt",
         ["Text Files", "*.txt", "Prolog Files","*.pro", "All Files", "*.*"],
         "Title: Please select a file",
         [],
         " ",
         _SelecFiles), !,
      note("You selected File ...", SelectedFile).
   onMessagesGetfilename(_Source, _MenuTag) :-
      note("You pressed the Cancel button").
```

Getfilename/6 needs the following input arguments. For the sake of readability I placed the arguments on separate lines. As you are going to select a file, getFileName wants a so-called filter that indicates what files it should show. In MSWindows most of the time we use the file extension (the part of the name after the dot) for this. The first argument indicates that when the window opens, it should show files with extension “*.txt”. The second argument is a list of paired strings. Every pair consists of a file type and the filter that is needed to select that file type. E.g. text files have (in general) the extension “*.txt”, so the first pair is “Text Files”, “*.txt”. In this list you specify every kind of file that the user may choose to select from. It is wise to offer also “All Files”,”*.*”. After this list you should specify the Title String for the window. The next argument (the fourth) specifies a list of special properties for the window. There you see the empty list [ ]; it indicates that I don’t want any special property. Argument five is a string with the name of the folder that should be used when the window opens. I put an empty string in that place; it means that Windows will decide which folder to show. On my computer it is the folder “My Documents”. The last argument is _SelecFiles. Here getFileName/6 returns the list of selected files. The underscore in front means that I don’t want to use this variable. GetFileName/6 also returns the selected file name as a function and I grab that name in the variable SelectedFiles. Normally the user will end with OK; an exception is when she hits the Cancel button. This is taken care of in the second clause.

Conclusion

In this chapter you learned about simple ways of communication with the user. You changed the Task Menu and used pop-up windows. In the mean time I hope you became familiar with the Code Expert and grew accustomed to using it. There are a lot of other possibilities for communicating with the user, we will meet them in later chapters. In chapter 7 e.g. we will customize forms and other dialogs.

By now you are able to do the following without much trouble.

- Create a new project
- Create new items in the Task Menu
- Activate items in the Task Menu
- Create A Form or a dialog and put some controls on it
- Enter Code with the help of the Code Expert.

In the next chapter we will take a closer look at the several parts of the Integrated Development Environment IDE and explore a little more what you can do with them.

Chapter 4 A closer look at the IDE[4]

The Integrated Development Environment IDE is a very nice and helpful tool to create Prolog programs. In this chapter we will make a tour around the IDE to give an overview

4.1 The IDE in general

The IDE is used to create, develop and maintain your Visual Prolog projects. Briefly speaking you will use the IDE for the following tasks in a project life cycle:

- Creation
  The project is created in the IDE, at creation time you choose important properties of your project, such as whether the project is an executable or a DLL, whether it uses GUI or is text based, etc.
- Building
  The project is built, i.e. compiled and linked, by the IDE. Compiling means that the Prolog code is translated into machine code so the processor can digest it. Linking means that the various parts of the program are knotted together.
- Browsing
  The IDE and the compiler collect information about the project, which is utilized in various ways for quick localization of entities, etc.
- Development
  During the development and maintenance of the project, the IDE is used to add and remove source files and GUI entities to the project and to edit these.
- Debugging
  The IDE is also used for debugging the project. The debugger is used to follow program execution and exploit the program state when the program runs. The debugger helps to find and repair mistakes that prevent your program from proper execution.

In the sequel we will look more detailed at each of these things. Now and then, especially in the beginning of this chapter, I shall repeat things that have been said before to keep the text complete and readable. In this chapter I shall use a project to illustrate things and to give examples. You should create that project and keep it at hand - no better way to learn than by doing.

Creation
First we will create a project by selecting *Project/New...* in the menu. In response to this you are presented to a dialog window that contains various properties of the project. It should be familiar to you by now. For this example project I have chosen

- that my project should have the name “ch04p01",
- that the project name is also used as the name of the target that is produced,
- that in this case the target is an *exe* file so the target name will be ch04p01.*exe*,
- that the target should be a GUI program, i.e. a program with a graphical user interface.

The Base Directory is the "base" of all your projects; you should choose a place that is convenient for you.

[4] This chapter contains many parts from the tutorial Environment Overview by Thomas Linder Puls

The new project will be created in a Sub-Directory of the base directory. By default this directory has the same name as the project itself. As usual for the moment you do not need to consider the remaining tabs and options.

Building

Before we make any modifications, we will build (i.e. compile and link) the project. In the Build menu you find commands for building, compiling and executing the project. If you choose Execute the project is first built, so that it is an up to date version that you execute. Therefore I will choose Execute. If you have not registered Visual Prolog you will be presented to a special screen telling you so. I will suggest that you register, but you can also choose "Continue Evaluation". In the Messages Window the IDE writes which files are compiled, etc. The process is monitored in the status bar at the bottom. If the build process succeeds, which it should in this case, the created program is executed. As a result you will see a little *doing-nothing* GUI program. You might notice that the program looks a bit like the IDE itself. This is no coincidence since the IDE is actually a Visual Prolog program. Later in this tutorial we will also see what happens, if the compiler or the linker detects errors in the build process.

Browsing

Right now we will turn our eyes to the Project Tree in the Project Window, and explore that a bit. The Project Tree itself is presented in the way that you know from the Windows Explorer. This is a standard Windows tree control so you should already be familiar with its usage. The Project Tree can be browsed in the same way as you browse folders in the Explorer. Now let's take a closer look at the contents of the tree.

Right now the project tree will look like figure 4.1.

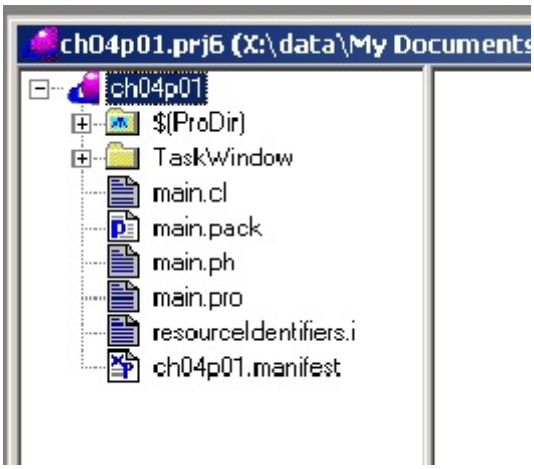


Figure 4.1 The Project Tree

Because the various folders constitute a tree structure, we will talk about the folders as nodes in a tree. The top node with the name "ch04p01" represents the project, and the project directory. It is the name you gave in the creation dialog. Right below that is the logical node (folder) $(ProDir). This directory contains (as we shall shortly see) libraries and library code from the Visual Prolog system. These are needed at compile and link time. In the Project Tree this folder is shown as a subdirectory

of your project, but in fact it may reside somewhere else on your hard disk. That's why we call this a logical node.

Below $(ProDir) comes another directory with the name TaskWindow, which is a real subdirectory of the project directory. This directory contains all the code needed to produce the Task Window of your program, its menu and toolbar(s) and the About dialog. The Task Window is the main window of your program, that is shown at startup. Because there are some traditional rules as how this window will look under MsWindows, the IDE knows what is expected and can create parts of the code without your specifications.

Finally, you see a number of files with the names "main..." and "resourceidentifiers.i". I will neglect the file "resourceidentifiers.i" for now. The other files are the files that will contain your program. Of course you will probably add more and more files, but these files are a start. You should realize that your program is spread over several files. This is a consequence of the fact that Visual Prolog is an object oriented language. Each file contains a part of your program. I shall elaborate on these files in chapter 8, for now a short introduction suffices. Your program is declared (defined) in file "main.cl". The extension ".cl" stands for "class". For now you can think of a class as a module. The Prolog code of your program resides in file "main.pro". There are two ways to go to the code in "main.pro". The first one is simply to double click on the name in the Project Tree. An editor is opened with the code in it. Try to find the piece of code that says:

```
clauses
   run():-
      TaskWindow = taskWindow::new(),
      TaskWindow:show().
```

Clauses are pieces of code that are executed. This clause says something like: "if you run this program, first make a new TaskWindow and then show it to the user". This is what happens when you execute the program ch04p01.
The other way to reach a particular piece of code is to single click on the name in the Project Tree. The right pane then shows the parts that are present in the file you clicked on, as shown in figure 4.2.

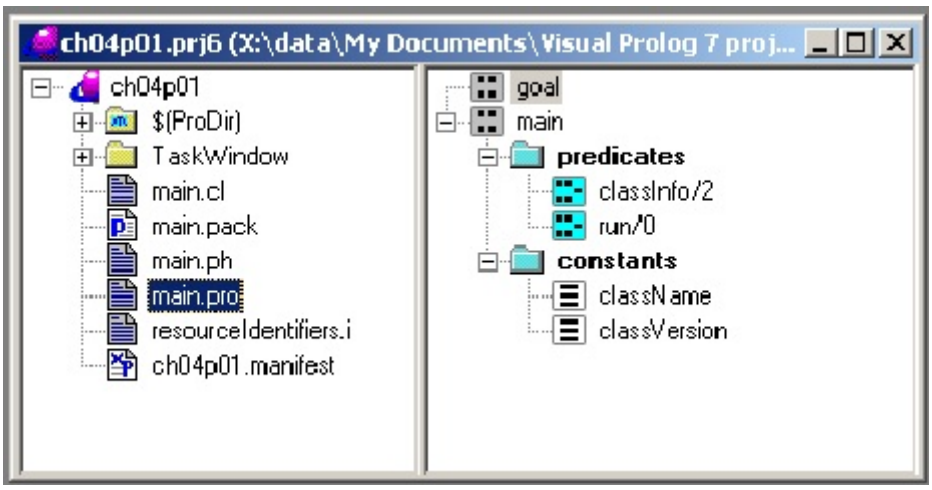


Figure 4.2 The Project Tree and the contents of "main.pro"

In the right pane you see words like “predicates”, “constants” and “goal”. To see everything as shown in figure 4.2, you may have to expand parts by clicking on + in front of a name. Don’t care too much about what words like “predicates”mean. The contents of the file “main.pro" are presented as a tree structure. This tree shows that the file main.pro contains a class called “main”, which contains two predicates called “classinfo/2" and “run/0" respectively. Think of these predicates as procedures. It also contains two constants, “className” and “classVersion” and there is also something called a “goal”. When you double click on an item, the IDE opens an editor and puts the cursor at the place where the code for that item is.

In the left pane you see that each filename has an extension. The extensions indicate the type of file. To understand these types, you must know about object oriented programming. I will explain it in chapter 8, but for now a short explanation should suffice. In object oriented programming (OOP for short) the central idea is that programs consist of parts that are called objects. Objects are created in a so-called class. A class is a general description of a certain type of objects. But a class is also a collection of activities that the objects can perform. When a program is run, the objects are created and perform activities. These activities produce the result that the programmer intended. As an example think of a database of students. This program could consist of a class “main” that is used to start the program by creating a taskwindow with a main menu. In this menu there will be options to create a student record or to search for a record. To create a student record, there could be a class called “student” that is used to enter and store the data of a student. When you click the menu option to create a new student record, the class “student”will be ordered to create a new object and then that object receives the data that are entered by the user and stores them somewhere in the database.

For such a program structure you need several types of files. You need

- classes - a collection of procedures and data
- interfaces - that tell how one class (or object) can order another object to perform an activity
- Prolog files - that contain the Prolog code of the program
- packages - groups of files that belong together. Think of a package as a folder under MsWindows.

In Visual Prolog the type of file is indicated in the extension. VIP uses the following conventions:

- *.cl file contains a class declaration
- *.i file contains an interface.
- *.pro file contains the class Prolog code also called the class implementation
- *.pack files contain #include directives that indicate the files that should be included in the package. The #include directives in *.pack files point to *.pro files and to *.ph files
- *.ph files are package header files that also contain #include directives
- The #include directives in *.ph files point to other *.ph files and *.i and *.cl files
- The #included files are only shown in the project tree when the project is compiled/built.

These are the extensions that you see in the left pane. I shall explain these concepts better in chapter 8 and later chapters, because you have to know about object orientation first to understand them.

4.2 TaskWindow in Project Tree

Now expand the node/folder with the name TaskWindow You will see a folder called “Toolbars” and several files with names like “AboutDialog” and “TaskWindow”. See figure 4.3. There is a pattern in

these files that you've heard about before. Let's take a look at the files with the name "Aboutdialog". You already know that a Dialog is a kind of form that is used to communicate with the user. The About Dialog is the little window that pops up when you click *Help/About* in the main menu of a program. The IDE keeps track of this window and its content in four files. The file "Aboutdialog.dlg" is the file in which you can edit the About Dialog.

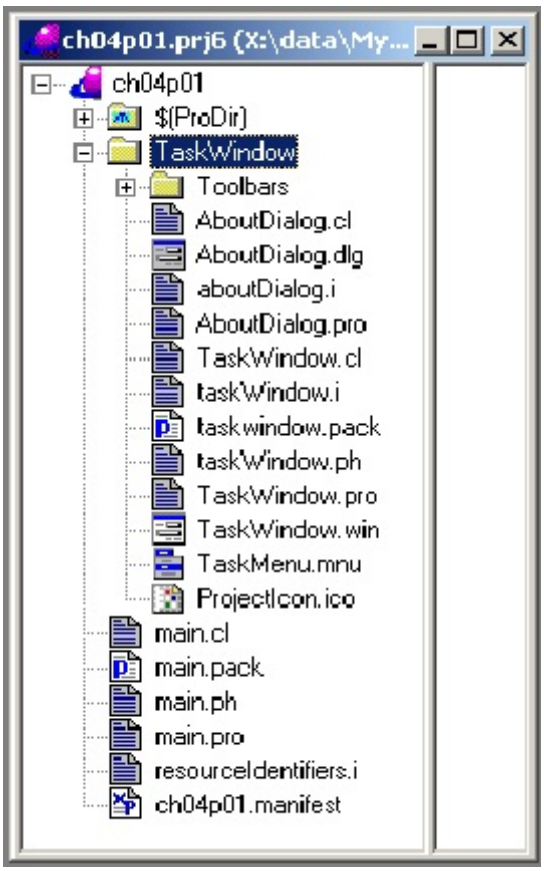


Figure 4.3 The contents of folder TaskWindow

Double click on "Aboutdialog.dlg" and a graphical editor opens that looks a lot like the forms editor that you already saw in chapter 2. You can change the contents that the IDE places by default in the dialog. The file "AboutDialog.cl" contains the declaration. Every item in your program must be declared; the About Dialog is a class (think of it as a module) and classes are declared in a ".cl" file. The file "AboutDialog.i" is an interface, more about that in chapter 8. Finally the code for the About Dialog is in file "AboutDialog.pro". This file consists of two parts. One part where you can freely enter, delete and change code, the second part is maintained by the IDE. It is generated every time you build the project after you changed "AboutDialog.dlg". This pattern is repeated throughout the IDE.

Now take a look at the files with the name "TaskWindow". By now you can guess that

- the file "TaskWindow.cl" contains the declaration,
- the file "TaskWindow.pro" contains the Prolog code,
- the file "TaskWindow.win" is the file to be edited. In this case when you double click on this file name, you get a dialog in which you can change the properties of the window. You do not really edit it as you do with e.g. the About Dialog. This is because the TaskWindow is built up

from separate items, like the taskmenu, the About Dialog, the Project Icon et cetera. "Taskwindow.win" is like a list of parts; here you specify what should be included in the Task Window.

The file "TaskMenu.mnu" contains the main menu of your program. You met it before. The Project Icon is a little picture that appears in the top left corner of your program.

Extensions in file names are important in the IDE. They indicate what type of file a file is and they help the IDE to open the correct editor when you double click a name. There are several different extensions

- *.dlg file contains a dialog
- *.frm file contains a form
- *.win file contains a window (task window or conventional PFC GUI window)
- *.mnu file contains a menu
- *.ico file contains an icon

If you continue to explore you might also meet:

- *.ctl files containing controls
- *.tb files containing toolbars
- *.cur files containing cursors
- *.bmp files containing bitmaps (pictures)
- *.lib files that are libraries

All these types are treated in different editors. The IDE recognizes the type of file and chooses the right editor. That's why it is important to choose the right extension - and never change an existing extension.

If you right click on a file name a context menu will appear, with commands that have relevance for that particular node. If you double click on a node the corresponding entity will be brought up in a suitable editor. All Prolog code is edited in a text editor, but windows resources like dialogs and menus are edited in graphical editors. As an example open the folder Toolbars and take a look at the first three items "ProjectToolbar.cl", "ProjectToolbar.pro" and "ProjectToolbar.tb". From the list above you know that a .tb file is a toolbar. When you right click on the name "ProjectToolbar.tb" a menu opens with four options

- Edit. This opens the graphical editor to edit the toolbar. It is equivalent to double click the file name
- Attributes. Opens a dialog to change attributes of the toolbar. E.g. you can change the position of it.
- Delete. Oops, it deletes the toolbar. Be careful.
- Properties. Displays several properties of the toolbar.

In this case the Code Expert is not available, as it is only used with the TaskWindow.

4.3 Creating a new item in the Project Tree[5]

The Project Tree shows the directories of your project and the items that are stored in them. To add a new item to the Project Tree, choose the option *File/New ...* from the task menu. But wait! Remember that every item is stored in a file, so in fact you are creating a new file. In an object-oriented

[5]The following text is partly from chapter 4 of Prolog for Tyros by Eduardo Costa

environment like Visual Prolog you will create many files. So a wise woman will see to it that the Project Tree will be a logical hierarchy of folders and files and that files that belong to a group will be placed in the same folder. Such a group is called a package, you can think of a package as a folder in MsWindows.

From time tot time you will need to create folders to store parts of your project. In this section we shall create a package in the project ch04p01 and put a form in it. The package will get the name *pack01* and the form will get the name *query*.

Step 1 Create the package
Add a new package to the project tree: ch04p01/pack01. Take the following steps.

- In the Project Tree click on the root directory ch04p01 to highlight it.
- In the Task Menu click on *File/New in existing package*; the Create Project Item dialog opens (figure 4.4)
- In the left pane click on “Package”
- In the field Name enter “pack01”. The field Parent Directory should be empty. The form should look like in figure 4.4.
- Click on the <Create>button to create the new package

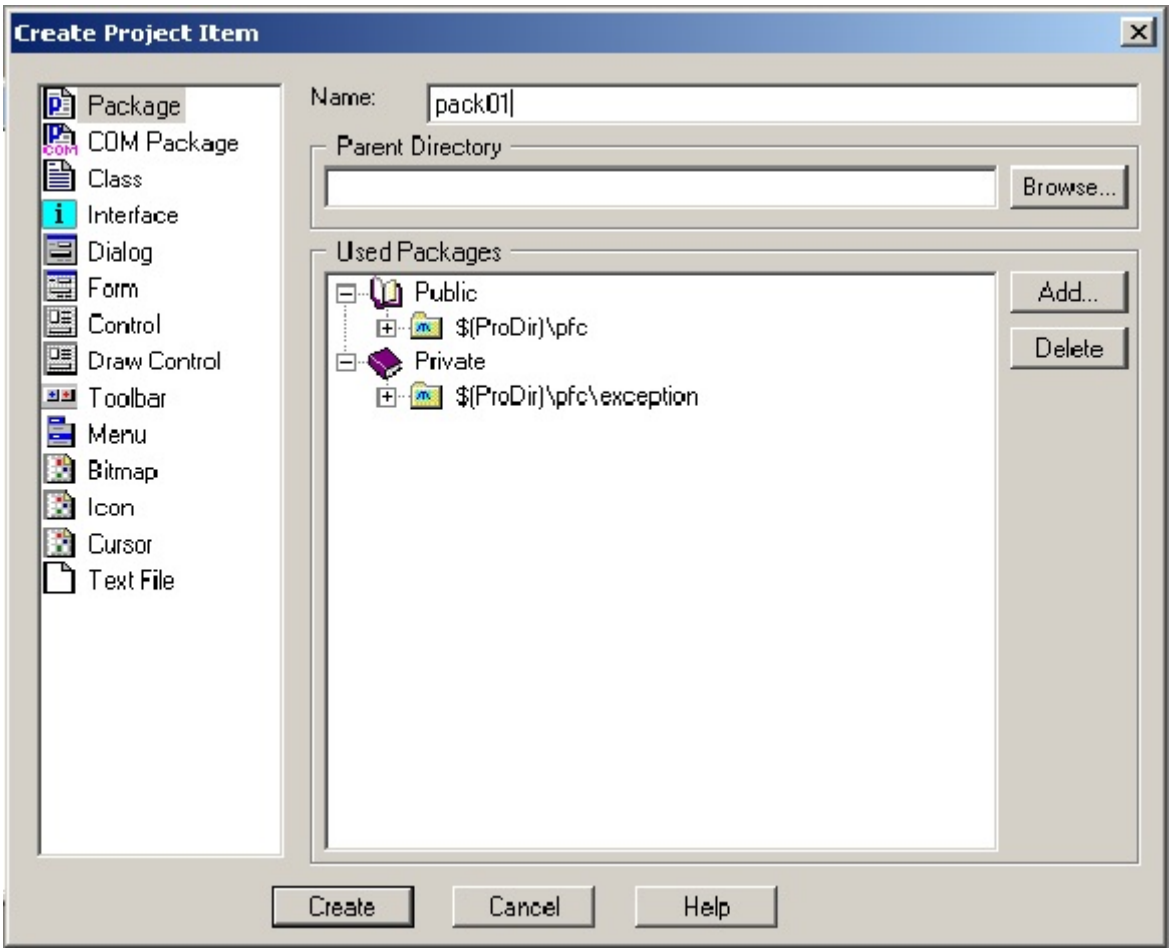


Figure 4.4 Create a new package

Take care where you place the package. If you want the package to be in the project root, (that is the main folder for your project), leave the parent directory empty, as in figure 4.4. One way to start with an empty parent directory is to select the project tree root before clicking *File/New in existing package*, as shown in figure 2.1. It is also possible to search for the right directory after you click File/New. In that case you click the Browse button at the right and the Create Project Item dialog shows you the directory structure on your computer. From here you may choose any directory that you want. But take care. Preferably you should choose a directory within the sub-directory of your project, unless you have good reasons not to do so.

Step 2 Create a form and store it in the package.
Now that you have created a package pack01, you will put a form with the name Query in it.

As you have done this before in section 2.1, the steps to do it should be familiar. For completeness sake I will give them again. Take the following steps.

- In the Project Tree click on the newly created package pack01 (not on pack01.pack!).
- In the Task Menu click on *File/New in existing package*. The Create Project Item dialog opens.
- In the left pane click on Form
- In the field Name type “query”
- At this point you must choose between the two radio buttons New Package and Existing Package. When you check New Package, the files for your form will be placed in a subdirectory in the directory pack01. When you check Existing Package, the files will be placed in the directory pack01 itself. I have chosen Existing Package.
- Click the <Create>button. The IDE shows a new form in the forms editor.

The IDE shows the forms editor (see figure 4.5). Don’t close it for the moment. We will take the opportunity to add an Edit Field and a button. Edit Fields allow the user of the program to enter text, a number or another input value or to change a value. A click on a button is an event, causing the program to execute some activity.

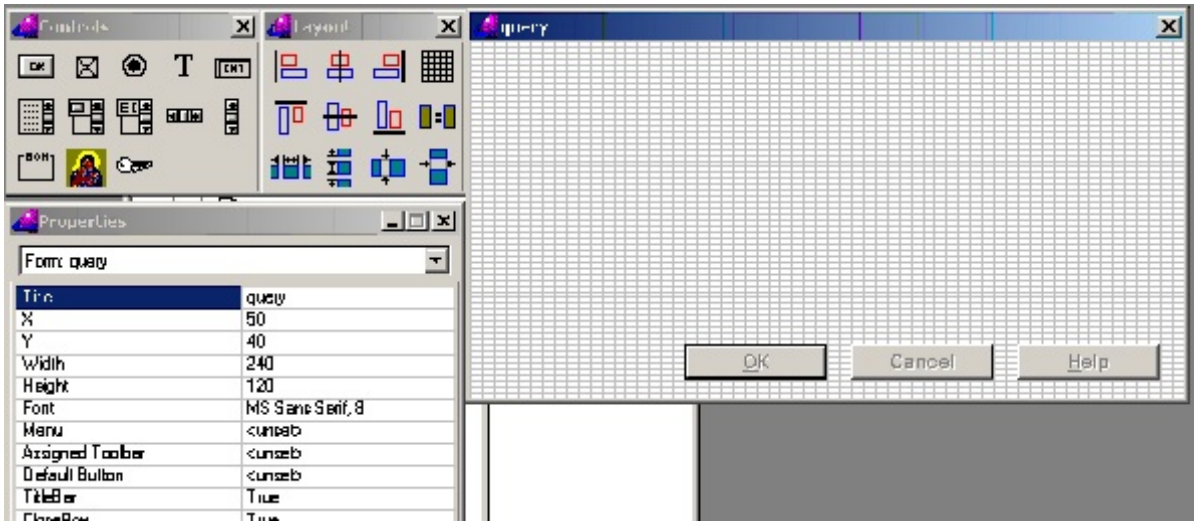


Figure 4.5 the forms editor with controls and layout toolbars and properties window

To add an Edit Field, do the following. In the Controls toolbar, click on the Edit Field Icon. It is the rightmost icon on the first row, with the word “Edit” in it. Now go to the form template and left click

at the place where you want the Edit Field. The IDE enters an Edit Field. You can resize it according to your wishes. To add a button, again go to the Controls toolbar and select (that is click on) the PushButton Icon. It is the leftmost icon on the first row, with the word OK on it. Then go to the form template and click on the place where you want the button to reside.

When you click on an item on the form template, the properties list shows the attributes of the item and their values. In the left column you find the name of the property, in the right column the value. E.g. click on the button that you added and the properties list shows every attribute of it. See figure 4.6. The third attribute (from the top of the list) is the attribute "Text", that is also called the Text Field. The default value for the Text attribute for a button is "Push Button". (In short you can say: the default text for a button is "Push Button"). You may want to change that text. Simply click on the text value field in the properties list and type the text you want. There is another field in the properties list that you should pay attention to. It is the first field, the Name Field. Here you find the name that will be used in the program to refer to this Push Button. By default the button gets the Name "pushButton_ctl" because it is a control with the name Push Button. But you will want to change that to make your program better readable. You could change the name into the same name as you entered in the "Text"-field. In that case, don't forget to add the extension "_ctl" to indicate that it is a control. You may want to look at the other options in this list. Later on, when the program is finished go back to the query.frm and change some other attributes. See e.g. what happens when you change the attribute Visible into "false.

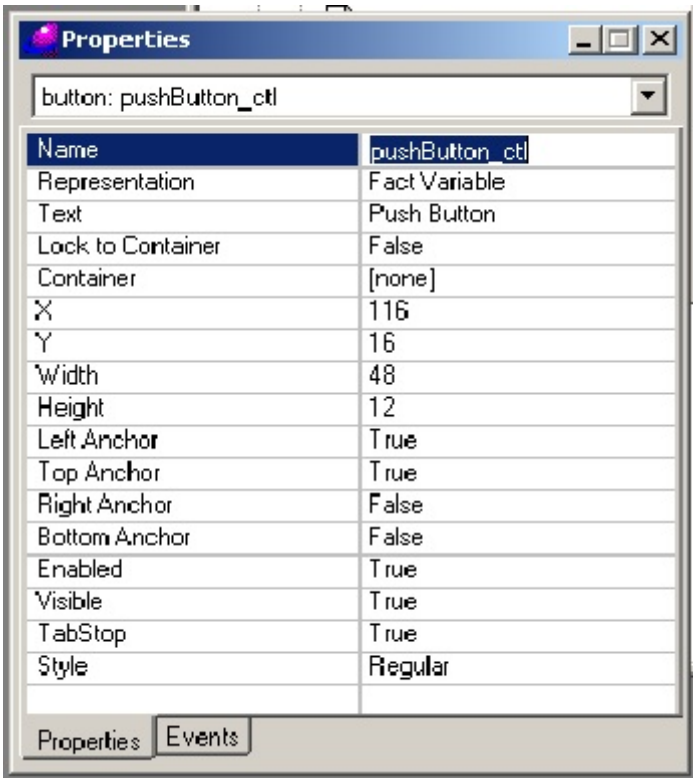


Figure 4.6 properties of PushButton_ctl

In order to put the form inside a package, be sure that the package folder is selected before going to *File/New*. For the name of the package and the form, choose names that have some meaning. For instance, if your package holds diagrams, name it *plots*; if it holds computer graphics, you could choose a name like Canvas Folder; if it contains queries, a good name is *queryForms*; if it contains

document filling forms, it is a *portfolio*; and so on. Never use names like Test or Folder01, unless you use the folder for tests or for number 01.

A form can be edited any time by going to the Forms Editor. You can reach the Forms Editor by right clicking on the file <formName>.frm in the project tree, and choosing the option *Edit* from the pop up menu. A shorter way is to double click on on the name of the form, e.g. “Query.frm”. Remember that the file with the extension “.frm” is the place to change the looks of the form. The file with extension “.pro” contains the code. You approach the files with code in two ways. The code for the Task Window is approached by way of the Dialog and Window Expert. The code for other items, like forms is approached by way of the events list in the properties window. More on that in section 4.5. Also you can reach the code by a double click on the file with extension “.pro” or by a right click on that name and choose “edit”. But I advise you to do that only when you know what you are doing.

4.4 The Code Expert and the Dialog and Window Expert

As its name reveals, the Code Expert is used to insert code, or snippets, into the program code of the Task Window. To reach the Code Expert, you must right click on TaskWindow.win. Then, choose the option *Code Expert* from the floating menu. The Code Expert opens the Dialog and Window Expert. The Dialog and Window Expert has a tree shape, that is shown in figure 4.7. This Expert shows the events that can take place and indicates between brackets in the title bar for which window the events are shown. In figure 4.6 the events of the Task Window are shown. It is up to you to choose an event for which you want to insert or change the code.

The Dialog and Window Expert is in fact a tree. A tree consists of branches and leaves, a leaf being the end of a branch. At a leaf you can manipulate the program code. To navigate the tree, and reach the point where you want to insert code, left-click on the appropriate branches of the tree until you arrive at the leaf you want. The Dialog an Windows Expert uses a simple procedure. You go to a leaf in the tree and click on it. Then there are three possibilities.

- There is no code inserted for this leaf. Then you click the <Add>button and the Expert inserts some standard code, also known as prototype code.
- There is already code inserted. Then you can delete it by clicking the <Delete>button.
- There is code inserted and you go there by double-clicking on the leaf.

So, if you want the Dialog and Window Expert to add standard code on a leaf, click on the leaf, and press the button <Add> that will appear at the bottom of the dialog. Then, double click on the leaf again, to reach the code. See figure 4.7.

If you want to add in CodeExpert (like you did in section 2.3)

```
clauses
    onFileNew(W, _MenuTag) :-
    S= query::new(W), S:show().
```

to TaskWindow.win/CodeExpert/Menu/TaskMenu/id_file/id_file_new, here are the steps that you must follow:

- In the Project tree
- - Open the folder *TaskWindow* of the project tree
- - Right click on *TaskWindow.win*, the floating menu opens
- - Choose the option Code Expert, the Dialog and Windows Expert opens
- In the Dialog and Windows Expert
- - Click on *Menu*

- - Click on *TaskMenu*
- - Click on *id_file*
- - Click on *id_file_new*
- - Press the button *Add* to generate prototype code
- - Finally, doubleclick on *id_file_new->onFileNew*. The editor opens the file Taskwindow.pro at the place where the code was inserted.
- In the file TaskWindow.pro - Add the requested code:

```
clauses
    onFileNew(Window, _MenuTag) :-
    NewWindow = query::new(Window), NewWindow:show().
```

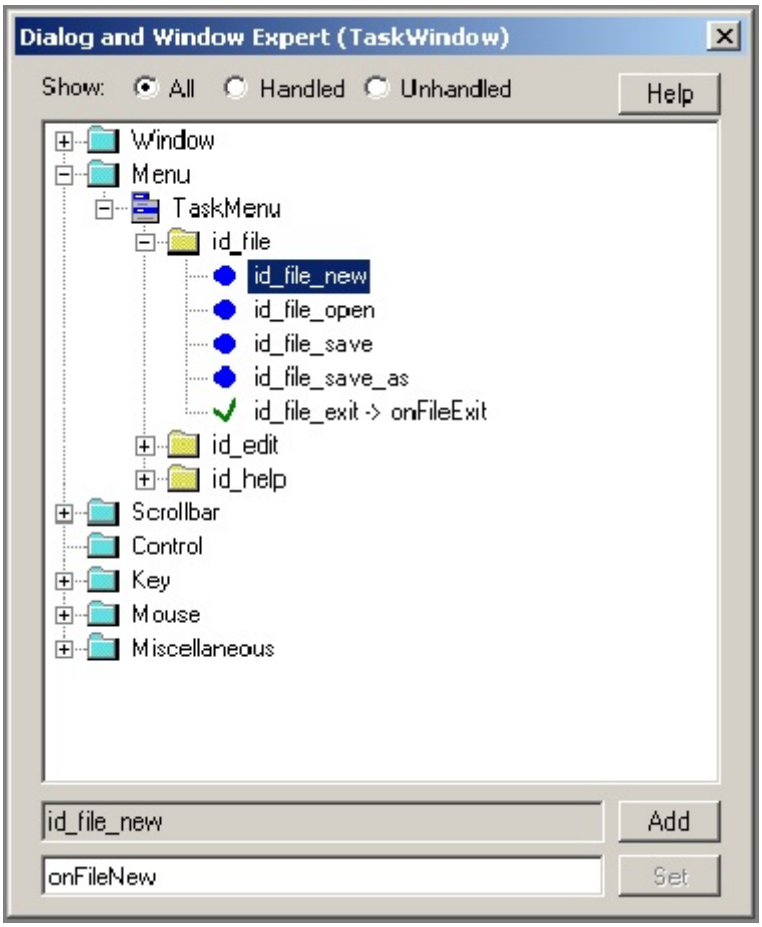


Figure 4.7 Dialog and Window Expert

When in figure 4.7 you take a closer look at the bar under the Title Bar, you see three options for what the Dialog and Window Expert should show. It is possible to show all the events (that is the default), but it is also possible to choose one of the options Handled and Unhandled. When you check Handled, only the events that have code attached are shown. When you check Unhandled, only the events that have not yet code attached are shown. When a window contains many events, it is sometimes handy to choose Handled or Unhandled. For now you can simply keep the default All.

In chapter 2 I told you to be careful. The Code Expert is a handy tool, but it should not make you careless. I repeat here a section from chapter 2, because I think it is important as an illustration of what I mean. When you <delete> code that was inserted by the Dialog and Window Expert, please do

good housekeeping. E.g. when you want to delete the code for “onFileNew” you open the TaskWindow.win in the Dialog and Window Expert (as you did in the previous section) , look for “id_file_new -> onFileNew”, highlight it and then click <Delete>. What happens is that the IDE deletes the line of code for the listener, but it does NOT delete the program code that you changed. The reason is that you probably have changed that code. Therefore it is difficult to decide what exactly should be deleted. A second reason is that by deleting, the IDE could destroy code that you want to keep to use it later or elsewhere. That’s why you must delete that code yourself by hand. Things may go wrong in this case. There are two possibilities

1. You delete the listener but not your own code
   When you build the project in this case you get a Warning that there are unused predicates in your program. The warning points at your own code, that is not used anymore as there is no listener to activate it. It is a warning because the remaining code doesn’t harm the program, it is simply not used.
2. You delete your own code but not the listener
   When you build the project in this case you get an Error that there is an unused identifier (say: variable) in your program. The listener points to an identifier variable (onFileNew) that it needs to react to clicking *File/New*. But you deleted that identifier so it is not found in the program code so the listener cannot direct the program control to it. That’s an error, you should repair it by entering your own code or by removing (via the IDE) the listener.

4.5 Reaching code via “Events”

You reach the code for the taskwindow via the Code Expert. Other code is reached by way of the properties window of the editor. Let’s play around a little bit. Please open the folder taskwindow and select (highlight) the file “AboutDialog.dlg”. Right click and choose “Edit”. The IDE opens the editor so you can edit the About Dialog. The properties window shows the properties of the About Dialog. Mark the two tabs at the bottom of the properties window. They are labeled “Properties” and “Events”. Click on the tab “Events”. The content of the windows changes. Instead of the properties, you now see a list of the events that can be handled by the dialog. It looks like in figure 4.8.

In this list you see the events that can happen when the About Dialog is open. Now locate the event with the name “CloseResponder”. This event is connected to the <Close>button that you find in the top right corner of a window.

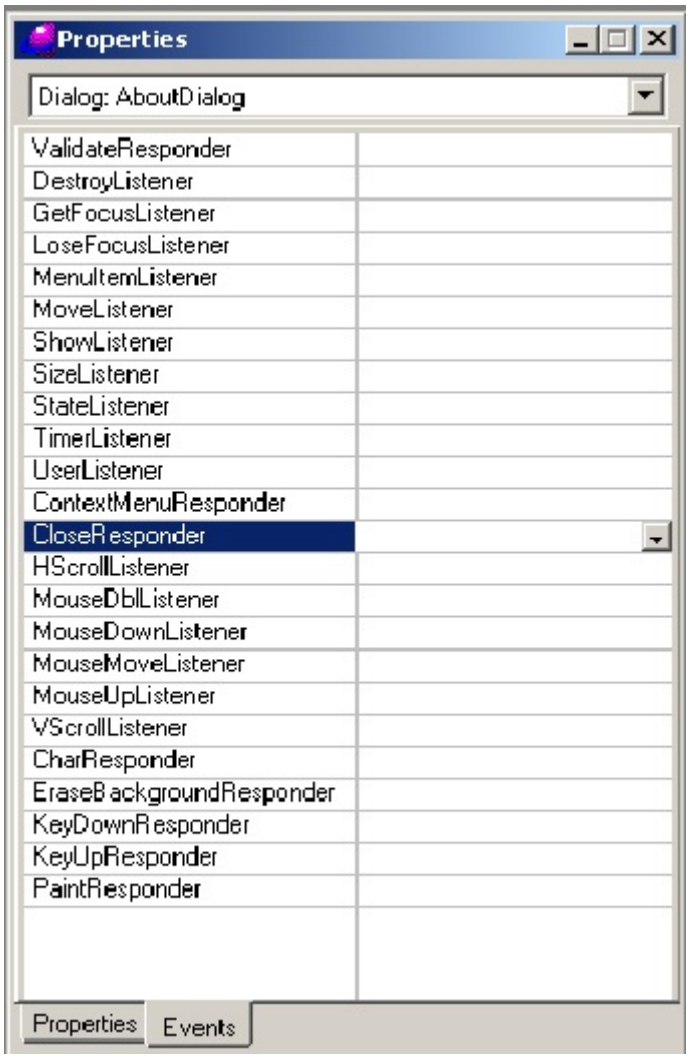


Figure 4.8 the events in the properties window of About Dialog

Whenever you click that button, the program recognizes that the event “CloseWindow” should happen and the CloseResponder responds by closing the window. But it is also possible to add more activities to this event. Let’s add a note that simply says what is happening. To add code, select “CloseResponder”. The value field gets an arrow. Click on it and the possible values for CloseResponder appear. In this case the only possibility is “onClose”. Choose it and double click on it. Now the IDE inserts a line of code that activates the listener and it inserts some standard code for the procedure “onClose”. It opens the Prolog Code editor and places the cursor at the inserted code. It looks like:

```
predicates
    onClose : frameDecoration::closeResponder.
clauses
    onClose(_Source) = frameDecoration::defaultCloseHandling.
```

Change the clauses part into:

```
clauses
    onClose(_Source) = frameDecoration::defaultCloseHandling :-
        vpiCommonDialogs::note("You hit the <close>button. \n Closing the About window now").
```

What is going to happen now is that when you click the close button, the program will (of course) close the About Dialog but first it shows you a note that tells you what you did and what is going to happen. By the way, do you notice the “\n” in the text? It creates a line break in the string.

Every other control can be given Prolog code in this way. Let’s add some code for when you click the <OK> button. Select the <OK>button in the form and bring the events to front in the properties window. It should look like figure 4.9.

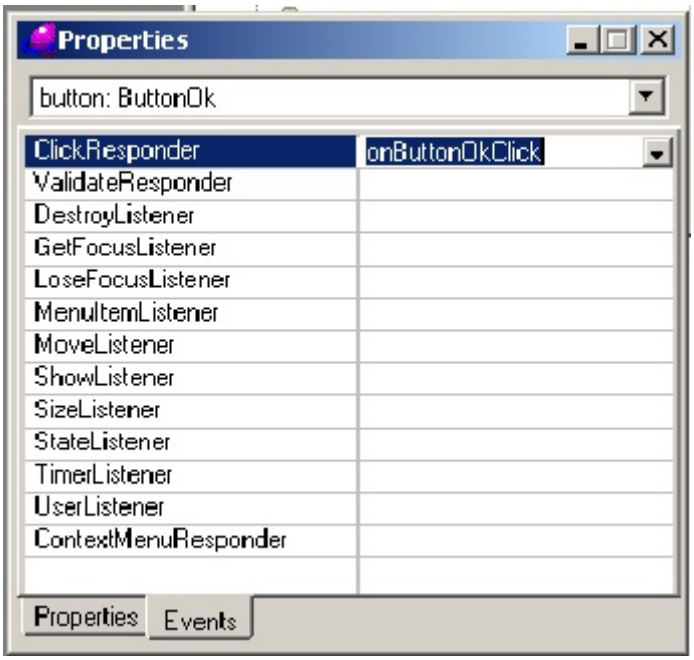


Figure 4.9 Events for the <OK> button with ClickResponder activated

Select ClickResponder. The ClickResponder responds to the event that you click the button. Choose the event “onButtonOkClick” (it is the only option :-)) and go to the code. The code that you see is something like:

```
predicates
   onButtonOkClick : button::clickResponder.
clauses
   onButtonOkClick(_Source) = button::defaultAction.
```

Change the clauses part into:

```
clauses
   onButtonOkClick(_Source) = button::defaultAction :-
      vpiCommonDialogs::note("OK, I'll close the AboutDialog").
```

Now close and save the code and other parts and build and execute the program. Click on “Help” and choose “About”. A more or less standard About Dialog opens. When you close it, you will see the note. In this case you see clearly that closing the window by way of the <close> button is a different event than closing the Dialog by clicking the <OK> button.

In practice you probably will never add program code to the About Dialog, but I thought it would be a funny example that shows that you can indeed change every item of a program. To add code to other parts the procedure is the same. Open the item (Dialog, Form, Toolbar, whatever) in an editor, select the item (push button, radio button, whatever) where you want to insert code and open the events list in the properties window. Select the event, double click on it and in the code editor that opens, change the inserted standard code.

Sounds easy, isn’t it? It is not too difficult. But what code to insert...., that’s another story. So in the next chapter let’s turn to programming in Prolog.

Chapter 5. Fundamental Prolog[6]

In this chapter you will learn about the very fundamental ideas of Prolog programming. I start with a more or less theoretical introduction and in the next section you will create “real” Prolog programs and run them.

Visual Prolog is object oriented, strictly typed and mode checked. You will of course have to master all this to write Visual Prolog programs. But here we will focus on the *core* of the code, i.e. the code when disregarding classes, types and modes and neglecting the graphical user interface.

You can learn Prolog with pen and paper, but it is better to start using a computer as soon as possible. That’s why we introduced the IDE in chapter 1. In this chapter 5 you will use the IDE to start a project called PIE. Project PIE is a program for using the language Prolog in a very classical way, more or less the way Prolog was used in the beginning (and probably is used still in some courses on artificial intelligence). The name PIE stands for Prolog Interpretation Engine. The program comes with Visual Prolog, you will find it in the examples. PIE is a "classical" Prolog interpreter, by using this you can learn and experiment with Prolog without at all being concerned with classes, types or GUI.

5.1 Horn Clause Logic

Visual Prolog and other Prolog dialects are based on Horn Clause logic. Horn Clause logic is a formal system for reasoning about things and the way these things are related to each other. An example: in natural language I can express a statement like:

John is the father of Bill.

Here I have two "things": John and Bill, and a "relation" between these, namely that one is the father of the other. In Horn Clause Logic I can formalize this statement in the following way:

```
father("Bill", "John").
```

In the formalization I put the name of the relation in front and put the two “things” between brackets, separated by a comma. The names of the two things are in computer slang known as string literals or strings. The apostrophes are used to delimit the strings. To add some more concepts: we say that “father” is a *predicate* (or a relation) taking two arguments, “Bill” and “John”. The predicate describes the relation between two *objects*, in this case Bill and John. In the interpretation that I give here the second argument “John” is the father of the first argument “Bill”.

Notice that ***I*** have chosen that the **second** object should be the father of the **first**. I might as well have chosen it the other way around: The order of the arguments is the choice of the designer of the formalization. However, once you have chosen, you must be consistent. So in **my** formalization the father must **always** be the second person. You can say that in a computer program you create a world and that in this world the second argument of the predicate “father” represents the father in the relation.

[6] This chapter is a remake of the tutorial Fundamental Prolog (Part 1) by Thomas Linder Puls (PDC)

I have chosen to represent the persons by their first names and to represent the names by strings. In a more complex world this would not be sufficient because many people have the same first name. But for now we will be content with this simple formalization.

With formalizations like the one above I can state any kind of family relation between any persons. Or more general, with a predicate and arguments I can specify any relation between objects. Take a look at some examples.

lives_in(“Thomas”, “Groningen”).
is_married_to(“John”,”Claire”).

are examples of relations between two objects. Any number of arguments is possible. E.g. one argument:

is_dumb(“John”).

But also three objects are possible:

has_degree(“Jeannet”, “shorthand”, “Spanish”).

The number of arguments is only limited by your imagination.

Specifications like these are called facts. A fact consists of a predicate, a number of arguments between brackets and separated by comma’s and it is ended by a full stop, a period. A fact describes a static relation between objects.

Facts are useful to know, but they get boring as you find out when you watch quizzes on TV. To make facts more interesting, you need more general relations between objects or groups of objects. General relations between (groups of) objects are called ***rules***. As an example we can extend the father-example with a rule about grandfathers.

> Person3 is the grandfather of Person1, if Person2 is the father of Person1 and Person 3 is the father of Person 2

where Person1, Person2 and Person3 indicate individual persons (objects). In Horn Clause Logic I can formalize this rule as:

```
grandFather(Person1, Person3) :-
        father(Person1, Person2), father(Person2, Person3).
```

In this rule we use, next to the predicates two so called logical operators: the words “if” and the word “and”. In this formalization the word “if” is replaced by the symbol “:-“ and the word “and” is replaced by a comma “,”. These are the symbols that are used in Prolog to represent logical operators. And clearly in “grandFather(Person1, Person3)” I want to indicate that Person3 is the grandfather of Person1. A formalization like this is called a *rule*. Take a closer look at grammar of the rule. The first part “grandFather(Person1, Person3)” is called the conclusion. The conclusion is followed by the if-sign “:-“. After the if-sign there are one or more fact-predicates, separated by commas and ended by a period. In a rule these fact-predicates are called the premisses of the rule. Notice also that “grandFather” starts with a lower case letter. This is Prolog syntax, a predicate must start with a lower case letter. Arguments like “Person1" start with an upper case letter. In Prolog this indicates that it is a variable. It means that in this place you can use any object that is of type person. Of course the rule will not hold for any combination of persons, but that is another matter. Notice that in this rule the premisses are facts. It is also possible to use conclusions of other rules as premisses.

If you know about logic, then you will recognize the pattern that is known as deduction. The difference is that in logic we are used to write something like:

IF p AND q THEN r

while in prolog we would write:

r :- p, q.

When the premisses are true, then the conclusion is also true. This is more or less the case in Prolog. The general form of a rule is:

conclusion :- premiss1, premiss2, ...,premissN.

Let's return to the grandfather rule. As it is formulated, it is difficult to understand. E.g. from the way it is written, you cannot see directly which Person is supposed to be the grandfather. It is better to use understandable names for the variables instead of Person1, Person2 and Person3. In Horn Clause Logic you can do that just as you like it. You are free to choose almost any variable name as you like, so you can e.g. rephrase the rule to:

```
grandFather(GrandChild, GrandFather) :-
        father(GrandChild, Father), father(Father, GrandFather).
```

Realize that this is only convenient for human readers of your code. Computers don't realize that a variable "Father" should have a child. Your computer would be just as happy with a rule:

```
g(X,Y) :- f(X,Z), f(Z,Y).
```

In the rule we use variables instead of object names. A rule is a general relation that holds between any objects. As long as there is an object Y that relates as a father to object X and at the same time an object Z relates to Y as a father, then in the world of your program object Z is the grandfather of object X. Period.

Summarizing: when reading rules you should interpret the symbol ":-" as (a logical) ***if*** and the comma that separates the predicates as (a logical) ***and***. In the variable names GrandChild, Father and Grandfather the first letters are upper case. In Prolog a variable name always starts with upper case. In this rule "grandFather(GrandChild, GrandFather)" is the conclusion and "father(GrandChild, Father), father(Father, GrandFather)" are the premisses. The rule is ended with a period.

By choosing ***variable*** names that help understanding better than X, Y and Z, I improved the readability of the rule for humans. Imagine that you had to decide from "grandfather(X,Y)" which one of X and Y is the grandfather. It could easily lead to confusion. Readability was also improved by writing the rule on two lines. One line for the conclusion and one line for the premisses. Or when the premisses are long, you can write every premiss on a new line.

By adding this rule I have introduced a predicate for the grandfather relation. That is efficient. Instead of writing a fact for every grandfather and grandchild in my world, I only specify one rule and use implicitly the facts about fathers and children. Again I have chosen that the grandfather should be the second argument. It is wise to be consistent like this, i.e. that the arguments of the different predicates follow some common principle.

In Prolog facts and rules are called *clauses*. With facts and rules we are ready to formulate theories. In general a ***theory*** is a collection of facts and rules, in Prolog we would say a theory is a set of clauses. Let me state a little theory that consists of three facts and one rule:

```
father("Bill", "John").
```

```
father("Pam", "Bill").
father("Sue ", "Jim ").
grandFather(GrandChild, GrandFather) :-
   father(GrandChild, Father),
   father(Father, GrandFather).
```

With this theory you could answer questions like:

Is Jim the father of Sue?

This question can be answered by looking at the facts. The third fact acknowledges that Jim is the father of Sue, so the answer is “yes”.

Is John the father of Sue?

There is no fact stating that John is the father of Sue, so the answer is “no”. Notice however that there is a fact that says that Jim is the father of Sue. This implies that John cannot be the father of Sue. Smart humans will notice this, but this is not the way Prolog reasons. In Prolog the answer is “no” because there is no fact stating that it should be “yes”.

Who is the father of Pam?

This question can be answered by looking at the second fact. It says that Bill is the father of Pam.

Is John the father of Ann?

As there are no facts about Ann, we cannot answer this question. In Prolog we then assume that the answer is “no”, as we cannot state that the answer is “yes”.

Is John the grandfather of Pam?

To answer this question we have to use the rule and the facts. The rule tells us that we need to find two facts. The first should say that Someone is the father of Pam and the other fact should state that John is the father of the same Someone. Looking at the facts you see that the facts are precisely saying this when we take Bill for Someone. So yes indeed, John is the grandfather of Pam.

In Prolog you can asks these questions to the inference engine that is built in Prolog. In Prolog such questions are called goals (maybe because the inference engine pursues them). Goals can be formalized like this (respectively):

The question “Is Jim the father of Sue” becomes:

```
?-father("Sue ", "Jim ").
```

The question “Is John the father of Sue?” becomes:

```
?- father("Sue", "John").
```

The question “What is the name of the father of Pam?” becomes:

```
?- father("Pam", X).
```

The question “Is John the father of Ann?” becomes:

```
?- father("Ann", "John").
```

The question “Is John the grandfather of Pam?” becomes:

```
?- grandFather("Pam", "John").
```

Such questions are called ***goal clauses*** or simply ***goals***. Together *facts*, *rules* and *goals* are called Horn clauses, hence the name Horn Clause Logic.

Some goals like the first, second and last are answered with a simple *yes* or *no*, because you are asking if a fact is true. For other goals like the third we seek a ***solution***, like X = "Bill". Some goals may even have many solutions. For example:

?- father(X, Y).

has three solutions (there are three facts that give a solution):

X = "Bill", Y = "John".
X = "Pam", Y = "Bill".
X= "Sue", Y= "Jim".

A Prolog program consists of a theory and a goal. When the program starts it tries to find a solution to the goal in the theory. This search for solutions is performed by a part of Prolog that is called the inference engine. In the search for solutions it uses the rules and the facts from the theory more or less in the same way as a human being would do. Of course the search for solutions by an inference engine is very formalized, but the analogy is useful when you want to understand how Prolog and the built in inference engine work. In the next section we elaborate on that.

5.2 PIE: Prolog Inference Engine

Now we will try the example created above in PIE, the Prolog Inference Engine. PIE comes as an example with Visual Prolog. Before we can use the program, you should install and build the PIE example. When you haven't yet installed the examples, please do so now.

- Select "Install Examples" in the Windows start menu (*Start/Visual Prolog/Install Examples*).
- Start Visual Prolog
- You find project PIE in the folder Examples that is installed. Open the project PIE in the IDE and **execute** the program. You can execute PIE as a normal project, as described in part 1. You don't have to change anything, so in fact you only use the IDE to execute the program.

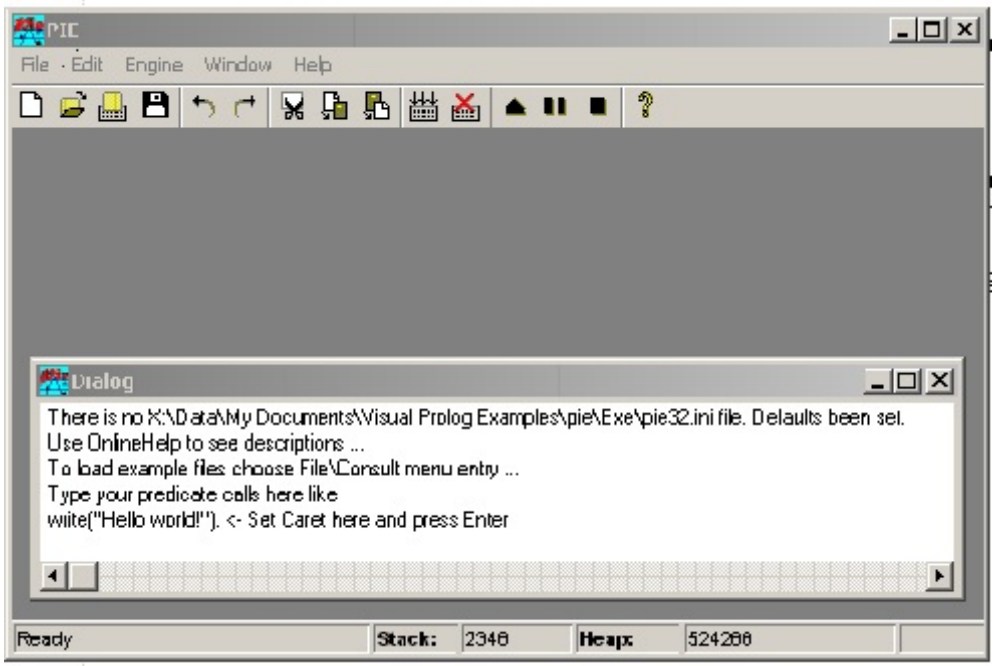


Figure 5.1 PIE and the dialog window

If everything goes well, PIE will start by showing the PIE Task Window and in it a dialog window. It looks like in figure 5.1.

The Dialog Window shows that a file “pie32.ini” was created with some default values. It tells you to use the Online Help for descriptions and a few other things. Here I want to execute the program that we developed in the previous section. To create that program, select *File/ New* from the Menu bar. PIE opens a second window, the Program Window that we will use to enter the program. There are now two windows open, the Program Window and the Dialog Window. The Program Window is used for the program, its title bar shows the name “File0.pro” that PIE gives by default. In the Program Window you place the facts and the rules of your theory. Sometimes we call these also the program, but remember that a program is not complete without a goal. The Dialog Window is used for the dialog with the inference engine. Here you enter the goals for the program, that is the questions that you want to be answered. Also the messages by PIE appear in this window. It is important to take care of which window has control (is active). To give control to a window click its Title Bar.

Please enter in the Program Window the father and grandFather clauses (that is the facts and rules) from the previous section. The result should look like figure 5.2.

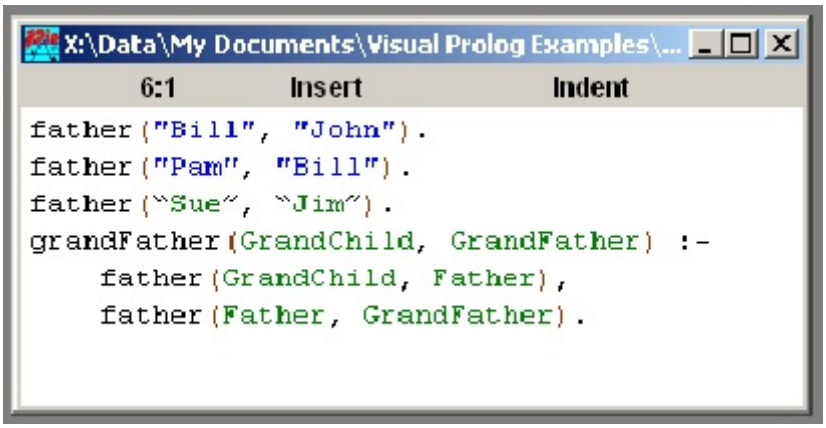


Figure 5.2 The Program Window

The clauses you entered are now in the Program Window. If you want the inference engine to use these clauses and rule, then you must load them into the inference engine. While the Program Window is active choose *Engine/Reconsult*. This will load whatever is in the editor into the inference engine, so the engine can work with it. In the ***Dialog*** window you should receive a message like this:

Reconsulted from: ....\pie\Exe\FILE0.PRO

Reconsult loads whatever is in the editor, ***without*** saving the contents to a file. If you want to save the contents use *File/Save*. A saved file can be brought back to the Editor window by *File/Consult*. This will load the contents of the file regardless of whether the file is opened for editing or not.

To make the inference engine search for a solution, you must give it a goal it can pursue. This is done in the Dialog Window. The procedure is always as follows.

- Create or load a file in the Program Window with *File/New* or *File/Consult*
- Save that file when necessary with *File/Save*
- Reconsult it by the engine with *Engine/Reconsult*
- Enter a goal in the Dialog window and hit <Enter>

Once you have "consulted" the theory, you can use it to answer goals.

On a blank line in the ***Dialog*** window type a goal (without the **?-** in front). Let's ask the question who is grandfather of who. In Prolog this becomes a goalwith two variables. We want the inference engine to find values (names) for the variables. So in the Dialog Window type (on a blank line):

grandfather(X,Y).

Take care to write a lower case "g" in grandFather and to write upper case "X" and "Y" and don't forget the full stop at the end. It should look like figure 5.3

Figure 5.3

Now place the cursor at the end of the line and press the **Enter** key on your keyboard. PIE will now consider the text from the beginning of the line to the cursor as a goal to execute. The inference engine does its work and you should see a result as in figure 5.4

Figure 5.4

If you get instead a message like:

grandFather(X,Y).
Unknown clause found grandfather(X$0,Y$1)
Execution terminated
No solutions

then you probably forgot to Reconsult. Take care to write the goal exactly as in the figure. If e.g. instance you write "GrandFather" instead of "grandFather" you will get an error message. A predicate should start with a lower case letter. And don't forget the full stop at the end. <Enter> alone isn't enough!

It is a good exercise to also try the other goals from the previous section. Take some time to practice, it is worthwhile the trouble. However you need to know a few things about specifying a goal. Here are some rules.

- You can use any predicate in the program to formulate a goal. In this case you can use "father" or "grandFather".
- With a predicate you must give the appropriate arguments. That means the right number and the right type. In this case the type is always string, so it is hard to make a mistake here.
- As an argument you can enter the name of a specific object or you can use a variable. Names of specific objects here are "Sue", "Jim", "Bill", et cetera but in fact any string is acceptable.
  Examples are:
  father("Sue","Jim").
  father("Sammy","Simson").
  grandfather("Sue","Jim").
  When you enter a goal like this, the inference engine will answer Yes or No depending on the rules and facts in your program.
- The second option is to use variables as arguments. Examples are
  father(X,Y).
  father(Child,Father).
  grandfather(GrandChild, GrandFather).
  Notice that a variable always starts with an upper case letter. When you enter a goal with

variables the inference engine will try to find values for the variables so that the goal matches a fact. When it succeeds, it will report the values for the variables. The output will be better readable when you use variable names that make sense.

- You can use object names and variables in the same goal. E.g. father(“Sue”, Father). In this case the inference engine tries again to find a value for the variable that will make the goal match with one of the facts.
- A goal may consist of more than one predicate. In that case you have to use the logical operators “,” for **and** or “;” for **or**. An example is the goal father(“Sue”,”Jim”),father(“Bill”,”John”).

Now please practice a little until you get the feeling for the way the inference engine produces answers.

5.3 Extending the family theory

Adding facts and rules seem a nice way to increase the intellectual power of your program. But take care! In prolog you are playing with logic and an inference engine. The combination of these two sometimes leads to unexpected results. In this section we will look at some common problems that you should avoid.

It is straight forward to extend the family theory above with predicates like “mother” and “grandMother”. You should try that yourself. You should also add more persons. I suggest that you use persons from your own family, because that makes it lot easier to validate, whether some person is indeed the grandmother of some other person, etc.

Given mother and father we can also define a parent predicate. You are a parent if you are a mother; you are also a parent if you are a father. Therefore we can define parent using two clauses like this:

```
parent(Person, Parent) :- mother(Person, Parent).
parent(Person, Parent) :- father(Person, Parent).
```

In these rules I have chosen that the second argument is the mother/father/parent of the first argument. Now the first rule reads

Parent is the parent of Person, if Parent is the mother of Person

The second rule reads that Parent is the parent of Person if Parent is the father of Person.
Together these rules state that there are two possibilities, either you are the mother, or you are the father, in both cases you are a parent. It is as if the facts offer you two chances. If you are not the mother, then try and look if you are the father.

You can also define the parent relation using the semicolon ";" which means (logically) “or”. Then the rule will look like this:

```
parent(Person, Parent) :-
   mother(Person, Parent); father(Person, Parent).
```

This rule reads:

Parent is the parent of Person, if Parent is the mother of Person ***or*** Parent is the father of Person

I will however advise you to use semicolons as little as possible (or better: not at all). There are several reasons for this:

- The typographical difference between "," and ";" is very small, but the semantic difference is rather big. ";" is often a source of confusion, since it is easily misinterpreted as ",", especially when it is on the end of a long line.
- Visual Prolog only allows you to use semicolon on the outermost level (PIE will allow arbitrarily deep nesting), so it is not very useful.
- When you combine "," and ";" in one rule, it soon becomes confusing which premiss has priority in evaluating the rule. Of course this is well defined, but with normal human cognition it leads easily to mistakes.
- There is a simple better way to represent a logical "or". Take the rule

```
parent(Person, Parent) :-
   mother(Person, Parent);      % Do you see the semicolon?
   father(Person, Parent).
```

  You can rewrite this rule as two separate rules:

```
parent(Person, Parent) :- mother(Person, Parent).   %Do you see the full stop?
parent(Person, Parent) :- father(Person, Parent).
```

  These two rules should be read as before: to see if Parent is parent of Person we first see if Parent is the mother. If that is not the case, then we see if Parent is the father. So in fact the two rules are combined with a logical "or".

Try creating a sibling predicate! And ask the inference engine to find siblings. Did that give problems?

You may find that siblings are found **twice**. At least if you say: two persons are siblings if they have the same mother, two persons are also siblings if they have the same father. I.e. if you have rules like this:

```
sibling(Person, Sibling) :- mother(Person, Mother), mother(Sibling, Mother).
sibling(Person, Sibling) :- father(Person, Father), father(Sibling, Father).
```

The first rule reads:

> Sibling is the sibling of Person if Mother is the mother of Person ***and*** Mother is the mother of Sibling

The second rule reads:

> Sibling is the sibling of Person if Father is the father of Person ***and*** Father is the father of Sibling

The reason that you receive siblings twice is that most siblings both have the same father and mother, and therefore they fulfill both requirements above. And therefore the inference engine finds a solution twice. We shall not deal with this problem now; currently we will just accept that some rules give too many results.

A fullBloodedSibling predicate does not have the same problem, because it will require that both the father and the mother are the same:

```
fullBloodedSibling(Person, Sibling) :-
   mother(Person, Mother),
   mother(Sibling, Mother),
```

father(Person, Father),
father(Sibling, Father).

You see that the four premisses that were spread over two rules are now in one rule, connected with "and".

5.4 Prolog is a programming language

From the description so far you might think that Prolog is an expert system, rather than a programming language. You ask a question and Prolog gives the answer. And indeed Visual Prolog can be used as an expert system, but it is designed to be a programming language.
To make it a usable programming language, We need two important ingredients to turn Horn Clause Logic into a programming language:

- Rigid search order and/or program control
  When a program is executed, it should behave predictable. That is, it must not react in a random way. Imagine a computer program that behaves on Monday differently than it behaves on Tuesday. For that we need among others a strict way in which the program handles the goal and the rules. E.g. it should search the rules in a rigid order.
- Side effects
  Horn clauses are great for answering questions and detecting whether a proposition is true. But a computer program needs to communicate with the user and its environment. For this we need something that is called "side effects" in Prolog. An example of a side effect is writing to the screen as we did in the previous chapter.

We will deal with these subjects in the next two sections.

5.5 Program Control

Prolog is a programming language, so Prolog knows a strict search and program control. In this section we will look into detail into the way Prolog tries to solve the goal and we will look at two important notions: fail and backtracking.

5.5.1 Finding a match

I said before that the inference engine tries to find a match for the goal that you enter. When you enter a goal that is simply a fact, like "father("Sue","Jim")." then the search is straightforward. The engine looks in the program at the facts with the predicate "father" and decides for Yes or No. But what when you enter as a goal not a fact, but the conclusions of a rule, like "grandFather(X,Z)"? In that case there is no fact to match with. With a goal like this one, the inference engine uses the rule to replace the compound goal "grandFather" with the two sub-goals in the premisses "father(X,Y)" and "father(Y,Z)". Now it has two goals to solve, but these sub-goals are not compound and can be found in the program facts and so the engine can solve the goal.

There are several possibilities and you should know how the engine handles these.

- When you use object names in the goal, the engine transfers the names from the compound goal to the sub-goals. E.g. when the goal is "grandFather(Sue", Jim)." then the sub-goals become "father("Sue",Y), father(Y,"Jim")."

- When you use variables in the goal, these are also transferred from the compound goal to the sub-goals. E.g. “GrandFather(“Sue”, Grandfather)” becomes “father(“Sue”,Y),father(Y,Grandfather).”.

In the next section we see how the inference engine uses matching to solve the goal. The strategy is to breakdown the goal in simple sub-goals that we hope to find in the facts in our theory.

5.5.2 Solving the goal

When you as a human being try to find a solution to a goal like:

?- father(X, Y).

you can do it in many ways. For example, you might only consider the second fact in the theory and then you only have the solution, X=”Pam”,Y=”Bill”. Or you might search the rules in a different order, e.g. from bottom to top or from top to bottom.

But Prolog does not use a "random" search strategy, instead it always uses the same strategy. It starts with the question that is asked and calls it the *current goal*. The system maintains this ***current goal*** throughout the search, and tries to solve it **left to right**.

Suppose the current goal is:

?- grandFather(X, Y), mother(Y, Z).

It says that you want to find the grandfather of X, save the name in Y and then want to find the mother of Y. In this case the system will always try to solve the sub-goal grandFather(X, Y) before it solves mother(Y, Z). If the first (i.e. left-most) sub-goal cannot be solved then there is no solution to the overall problem and the second sub-goal is not tried at all.

When solving a particular sub-goal, the facts and rules are always tried from **top to bottom**. In short, the inference engine in Prolog works from left to right and from top to bottom, just like you do when you read a book or a cartoon in English.

When a sub-goal is solved by using a rule, the right hand side replaces the sub-goal in the current goal.

Let us look at the following goal.

?- grandFather(X, Y), mother(Y, Z).

The engine starts by solving the leftmost part, that is “grandFather(X, Y)”. To solve it, it looks in the facts and rules of the theory and it finds the rule

grandFather(GrandChild, GrandFather) :- father(GrandChild, Father), father(Father, GrandFather).

Because the conclusion of this rule matches the first sub-goal, we use the rule to solve it. Thereto we substitute the premisses of the rule in the first sub-goal, then the resulting current goal will be:

?- father(X, Father), father(Father, Y), mother(Y, Z).

This may seem stupid as we now have three sub-goals instead of two! But the idea behind it is that premisses are less complex (compound) than conclusions and therefore easier to check for truth. But don’t bother about that too much now. Notice also that some variables in the substituted premisses have been replaced by variables from the sub-goal as was explained in the previous section.

With the new formulation of the goal we start looking for facts and rules that will solve the goal. When a sub-goal is checked against a fact and found to be true, it can be removed from the goal.

When all the sub-goals are found to be true, the goal is true (remember the sub-goals are separated by comma's, that are logical *and*'s) and Prolog will show a message.

Given this evaluation strategy you can interpret clauses much more ***procedural***. Consider this rule:
grandFather(Person, GrandFather) :- father(Person, Father), father(Father, GrandFather).

Given the strict evaluation we can read this rule like this:
To solve grandFather(Person, GrandFather) **first** solve father(Person, Father) **and then** solve father(Father, GrandFather).

Or even like this:

> When grandFather(Person, GrandFather) is ***called***, first ***call*** father(Person, Father) and then ***call*** father(Father, GrandFather).

With this ***procedural*** reading you can see that predicates correspond to procedures or subroutines or functions in other languages. There is a distinction between procedures/subroutines and functions. A function returns only one value and a procedure returns one value for every output variable. But in Prolog a predicate can return several solutions to a single call (compare it with a goal that gives more than one solution) or even ***fail***. To fail means that is doesn't give a solution at all. It is one of the funny sides of Prolog that failing can be very useful. These things will be discussed in details in the next sections.

5.5.3 Failing

A predicate call or invocation might not have any solution in the program, for example calling

```
parent("Hans", X)
```

has no solution when there are no parent facts or rules that apply to "Hans". We say that in this case the call to the predicate ***fails***. The goal, as it is a kind of predicate, can also fail. If the goal fails then there is simply no solution to the goal in the theory.

There are two ways for a predicate to fail. The first way is when there is no matching fact for itself or one of its premisses. The second way is when the word "fail" is encountered in the premisses of a predicate. At that point the predicate fails. This may seem stupid, but in combination with the so-called backtracking, it will turn out to be a very powerful technique.

The next section will explain how failing is treated in the general case, i.e. when it is not the goal that fails.

5.5.4 Backtracking

By now you know that the inference engine tries to solve the goal by matching facts. The engine searches the rules and the facts from left to right and from top to bottom. In its relentless search for solutions the inference engine uses something called backtracking. You should understand what it means and how it is applied. Backtracking is the subject of this section.

Suppose you have written a program with the following facts and rule.

```
father("Bill", "John").
father("Pam", "Bill").
```

```
father(“Sue”, “Jim”).
grandFather(GrandChild, GrandFather) :-
   father(GrandChild, Father),
   father(Father, GrandFather).
```

Rather familiar, isn’t it? Suppose now that you ask the goal:

```
?-father(“Sue”,”Jim”).
```

You hit <Enter> , the inference engine takes the goal and sees that the predicate “father” in the goal matches with three facts in the program. So it has three chances to find a match for the goal. Because it works from top to bottom, it is going to start with the first fact “father("Bill", "John").” to see if it matches the goal. But before it does so, the engine puts a pointer at the second fact “father("Pam","Bill").”. This pointer is called a “backtrack point”. It is the point where the engine should return when it has treated the first fact. After creating the backtrack point the engine takes the first fact “father("Bill", "John").” and sees if it matches the goal. It clearly doesn’t as “Bill” is different from “Sue” (and “John” is different from “Jim”, but that is not important anymore). The engine now returns to the backtrack point to check the next fact, but before it does the check it sets a backtrack point at the third fact “father(“Sue”, “Jim”).”. It then checks the second fact, that doesn’t match the goal either. The engine returns to the backtrack point. As this is the last fact with predicate “father”, the engine cannot set another backtrack point. So it checks if there is a match and yes the fact and the goal match: the predicate and the objects match. So the engine reports “Yes” to you.

Suppose now that you ask the goal:

```
?-father(“Sue”,X).
```

“X” is upper case, it is a variable that has not yet a value. I should liked to have used “Father” instead of “X”, but it may be confusing when there are two identifiers “father”(the predicate) and “Father” (the variable) in the same goal.

You hit <Enter> and the story is repeated more or less. The inference engine takes the goal and sees that the predicate “father” in the goal matches with facts in the program. It sets a backtrack point at the second fact “father("Pam","Bill").” and then takes the first fact “father("Bill", "John").” and sees if it matches the goal. It clearly doesn’t as “Bill” is different from “Sue”. The engine now returns to the backtrack point to check the next fact, but before it does the check it sets a backtrack point at the third fact “father(“Sue”, “Jim”).”. It then checks the second fact, that doesn’t match the goal either. The engine returns to the backtrack point and checks if there is a match between the goal and “father(“Sue”,”Jim).”. The predicate and object “Sue” match, but how about the variable? Well in this case, the variable “X” does not yet have a value. So the inference engine is free to make a match by giving the value “Jim” to “X”. In Prolog this is called “to bind a variable”. Now that X is bound to “Jim”, the goal and fact match. The engine reports this by reporting “X= “Jim” “. We call that the solution of the goal.

When the inference engine has found a solution, it doesn’t stop. When there are more facts to check, it continues till it runs out of facts. This is sometimes called the relentless search for solutions.

In the examples so far I only used a simple goal and facts. Now let’s take a look at an example with a compound goal and facts and rules. Consider the clauses in this program:

```
mother("Bill", "Lisa").
father("Bill", "John").
father("Pam", "Bill").
father("Jack", "Bill").
parent(Child, Parent) :-
   mother(Child, Parent);          % Mind the semicolon!
   father(Child, Parent).
```

And then consider this goal:

```
?- father(AA, BB), parent(BB, CC).
```

This goal states that we want to find the names of three persons AA, BB and CC, such that BB is the father of AA and CC is a parent of BB. All three identifiers start with upper case, so at the beginning they are variables and they are not bound to a value.

As mentioned before the inference engine always solves the goal from left to right, so first it calls the father predicate. When executing the father predicate it goes along the clauses in the program from top to bottom. The first predicate is the mother-predicate. It is skipped because the engine looks for a father-predicate. The next predicate in the program is the first father-predicate. The predicate name matches what the engine is looking for so it will be evaluated. But first the engine creates a backtrack point to the second father-predicate clause, and then it uses the first clause for evaluation.

Using the first clause we arrive at a solution when AA is made equal to "Bill" and BB is made equal to "John". As AA en BB are variables that have not yet a value, nothing stands in the way to do so. So the engine binds AA=”Bill” and BB=”John”. It now has a solution for the first sub-goal, so it turns to the second sub-goal

```
?- parent(BB, CC).
```

But BB has just got a value. So we now effectively have the goal:

```
?- parent("John", CC).
```

The inference engine now calls the parent predicate, i.e. it searches the program for a predicate “parent” and it arrives at the last clause. The rule in that clause says that

Parent is parent of Child **if** Parent is mother of Child **or** Parent is father of Child

The question now is, can the inference engine use this rule. The answer depends on the match between the sub-goal and the conclusion of the rule. If they match, then the rule can be used; if they don’t match, then the rule cannot be used. In this case the sub-goal

```
parent(“John”,CC).
```

matches the conclusion

```
parent(Child,Parent).
```

when the engine binds Child to “John”. Using a rule means that the conclusion is replaced by the premisses. But before doing so, the engine must take care of the bindings that were made when the rule was matched with the sub-goal. In the conclusion Child was bound to “John”, so in the premisses the engine binds Child also with “John”. Effectively the rule looks now like:

```
parent(“John”,Parent) :- mother(“John”,Parent); father(“John”,Parent).
```

Now the engine is ready to make the substitution in the current goal. The current goal was

```
parent(“John”,CC).
```

It becomes

```
?- mother("John", CC); father("John", CC).    % Mind the semicolon
```

The variables in the clause have been replaced with the actual parameters of the call (exactly like when you call subroutines in other languages).

The current goal is an "or" goal. It means that when the first premiss is not met, we have another chance at the second one. So the engine creates a backtrack point to the second alternative and evaluates the first. You should realize that there are now two active backtrack points, one in the sub-goal to the second alternative (father(“John”,CC)) and one in the program to the second clause in the father predicate.

After the creation of this backtrack point the engine takes care of the sub-goal:

?- mother("John", CC).

So it calls the mother predicate. The mother predicate fails when the first argument is "John" (because it has no clauses that match this value in the first argument). In case of failure the engine backtracks to the last backtrack point it created. That is the point in the goal, so it will now pursuit the goal:

?- father("John", CC).

When calling father this time, we will again first create a backtrack point to the second father clause. Recall that we also still have a backtrack point to the second clause of the father predicate, which corresponds to the first call in the original goal. Let’s make a sketch of the situation at this moment.
There is a goal to solve. It was:

?- father(AA, BB), parent(BB, CC).

But it has changed, because the engine found a solution for the father predicate and a rule for the parent predicate. So it now looks like:

father(“Bill”,”John”), mother(“John”,CC); father(“John”,CC).

The first sub-goal is solved, it set a backtrack point at the predicate father(“Pam”,”Bill”) in the program.
The second sub-goal was evaluated and failed as the mother predicate in the program did not match.
The third sub-goal is under evaluation. It is about to evaluate the first father predicate in the program, so it set a backtrack point at the second father predicate. The (partial program look like:

```
                                mother("Bill", "Lisa").
                                father("Bill", "John").
Backtrack point 1st sub-goal->  father("Pam", "Bill"). <-backtrack point third sub-goal.
                                father("Jack", "Bill").
```

By now it may seem very confusing, so I suggest that you use two colored ball pens to represent the backtrack points and move them according to the creation of backtrack points.

The engine now tries to use the first father clause to solve the third sub-goal, but that fails, because the first arguments do not match (i.e. "John" does not match "Bill"). Therefore it backtracks to the second clause, but before it uses this clause it creates a backtrack point to the third clause. (Move your ball pen!). The second clause also fails, since "John" does not match "Pam", so the engine backtracks to the third clause. (No new backtrack point, remove the ball pen). This also fails, since "John" does not match "Jack".

So the two premisses of the rule did not bring a solution. Does that mean there is no solution at all? No, there are still some chances to find a solution, because there is still a backtrack point left. The

engine backtracks all the way back to the first father call in the original goal; here we created a backtrack point to the second father clause. The engine sets a backtrack point to the third clause and evaluates the second one. Using the second clause it finds that AA is "Pam" and BB is "Bill". So we now effectively have the goal:

```
?- parent("Bill", CC).
```

When calling parent we now get:

```
?- mother("Bill", CC); father("Bill", CC).
```

Again the engine creates a backtrack point to the second alternative and pursuits the first:

```
?- mother("Bill", CC).
```

This goal succeeds with CC being "Lisa". So now there is a solution to the goal:

```
AA = "Pam", BB = "Bill", CC = "Lisa".
```

When trying to find additional solutions the engine backtracks to the last backtrack point, which was the second alternative in the parent predicate:

```
?- father("Bill", CC).
```

This goal will also succeed with CC being "John". So now we have found one more solution to the goal:

```
AA = "Pam", BB = "Bill", CC = "John".
```

The engine continues to try to find solutions and there are some more. It will find:

```
AA = "Jack", BB = "Bill", CC = "John".
AA = "Jack", BB = "Bill", CC = "Lisa".
```

After that the engine will experience that everything will eventually fail leaving no more backtrack points. So all in all there are four solutions to the goal.

Phew! Take a short break and feel good. When you understand the work of the inference engine till here you won't have problems with Prolog as a programming language. Of course a program can be more complicated (and in the next sections it will) but when you understand the basics till now, you can handle also greater complexity.

5.5.5 Preventing Backtracking: the Cut[7]

Backtracking leads to what is called "the relentless search for solutions" by the inference engine. The engine tries to find every possible solution. Sometimes that is OK, but sometimes it's not. Think of the case that you one solution will satisfy your goal. E.g. when you are looking for a piano tuner, you look into the yellow pages and probably you are satisfied with the first name that you see. No need to search the complete database for every piano tuner in town. In this case, you want the program to stop after it finds the first solution.

Visual Prolog contains the cut, which is used to do this. It prevents backtracking and thus prevents the search of more solutions. The cut is written as an exclamation mark "!". The effect of the cut is

[7] This section leans heavily on the language tutorial that came with Visual Prolog 4.0

simple: once you pass the cut it is impossible to backtrack to predicates that precede the cut. The cut is like burning your resources behind you. Whenever you pass it, there is no way back to an alternative.

You place the cut in your program in the same way you place a subgoal in the body of a rule. When processing comes across the cut, the call to cut immediately succeeds, and the next subgoal (if there is one) is called. Once a cut has been passed, it is not possible to backtrack to subgoals placed before the cut in the clause being processed, and it is not possible to backtrack to other clauses defining the predicate currently in process (the predicate containing the cut).

There are two main uses of the cut:

- When you know in advance that certain clauses will never give rise to meaningful solutions, it's a waste of time and storage space to use them in looking for alternate solutions. If you use a cut in this situation, your resulting program will run quicker and use less memory. This is called a green cut.
- When the logic of a program demands the cut, to prevent consideration of alternate subgoals. This is a red cut.

How to Use the Cut

Here I give examples that show how you can use the cut in your programs. In these examples, we use schematic Visual Prolog rules (r1, r2, and r3), which all describe the same predicate r, plus several subgoals (a, b, c, etc.). The content of the rules is not very important as the focus here is on the program flow.

1. Prevent Backtracking to a Previous Subgoal in a Rule

Take a look at this rule:

```
r1 :- a, b, c.
```

Rule r1 has three subgoals, a, b and c. Normally the inference engine will first evaluate subgoal a, then b and finally c. When c fails, it would backtrack to b (if there were any backtracking points with b) or maybe back to subgoal a. Now let’s put a cut in this clause.

```
r1 :- a, b, !, c.
```

The cut is the exclamation mark; it is placed like and looks like any other subgoal. Now there is a cut between subgoal b and c. This cut succeds as a subgoal, so the engine passes it. The side effect is that when the inference engine passes the cut, any subgoal in front of it (here: a and b) will no longer be considered as a potential backtrackpoint. This is a way of telling Visual Prolog that you are satisfied with the first solution it finds to the subgoals a and b. Although Visual Prolog is able to find multiple solutions to the call to c through backtracking, it is not allowed to backtrack across the cut to find an alternate solution to the calls a or b. Also it is not allowed to backtrack to another clause that defines the predicate r1. So all in all you lose the backtracking of the the subgoals in this clause AND you lose the backtracking of alternate clauses of the same predicate r1.

As a concrete example, consider the next clauses.

```
person(“Pete”, “haircutter”, 20).
person(“John”, “piano_tuner”, 50).
person(“Anna”, “piano_tuner”, 40).
person(“Carl”, “piano_tuner”, 44).
```

There are four persons in a database. Each person has a name, a profession and asks a certain amount of money for a service. Assume that you are looking for a piano tuner, but you don't want to pay more than 44 Euro's for tuning a piano. Then you could have the enference engine search this database with the goal:

```
run :-
        person(Name,"piano_tuner", Amount),
        Amount < 45,
        write(Name).
```

In this case the inference engine should come up with two names, Anna and Carl, that meet the requirements. But take care, there is a peculiarity in PIE. When you enter the goal in the dialog window, it works as expected. But when you enter the goal in the program and invoke it in the Dialog Window with

```
run.
```

then PIE only shows the first solution. To see every possible solution in this case, add an arbitrary free variable to the command in the Dialog Window. E.g.

```
X = 1, run.
```

will give you both solutions.

But two solutions is too much. Given your requirements, the solution Anna suffices. To make the engine stop after the first solution, we introduce the cut:

```
run :-
        person(Name,"piano_tuner", Amount),
        Amount < 45, !,
        write(Name).
```

I place the cut after the subgoal that fulfills the last requirement needed. In this case when

```
Amount < 45
```

succeeds, you know for sure that the solution is feasible and that there is no need for backtrtacking to another person. What happens is that the engine skips Pete as he is no paino_tuner. Then it finds John, but the subgoal

```
Amount < 45
```

fails, so the engine backtracks to person Anna. Now the subgoal

```
Amount < 45
```

succeeds, the engine passes the cut and loses every backtrack point that it has in this clause and in the predicate "run". So it writes the name "Anna" and stops.

2. Prevent Backtracking to the Next Clause

The cut can be used as a way to tell Visual Prolog that it has chosen the correct clause for a particular predicate. And that there is no need to go to another clause for this predicate. For example, consider the following code:

```
r(1):- ! , a , b , c.
r(2):- ! , d.
r(3):- ! , c.
r(_):- write("This is a catchall clause.").
```

Using the cut makes the predicate r deterministic. That means that of the predicate “r()“ only one clause will be used. It’s like looking in a telephone guide for a number: only one entry will do (as long as there is one telephone number per person). Here, Visual Prolog calls r with a single integer argument. Assume that the call is r(1). Visual Prolog searches the program, looking for a match to the call; it finds one with the first clause defining r. Since there is more than one possible solution to the call, Visual Prolog places a backtracking point at the clause for r(2). The clause for r(1) fires and Visual Prolog begins to process the body of the rule. The first thing that happens is that it passes the cut; doing so eliminates the possibility of backtracking to another r() clause. This eliminates backtracking points and so increases the run-time efficiency. It also ensures that the error-trapping clause is executed only if none of the other conditions match the call to r(). Note that this type of structure is much like a "case" structure written in other programming languages. Also notice that the test condition is coded into the head of the rules. You could just as easily write the clauses like this:

```
r(X) :- X = 1 , ! , a , b , c.
r(X) :- X = 2 , ! , d.
r(X) :- X = 3 , ! , c.
r(_) :- write("This is a catchall clause.").
```

However, you should place the testing condition in the head of the rule as much as possible, as doing this adds efficiency to the program and makes for easier reading.

As another example, consider the following program. Run this program and give it the goal:

```
friend(“bill”, Who)
```

to find a friend of Bill.

```
friend(“bill”,”jane”):-
        girl(“jane”),
        likes(“bill”,”jane”),!.
friend(“bill”,”jim”):-
        likes(“jim”,”baseball”),!.
friend(“bill”,”sue”):-
        girl(“sue”).

girl(“mary”).
girl(“jane”).
girl(“sue”).

likes(“jim”,”baseball”).
likes(“bill”,”jane”).
```

Without cuts in the program, Visual Prolog would come up with three solutions: Bill is a friend of both Jane, Jim and Sue. However, the cut in the first clause defining friend tells Visual Prolog that, if this clause is satisfied, it has found a friend of Bill and there's no need to continue searching for more friends. A cut of this type says, in effect, that you are satisfied with the solution found and that there is no reason to continue searching for another friend.
Backtracking can take place inside the clauses, in an attempt to satisfy the call, but once a solution is found, Visual Prolog passes a cut. The friend clauses, written as such, will return one and only one friend of Bill (given that a friend can be found).

Determinism and the Cut

If the friend predicate (defined in the previous program) were coded without the cuts, it would be a so-called non-deterministic predicate. That means that the predicate has multiple clauses that can generate multiple solutions through backtracking. In many implementations of Prolog, programmers must take special care with non-deterministic clauses because of the possible demands made on memory resources at run time. However, Visual Prolog makes internal checks for non-deterministic clauses, reducing the burden on you, the programmer.

You can change a non-deterministic predicate into a deterministic predicate by inserting cuts into the body of the rules defining the predicate. For example, placing cuts in the clauses defining the friend predicate causes that predicate to be deterministic because, with the cuts in place, a call to friend can return one, and only one, solution. More on this in chapter 9.

5.6 Recursion

Most family relations are easy to construct given the principles above. But when it comes to "infinite" relations like *ancestor* we need something more. If we follow the principle above, we should define ancestor like this:

```
ancestor(Person, Ancestor) :- parent(Person, Ancestor).
ancestor(Person, Ancestor) :- parent(Person, P1), parent(P1, Ancestor).
ancestor(Person, Ancestor) :- parent(Person, P1), parent(P1, P2), parent(P2, Ancestor).
...
```

The problem with this formalization is that this line of clauses never ends. The way to overcome this problem is to use a principle that is called recursion. Recursion is created by a recursive definition, i.e. a definition that is defined in terms of itself. like this:

```
ancestor(Person, Ancestor) :- parent(Person, Ancestor).
ancestor(Person, Ancestor) :- parent(Person, P1), ancestor(P1, Ancestor).
```

The first clause states that you are an ancestor of a Person if you are a parent of Person. The second clause states that you are (also) an ancestor if you are an ancestor of a parent of Person. If you are not already familiar with recursion you might find it tricky (in several senses). Recursion is however fundamental to Prolog programming. You will use it again and again, so eventually you will find it completely natural.

Let us use the two ancestor clauses in a program with other family clauses

```
parent("Bill", "John").
parent("Pam", "Bill").
ancestor(Person, Ancestor) :- parent(Person, Ancestor).
ancestor(Person, Ancestor) :- parent(Person, P1), ancestor(P1, Ancestor).
```

and try to execute an ancestor goal:

```
?- ancestor("Pam", AA).
```

We identify ourselves with the inference engine and create a backtrack point to the second ancestor clause, and then we use the first clause, finding the new goal:

?- parent("Pam", AA).

This succeeds with the solution:
AA = "Bill".

Then we try to find another solution by using our backtrack point to the second ancestor clause. This gives the new goal:
?- parent("Pam", P1), ancestor(P1, AA).

Again "Bill" is the parent of "Pam", so we find P1= "Bill", and then we have to evaluate the goal:
?- ancestor("Bill", AA).

To solve this goal we first create a backtrack point to the second ancestor clause and then we use the first one. This gives the following goal
?- parent("Bill", AA).

This goal gives the solution:
AA = "John".

So now we have found two ancestors of "Pam": "Bill" and "John". If we use the backtrack point to the second ancestor clause we get the following goal:
?- parent("Bill", P1), ancestor(P1, AA).

Here we will again find that "John" is the parent of "Bill", and thus that P1 is "John". This gives the goal:
?- ancestor("John", AA).

If you pursue this goal you will find that it will not have any solution. So all in all we can only find two ancestors of "Pam".

Recursion is very powerful but it can also be a bit hard to control. Two things are important to remember:

- the recursion must make progress.
  It is no use as recursion leads to an endless loop in one clause. That would produce the same answer over and over.
- the recursion must terminate.
  When recursion doesn't terminate, your program will keep on creating new backtrack points and will never end. Even when you go back to Adam and Eve as ancestors, this is not what you want.

In the code above the first clause ensures that the recursion can terminate, because this clause is not recursive (i.e. it makes no calls to the predicate itself). In the second clause (which is recursive) we have made sure, that we go one ancestor-step further back, before making the recursive call. I.e. we have ensured that we make some progress in the problem.

5.7 Side Effects

Besides a strict evaluation order Prolog also has side effects. For example Prolog has a number of predefined predicates for reading and writing. These predicates are special, because:

- they always succeed
- they do more than simply evaluate to true or false.

In Prolog every predicate is evaluated. In the evaluation the predicate may come true, that we call succeed, or it may turn out to be false (untrue), in that case we say it fails. Now any predicate can in principle succeed or fail. But some predicates can only succeed or fail. You already encountered the predicate *fail*, that always fails.

There are predicates in Prolog that are used for their side effects. These predicates always succeed, so that aspect is not interesting. What is interesting is what they do more than succeeding. This is called their side effect. Important examples of predicates that always succeed and are used for their side effect are predicates for reading and writing. The following goal will write the found ancestors of "Pam":

```
?- ancestor("Pam", AA), write("Ancestor of Pam : ", AA), nl().
```

How does it work? Well, the ancestor call will bind an ancestor of "Pam" in AA. The write call succeeds and as a side effect will write the string literal "Ancestor of Pam : " and then it will write the value of AA. Because “write” succeeded, the next sub-goal nl() is executed, The nl call will as a side effect shift to a new line in the output.

When running programs in PIE, PIE itself writes solutions, so the overall effect is that your output and PIE's own output will be mixed. This might of course not be desirable. A very simple way to avoid PIE's own output is to make sure that the goal has no solutions. Consider the following goal:

```
?- ancestor("Pam", AA), write("Ancestor of Pam : ", AA), nl(), fail.
```

Fail is a predefined call that always fails (i.e. it has no solutions). The first three predicate calls have exactly the same effect as above: an ancestor is found (if such one exists, of course) and then it is written. But then we call fail; this will of course *fail.* Therefore we must pursue a backtrack point if we have any. When pursuing this backtrack point, we will find another ancestor (if such one exists) and write that, and then we will fail again. And so forth. So, we will find and write all ancestors. and eventually there will be no more backtrack points, and then the complete goal will fail. Imagine: we find every solution, but the goal fails.

There are a few important points to notice here:

- The goal itself did not have a single solution, but nevertheless all the solutions we wanted were given as side effects.
- Side effects in failing computations are not undone.

These points are two sides of the same thing. But they represent different level of optimism. The first optimistically states some possibilities that you can use, while the second is more pessimistic and states that you should be aware about using side effects, because they are not undone even if the current goal does not lead to any solution.

Anybody, who learns Prolog, will sooner or later experience unexpected output coming from failing parts of the program. Perhaps, this little advice can help you: Separate the "calculating" code from the code that performs input/output.
In our examples above all the stated predicates are "calculating" predicates. They all calculate some family relation. If you need to write out, for example, "parents", create a separate predicate for writing parents and let that predicate call the "calculating" parent predicate.

5.8 Conclusion

In this tutorial we have looked at some of the basic features of Prolog. You have seen ***facts***, ***rules*** and ***goals***. You learned about the ***execution strategy*** for Prolog including the notion of ***failing*** and ***backtracking***. You have also seen that backtracking can give ***many results*** to a single question. And finally you have been introduced to ***side effects***.

Chapter 6. Data modeling in Prolog[8]

Loosely speaking a computer program (or wider: an information system) consists of data and procedures to manipulate the data and a user interface. In the previous chapter we paid attention to the basics of how Prolog manipulates data to find answers for the user. In this chapter we will dig deeper into the way the data in a Prolog Program are being structured. You will find out that there are various and elegant ways to create data structures. We are going to concentrate on how data is *modeled* in Prolog, before we can perform actions on them. Hence, there are not much examples concerning code execution here. As in the previous chapter we shall continue to use the **PIE** environment to develop and learn Prolog. We shall get into the **Visual Prolog IDE** in the later chapters

6.1 Domains

In chapter 5 all the people were represented as "Bill", "John" and "Pam" etc. To human beings "Bill", "John" and "Pam" are just the names of the individuals. In Prolog we take a different look at data. To a programming language as Prolog, “Bill” is a sequence of letters (and maybe other signs) delimited by double quotes. Such a sequence is called a string. A string is a way to represent a value. In this case “Bill” is the value that could be assigned to the variable “Name”. Such a way of representation is called a data type. There are several data types available in Prolog. To mention a few:

- char
  this type allows values that consist of one character between single quotes. Examples are ‘a’, ‘b’.
- string
  a string consists of a series of characters between double quotes. Examples are “this is a long string” and “This is a string with @#$%^&*() strange characters”.
- integer
  an integer is an integral signed number. When the sign is omitted, it is assumed to be positive.
- unsigned
  values of the type unsigned are integral unsigned numbers
- real
  values of the type real are (signed) numbers that may contain a decimal part.

A data type describes the values that may occur. You may think of a data type as a set of possible values. In Prolog such a set of values is called a domain. The domains mentioned here are called simple data types or ***simple domains*****.** In the case of the names of the people, the ***simple domain*** was a ***string***. A domain offers a set of values for a variable. The domain “numbers” offers a range of numeric values. Most of these domains will be familiar to you and offer traditional values that you expect from it.

Simple domains offer values that are used to describe one characteristic. However, individuals are represented by more characteristics than just their names. What if we need to represent all those characteristics together, instead of representing only one? In that case we look for a way to represent a

[8]This chapter is a slight remake of the tutorial Fundamental Prolog (Part 2) by Sabu Francis (Tangential Solutions) and Thomas Linder Puls (PDC)

set of characteristics. It means, we need some mechanism to represent ***compound domains***; a collection of simpler ***domains*** held together in one package.

6.2 Improving the Family Theory

If you continue to work with the family relation from chapter 5, you will probably find out that you have problems with relations like *brother* and *sister*, because it is rather difficult to determine the sex of a person (unless the person is a father or mother).

The problem is that we have chosen a bad way to formalize our theory. The reason that we arrived at this theory is because we started by considering the ***relations*** like father and parent between the entities or objects. If we instead first focus on the ***entities***, then the result will become different. Our main entities are persons. Persons have a name (in this simple context we will still assume that the name identifies the person, in a real scale program this would not be true). Persons also have a gender. Persons have many other properties, but none of them have any interest in our context.

Therefore we define a person predicate, like this:

```
person("Bill", "male").
person("John", "male").
person("Pam", "female").
```

The first argument of the person predicate is the name and the second is the gender. If you want to see a predicate as a relation, you can think of the person predicate as a relation between a name and a gender. Or as a relation within an entity instead of between entities. A predicate like this is called a functor. More about functors in section 6.3.

Instead of using mother and father as facts, I will choose to have parent as facts and mother and father as rules:

```
parent("Bill", "John").
parent("Pam", "Bill").
father(Person, Father) :- parent(Person, Father), person(Father, "male").
mother(Person, Mother) :- parent(Person, Mother), person(Mother, “female”).
```

Notice that when father is a "derived" relation like this, it is impossible to state *female* fathers. So this theory also has a built-in consistency on this point, which did not exist in the other formulation.

You see that it is possible to formulate a theory in several different ways. It may seem confusing (and to some extent it is) but it resembles the fact that any theory can be formulated in different ways, as can relations between individuals. It is up to you to choose the best way. But maybe we should warn you. As always in science (and life) there is seldom only one best way.

Here we specified the data of a certain person as a list of arguments. However, there is another elegant method to sharpen our focus on the ***entities*** being represented. We can pack both the Name and the Gender into a package using a formalization known as a ***compound domain***. Think of it as a set of values that characterizes a person. That has some advantages as Prolog offers you the

possibility to represent the entire package using one variable, but you can also reach each argument apart from the other. As an example, in Prolog you can make the statement

P = person(“Bill”,”male”).

Then the variable P is bound to (represents) the “whole” person Bill. But also this statement is possible:

person(Name,Gender) = person(“Bill”,”male”).

In this case the variable Name is bound to the value “Bill” and the variable Gender is bound to “male”.
The advantages of this will become clearer in the next section.

6.3 Compound domains and functors

The above facts about persons can be generically represented in a declaration for a ***compound domain*** thus:

person = person(string Name, string Gender)

This declaration says that with “person” you can expect two arguments, both of type “string” one is the Name, the other one is the Gender. Note that for the compiler the “string” is important as a type declaration, the words Name and Gender are there for human readers. Note also that this is neither a fact nor a predicate. A fact contains only one value, while this “person” contains two values. It is also not a predicate, because a predicate is used to manipulate data while in “person”we only store data of one person. The declaration states that there is some ***compound domain*** called ***person*** in the system, and that each entity from that domain. has two characteristics, represented by the logical variables Name and Gender. Remark that we talk about “values” from a simple domain and about entities from a compound domain. That is because an entity from a compound domain consists of more than one value. Each of those values comes from a simple domain.

The word “person” is known as a ***functor***, and the variables are its ***arguments***. We shall now package our facts using these ***functors***. As a ***compound domain*** always has a ***functor***, we shall henceforth use the term ***functor*** in this chapter, to represent the respective ***compound domain.***

For now, let us modify the first example of the previous chapter, so that we use our newly defined ***functor*** *person***.** So now we thnk of a person as a set of two values. It means that we specify Bill as:

person(“Bill”,”male”).

And we specify Pam as

person(“Pam”,”female”).

To indicate that Bill is the father of Pam, we now have to write

father(person("Pam","female"),person("Bill","male")).

The pattern is the same as when we wrote

father(“Pam”,”Bill”).

in chapter 5, the only difference is that now the arguments are from a compound domain. The family program becomes:

```
father(person("Bill","male"),person("John","male")).
father(person("Pam","female"),person("Bill","male")).
```

```
father(person("Sue","female"),person("Jim","male")).
grandfather(Person,Grandfather) :-
    father(Father,Grandfather),
    father(Person,Father).
```

Enter these lines in the program editor of PIE, the result will look like in figure 6.1

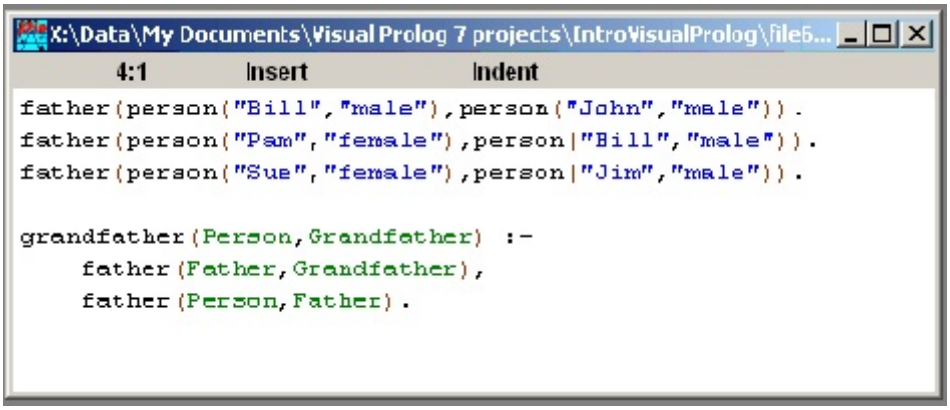


Figure 6.1 A program with compound domains

Please note that in the **PIE** (Prolog Inference Engine) that we are using, we can directly use a ***compound domain*** without any prior intimation to the Prolog engine (for example. we need NOT declare a predicate or a fact called *person* for our code to work). In fact you can declare a compound domain in PIE by simply using it in your predicates. This is different from Visual Prolog where you have to declare what you are going to use. I will come back to that in chapter 9.

If you study the facts for the father relationship, you will notice that the persons described are richer than what was done before. This time, each person is described with both the person's name ***and*** his/her gender, using the person ***functor***, whereas in chapter 5 (Fundamental Prolog Part1) we were only using the person's name.

After you've written the new code, ensure that the PIE engine is reset. Use *Engine/Reset*. Then, you re-consult the code using *Engine/Reconsult*.

As before, on a blank line in the Dialog Window type a goal (without the "?" sign- in front).
For example type the goal

```
grandfather(X,Y).
```

And hit <Enter>. The inference engine answers

```
X=person(Pam,female), Y=person(Bill,male).
1 solution
```

As is shown in figure 6.2

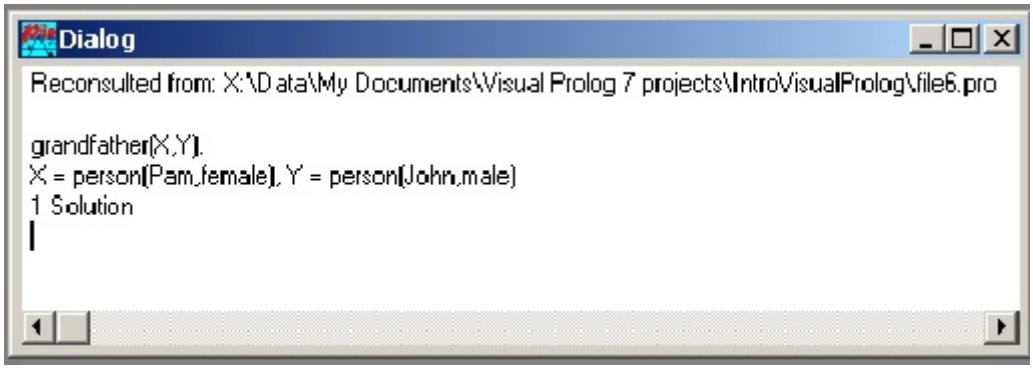


Figure 6.2 Dialog with the family

Notice that in this case variable X contains everything we know about the person Pam, her name and her gender. It is also possible to approach the name and gender separately. Enter in the Dialog this goal:

```
grandfather(person(ChildName, ChildGender),person(GrandName, GrandGender)).
```

Take care of the uppercase letters and the full stop and hit <Enter>. The inference engine answers

```
CHILDNAME = Pam, CHILDGENDER = female, GRANDNAME = John,
GRANDGENDER = male
1 Solution
```

This gives you a detailed answer. More about this in the next section

6.4 Using functors

If you take a look at the grandfather predicate, you will notice that it has a subtle bug: A person normally has two grandfathers: one from the mother's side and one from the father's side, but the grandfather predicate as defined earlier will only yield the grandfather on the father's side.
The grandfather predicate should be re-written as follows:

```
grandFather(Person,TheGrandFather):-
    parent(Person,ParentOfPerson),
    father(ParentOfPerson,TheGrandFather).
```

In this predicate the logic states that a father of *any* parent could be the grandfather of the person in consideration.
For this predicate to work, we need to define a predicate called father using the person ***functor.*** This predicate would troll through a database of facts explaining the "*parents*" defined in the system. This is a more elegant method for finding out the fathers, instead of presenting them as facts (as shown previously) because later on we can extend this concept to find out "*mothers*" in a similar fashion. This can be done in either of the following ways:

/*1st version */
Let us assume that a variable Person is bound to a Name and a Gender and that now we call the predicate

```
father(Person, Father) :-
    parent(Person, Father),              %Line 1
```

```
        Father = person(_, "male").          %Line 2
```

to see if there is a father. This predicate is called with Person bound to a specific value and it gives back the Name of the father. What happens here is that when the predicate “Father(Person,Father)” is called, the inference engine in Line 1 looks in the database to find the “father”-clause that matches the name and gender bound in Person and then binds Father to the second argument. Notice that in this case Father is bound to the “whole” person. In Line 2 the engine checks if the second argument of the newly bound Father is equal to “male”. If it is, the predicate succeeds and a new father is found. In this version, the code in Prolog systematically examines each parent fact asserted into the code, and sees if the first logical variable (Person) matches that passed down from the predicate head. If the variable does match, it checks if the second argument consists of a person ***functor***, whose second argument is the string literal "male".

This example shows one important feature of ***functors***: The arguments of a ***functor*** can be taken apart and examined using regular Prolog variables and bound values (like the string literal in this example) If you see Line 2, you will notice that we have used an underscore. In Prolog this is an anonymous variable, it indicates that there is a value in that place, but that we are not interested in it. Here we use the anonymous variable for the first argument of the person ***functor*** as we are (in that predicate) not interested in the name of the father. We only want to know if there is a father. Notice also the use of the “%”-sign. It is an indication that the rest of the line is a comment and should be skipped by the inference engine.

The other way is this:

```
/* 2nd version */
        father(Person, person(Name, "male")) :-
            parent(Person, person(Name, "male")).
```

This predicate is also called with Person bound to a variable and gives back the Name of the father. The logic behind both versions is the same, but the manner in which the logic is indicated to the Prolog engine is different. In this second version, the set of parent facts is also examined. On arriving at the correct value for Person, the code halts and returns the correct ***functor*** data back to the predicate head, *provided the second argument of that* ***functor*** *is the string literal "male"* . If you note, the functor was NOT bound to any intermediate Prolog variable, as it was done in the first version. The second version is much terser than the first one, and sometimes this method of writing the code can be less legible to beginners. So feel free to use the first version as it is correct Prolog code.

Let us now use this code in a complete example, where we will give a suitable set of person facts: Please enter into the Program Editor of PIE the following clauses:

```
parent(person("Bill","male"),person("John","male")).
parent(person("Pam","female"),person("Bill","male")).
parent(person("Pam","female"),person("Jane","female")).
parent(person("Jane","female"),person("Joe","male")).

grandfather(Person,Grandfather) :-
   parent(Father,ParentOfPerson),
   father(ParentOfPerson,GrandFather).

father(Person, person(Name, "male")) :-
```

parent(Person, person(Name,"male")).

The result should look like in figure 6.3

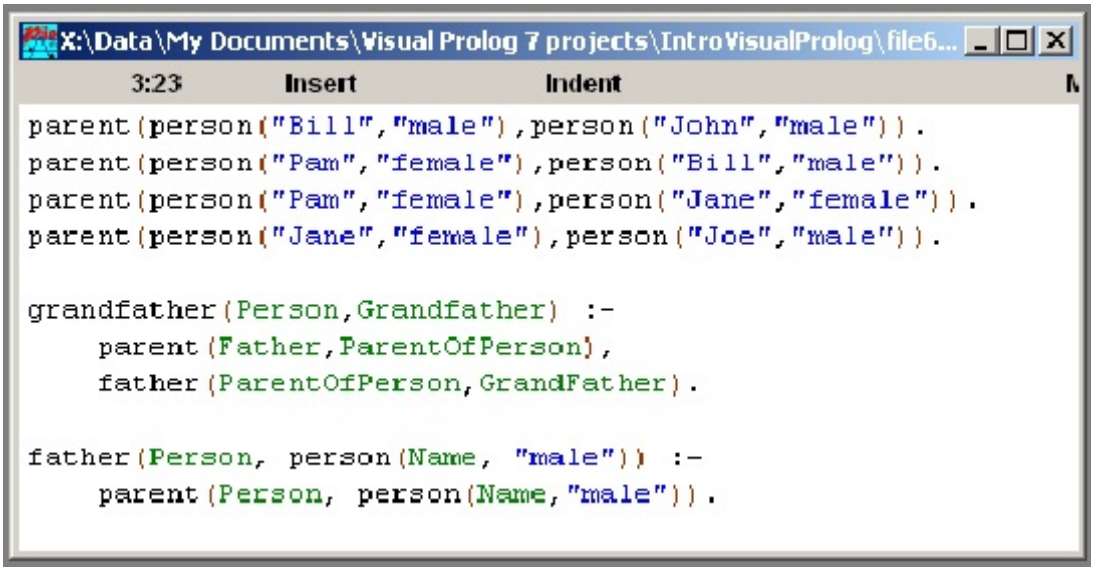


Figure 6.3 Another family theory

Now, after you reset the inference engine and reconsult the above code into **PIE**, and give the following goal:

grandFather(person("Pam","female"),W)

the result will be:

grandFather(person("Pam","female"),W)
W = person(John,male)
W = person(Joe,male)
2 Solutions

We get the names of both grandfathers. Feel free to fool around with this program, add some more clauses and ask different goals. You need to get well acquainted with functors as the use of functors in Prolog is important.

6.5 Functors and Predicates

Functors and predicates look almost the same, but they are very different.Technically, a ***functor*** represents a logical function that binds several domains together. In simpler words, a ***functor*** is a mechanism that teaches the Prolog Engine how to put together data from its component parts. It effectively puts parts of the data into a common box. That's why I said before that a functor is for storing and retrieving data values. You can also retrieve the parts subsequently, whenever required (like seen in the previous examples). It may look like a Prolog fact or a predicate call, but it isn't. It's just a piece of data, which you can handle in much the same way as a string or a number.

Please note that a ***functor*** has nothing to do with a function in other programming languages. It does not stand for some computation to be performed. It simply identifies the ***compound domain*** and holds

its arguments together. When you know the programming language Pascal, then you may compare a compound domain to the “record” type in Pascal.

When you study the above example, you will notice that nothing special needs to be done when logical variables are used to stand in for data represented by ***functors***. The logical variable that is used to stand in for such data, is written just like any other logical variable: a word starting with a capital letter. Thus, if we take a look at the *grandFather* predicate in the example in this tutorial; you see that nothing has changed when you compare it with the same predicate that was defined in chapter 5. After all, the logic of that predicate has not changed. So, you would not find any changes to the variables used inside that predicate.

The biggest advantage of using ***a functor*** is that you are free to change the internal arguments of the ***functor*** without changing much of the predicates that uses such a ***functor*** as you keep revising the code. That means, if you decide (in a later version of the code you are writing) to have one more argument for the functor *person*, you still need not change anything in the *grandfather* predicate. That is to say, as long as you bind the functor “as a whole”. If you want to retrieve a specific value from within the functor, then you could be obliged to change the calling code.

6.6 Functors as arguments

In the previous section, the person ***functor*** was having two arguments: the Name and the Gender. Both happened to be simple domains, in this case both were of domain “string”. The values that are used, like “Bill” and “male” are in general called constants as they do not change in the course of the program. However, there is nothing that prevents us from putting a ***functor*** itself as an ***argument*** of another ***functor***.
Suppose, you wanted to define a ***functor*** for a *couple* (i.e. a husband and a wife). Here is an example of how you would use such a ***functor***: A couple consists of a male and a female, at least let us assume that. So a couple is a kind of package of a male and a female and their names. A clause for the functor couple could look like:

```
couple(person(“John”,”male”),(“Anna”,”female”))
```

In your program code there could be a predicate “myPredicate()” that could do something with a couple:

```
myPredicate(ACouple):-
    ACouple=couple(person(Husband,"male"), person(Wife,"female") ),
        ...
```

In this example, you will notice that the ***functor*** couple is defined by using two ***functors***, both of the compound domain “person”, each of which is a mixture of ***variables*** and ***constants***. This can be done to reflect the logic of the data being represented. The logic used here is that a husband is always a "male" and wife is always a "female"; and a couple consists of a husband and a wife. All of which is consistent with a common interpretation of what one means by a *couple*. But of course it describes only a part of the fascinating reality around us :-).

Though in **PIE** you cannot predefine the kind of grammar ***functors*** can actually have; in **Visual Prolog** you can make such definitions. The advantage of defining a ***functor*** in such a manner is that

later, when you create actual data using this ***functor***, the Prolog engine will always check if the data being created are consistent with the grammar that data was supposed to adhere to.

This comes to another characteristic of ***functors***: the *couple* ***functor*** has been defined with two arguments and the ***position*** of these arguments reflects their logical association. It was explained in chapter 5 that the positions of the arguments of predicates have to be formalized by the programmer, who designs the code. Once formalized; the same positional formalization should be consistently used. The same strategy applies to the creation of **functors**: In the case of the *couple* ***functor***, it so happened that we decided to represent the husband as the first argument, and the wife as the second one. Once this is done, then whenever data are created using such a ***functor***, then we need to ensure that the husband is always at the first position, and the wife in the second.

Now, when we look back on the *couple* ***functor,*** someone can correctly argue that if we are so sure that the first argument is always a husband (who is always a male), and the second one is always a wife (who is always a female) then what is the need for using the strings "male" and "female" in making the predicate call? If you are sure that the first argument is always a husband and the second is always a wife, then the call could be simplified to

```
myPredicate(ACouple):-
    ACouple=couple(Husband,Wife), ...
```

But, when you are not sure which of the two is on the first position, then you probably want to make sure who is who. Here is a version for the call that catches it.

```
myPredicate(Couple):-
   Couple=couple(person(PersonsName,PersonsGender),person(SpouseName,SpouseGender) ),
    ...
```

In the above ***functor***, both the following examples make logical sense:

```
myPredicate(C1):-
   C1=couple(person("Bill", "male"),person("Pam", "female")),...
/*or*/
myPredicate(C2):-
   C2=couple(person("Pam", "female"),person("Bill", "male")),...
```

In the calls you bind the gender of Person to PersonGender and the gender of Spouse to SpousGender. In the remainder of "myPredicate" you will have to find out who is who.

It should be pointed out that in the PIE (and many other Prolog Engines), there is no way to indicate whether a functor will receive simple domain or compound domain arguments, by looking at the variables that define the functor. This stems from the fact that in **PIE**, ***compound domains*** are directly used, without being declared anywhere, other than in the mind of the programmer.
For example, if you wanted to use a ***compound domain*** as:

```
person(Name,Gender)
```

then **PIE** would as easily accept a logical variable such as:

```
regardingAPerson(Somebody):-
    Somebody=person("Pam",
        person("Pam",
            person("Pam",
```

```
                person("Pam","female")
            )
        )
    ), ...
```

which actually does not make any logical sense, in the current context. Luckily, **Visual Prolog**, does a much better job of differentiating between ***simple domains*** and ***compound domains,*** so, when you get to the next chapters you would not have any such problems.

6.7 Recursion Using Functors

When data are described using ***functors***, they can be treated like any other piece of data. For example, you can write a predicate that can be made to recursively search through data which use ***compound domains*** using ***functors***.
Let us get back briefly to the ancestor predicate, which we have used in chapter 5.

In order to determine if somebody is an ancestor of someone else, we used a recursive definition for the predicate, i.e. a definition that is defined in terms of itself. like this:

```
ancestor(Person, Ancestor) :- parent(Person, Ancestor).
ancestor(Person, Ancestor) :- parent(Person, P1), ancestor(P1, Ancestor).
```

This declaration states that a parent is an ancestor, and that an ancestor to a parent is also an ancestor. If you examine the variables Person and Ancestor in the above definition, they can very well stand for either ***simple domains*** or ***compound domains.***

Let us define the data thus:

```
parent(person("Bill","male"),person("John","male")).
parent(person("Pam","female"),person("Bill","male")).
```

If we now ask **PIE** to solve the following goal,

```
P=person("pam","female"),ancestor(P,Who)
```

...this is what we'll get:

```
P=person(Pam,female), WHO = person(Bill,male)
P=person(Pam,female), WHO = person(John,male)
```

The **PIE** engine has recursively examined the parent facts to reveal the two possible solutions to the answer. By the way, in the above query, you also see that we have bound a variable P to the person ***compound domain*** when specifying the goal to **PIE**. This was done to make the code more readable but it also demonstrates again the fact that we can bind a piece of data specified as a ***compound domain*** into any Prolog variable.

6.8 Strategies for Using Functors

Software will be only as good as allowed by the modeled data. By the term *modeling*, we mean the establishment of a relationship between the subset of the outside real-world that is being tackled with

the internal data structures of the software. If the modeling of data is poor, then it is likely that the software will also be poor; or at best, inefficient. This is true of any software written in any programming language. That was the reason that in the latter part of the previous chapter we had tried to focus on the ***entities*** and tried to correct the situation by inserting richer facts. Similarly, we introduced the concept of ***functors*** in this chapter to get even more clarity on the data regarding ***entities*** that are being modeled.

The advantage of Prolog is that it allows easy ***description*** of the real-world data in a form that can be internally utilized by the code in an efficient manner. Simultaneously, it also makes the code very readable by fellow programmers who are working together on a project.

Functors can be used to create any type of ***compound domain*** to help in this modeling process. You would have to carefully examine the various parts of real-world data that you plan to process and convert them using ***functors*** (and other data types that you will encounter in other chapters) keeping in mind their usage in all critical parts of the software. Some data structures that were useful in one area of the software may prove to be a hindrance in other areas.
You should take a holistic approach while zeroing in on the ***functors*** (and other data structures) you plan to use in your software. Pause and look around at all the corners of your software. Only then should you code the necessary ***functors***.

Just thinking carefully about the data structures is not the only thing that is required. You would also need to (often simultaneously) write/modify the various goals and sub-goals (i.e. predicates) and then use the data developed thus far in those predicates. The ***side effects*** of attaining those goals and sub-goals would make your software work and you can get valuable feedback with which you can further refine your data structures.

In this chapter we have not fleshed out a software design holistically. We have only nibbled at bits and pieces of data which establishes the ***feel*** for some kind of real-world data concerning relationship between people (parents, grandparents, families, ancestors etc.) In later chapters we will elaborate these topics, eventually ending up with useful software that requires such data structures.

6.9 Conclusion

In this chapter we learnt that data can comprise of ***simple domains***, or they can be ***compound domains*** represented using ***functors***. We found that the arguments of ***functors*** must be positionally consistent with the logic they were meant to represent. We understood that a ***functor*** need not have just simple domains as arguments, but it can also have other ***functors*** as arguments. We then learnt that data represented as ***functors*** can be used like any regular Prolog variable, and we can even perform all operations; including recursion, on such data.
We also learnt that we must spend time modeling our subset of the real world for which we are developing the software, and get a feel for the data. We should simultaneously experiment with the predicates that we may wish to use with the data that is being modeled, so that they can be refined further.

Chapter 7 Using Forms or Dialogs and Controls: a minimal database

In this chapter we will take a closer look at the form and the so-called controls in it. As an introduction I develop a simple database with some functionality and a simple user interface that consists of a few forms. To these forms we will add the desired functionality. That will make you understand how the controls in the form are used to trigger your program. In appendix A1 you will find a comprehensive description dialogs and forms. The text in that section comes from the Visual Prolog Help File.

In the first sections I shall repeat things that were already said before. If you find that boring, skip it.

7.1 A minimal database

A database consists of a structured set of data and procedures to work with them. In the past several structures have been proposed. In practice the relational database structure is the most applied one. A relational database consists of flat tables, that are somehow connected. The structure of a table is that of a matrix. Table 7.1 shows a table with data about some people

PersonID	Name	Weight	Length	IQ	...
1	John	85	185	110	...
2	Thomas	78	190	150	...
3	Anna	65	175	130	...
4	Lidy	55	160	123	...
...	...	...	...	...	...
...	...	...	...	...	...
	...		...		...

Table 7.1 a table from a database

Every row in the table describes the attributes of one person. So we see that John weighs 85 kilograms, is 185 centimeters tall and has an IQ of 110. A row in a database is called a record. Every column is dedicated to one characteristic (weight, length, et cetera). In databases such a characteristic is called an attribute. When you are used to use statistical programs, then you will notice the resemblance with a data matrix. This is not hard to understand, I think. But two remarks need to be made. The first is that the attribute PersonID is a special one. In the database it is used to identify the record. You may think that the name of a person can be used for that purpose, like you do with your friends. But people do have the same name now and then, in that case the identification is no longer unique. So in databases we tend to use identification numbers like the social security number. The second remark concerns the first row. It is a title row, meant for human readers. In the database this row is omitted from the tables. That means that in the database we only register “1", “John”, “85" and the other data. It is the responsibility of the database designer and programmer to make sure that the data are presented to the user in a way that makes clear what the data mean.

A database consists of many tables, but in this section we will build a minimal database. It will consist of one table. In the table we put data about people. To prevent you from a lot of typing, I will define the table as having five columns that contain respectively PersonID, Name, Weight, Length and IQ of a person, just like in table 7.1. The number of rows is not too important, Let's take the data of the four persons from table 7.1.

To represent a table in Prolog, I shall use a functor. The functor will have the structure:

```
person(id, name, weight, length, iq).
```

I do take care not to use uppercase to prevent that Prolog will interpret it as a variable.

There is a convenient way to store data in a Prolog program. It is with the help of facts. The *facts section* is a special part of a Prolog program. There you can store facts, which will not be a surprise. But what is a fact? The simplest form of a fact is the tuple of an Object, an Attribute and a Value, an OAV-tuple. E.g. I am object Thomas, I have an attribute Length and an observation will tell you that with that attribute there goes a value of 190. I hope you understand that this value means 1.90 meter. When you take a look at table 7.1, you see that it is loaded with facts! In fact every entry in the table is a fact on its own. To represent facts we could write:

```
fact("Thomas ","Length",190).
fact("John","Length",185).
fact(("Anna","IQ",130).
```

Et cetera. But you will remember the functor from chapter 6. Instead of all the separate facts, we can state every fact about one object with a functor. In this case it could be the functor person() with the appropriate arguments. In PIE we could simply use the functor, in Visual Prolog we have to tell VIP that we are going to use this functor. That is done by declaring that we are going to store facts in the form of a functor. At the appropriate place we declare:

```
facts
   person : (string ID, string Name, integer Weight, integer Length, integer IQ).
```

With this declaration we tell that there will be a functor "person" with five arguments.

- The first will be of type string and will be referred to as ID.
- The second will be of type string and will be referred to as Name.
- The third will be of type integer and will be referred to as Weight.
- The fourth will be of type integer and will be referred to as Length.
- The fifth will be of type integer and will be referred to as IQ.

The names ID, Name, et cetera are for the human reader. The compiler skips them.

After this declaration we can use in our program clauses like

```
person("1", "John", 85, 185, 110).
person("2", "Thomas", 75, 190, 150).
```

It may seem strange to make the ID a string, while it is clearly a number, but there are technical reasons to do this. The main reason is that the ID looks like a number, but it is a code, so the type

string is more appropriate. A statistician would say that the scale of measurement is Nominal or Identification.

7.2 The database in VIP

Let's create a new project for the minimal database. I called this project "chap07db" but of course you are free to give it another name. Choose "Exe" as the target. Create and build the project and then it is time to add the part where the database will reside. As it will be a database of people, I decided to create a module to contain the database. To keep the structure of the project clear, I also decided to put this module in a separate package that I called "person".

So first create the package "person". In the Project Tree click on the root and then create the package: click *File/New in existing package*, in the Create Project Item Dialog choose "Package" in the leftmost window and enter the name of the package "person". See figure 7.1.

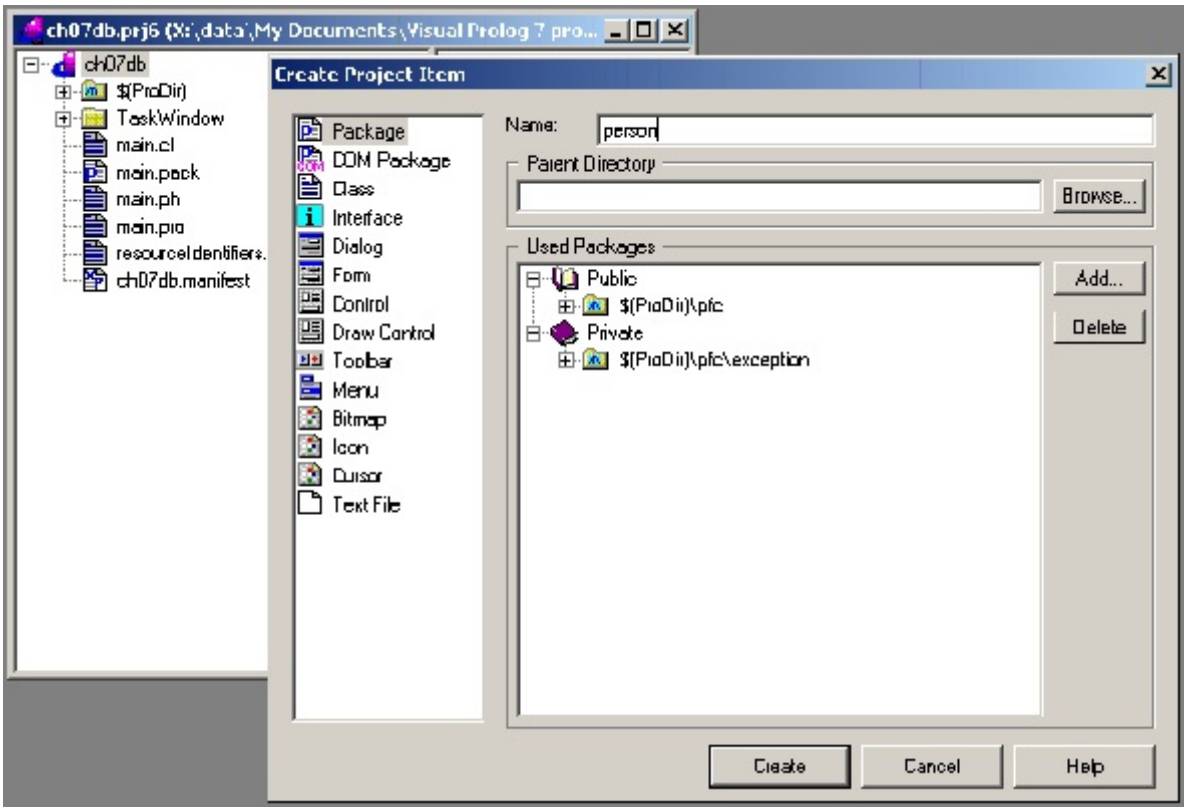


Figure 7.1 Create Package "person"

Click <Create> and the package appears in the Project Tree. Now build the project again (somehow I think it is better to build the project very often, but that may be too often) and now we must create the module where the database will reside.

VIP is an object oriented language. I will elaborate on that in the next chapter, but that means that in this chapter you will encounter things that may seem like a magic formula. Things will become more clear in the next chapter. For now it suffices to know that in an object oriented environment, a computer program consists of so-called classes. You may think of a class as a module, but keep in

mind that this is only a partial description. A class is much more. But for now a class is a fine place to put the database in.

We will create a class and to keep the structure of the project clear, this class will be put in the package “person”. So highlight the symbol for package “person” and click *File/NewInExistingPackage*. In the Create Project Item Dialog that opens, choose Class in the leftmost pane and enter the name of the class “person”. It may seem strange to give the same name to package and class, but it cannot do any harm: class and package are totally different items. Choose further the radio button for Existing Package and make sure the line behind the option says “person.pack”. Finally uncheck Creates Objects, keep In Namespace unchecked and click <Create>. See figure 7.2

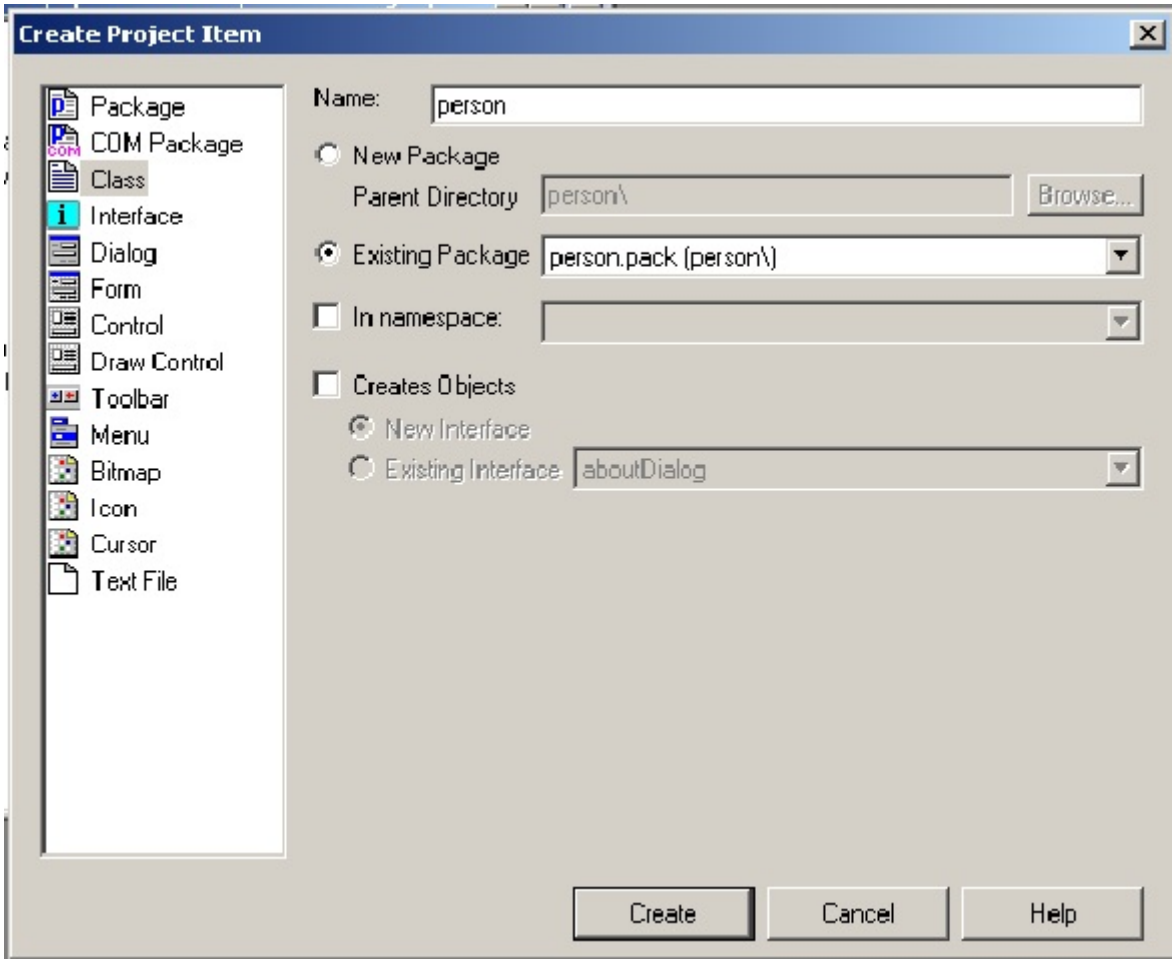


Figure 7.2 Create class “person”

When you click <Create>, the IDE create two files in the package “person”. One is called “person.cl” the other one is called “person.pro”. The IDE opens both files. You can close “person.cl” for the moment, we will attend that one later on. Now go to open “person.pro”. This is the file where we will place the code that we need for the database. First we have to declare that there will be a database. In VIP we use the declaration word “facts” for this. In “person.pro” add a clause for class facts in the code. The parts for constants and clauses are already there.

```
constants
    className = "person/person".
    classVersion = "".

class facts
    person : (string ID, string Name, integer Weight, integer Length, integer IQ).

clauses
    classInfo(className, classVersion).
```

With this declaration you state that there is a database within this class that consists of facts. Each fact is a functor with five arguments, two of them are strings and three are integers. Now save the file and close the editor. As this is probably clear to you, I shall not mention to save the file in the remainder of this chapter.

When you try to execute the program, the database is empty. In fact the compiler warns you that there is an “Unused fact 'person::person/5' “. This tells you that you declared a factvariable but that you don’t use it. To be able to use the database, we must add functionality to the program. It seems to be wise first to fill the database with a few records. I chose to create a menu option in the Task Menu to hold the functionality for the database. So go to the TaskMenu and add a menu item “Database” with the sub-item “FillDatabase”. See figure 7.3. Later we will add more sub-items.

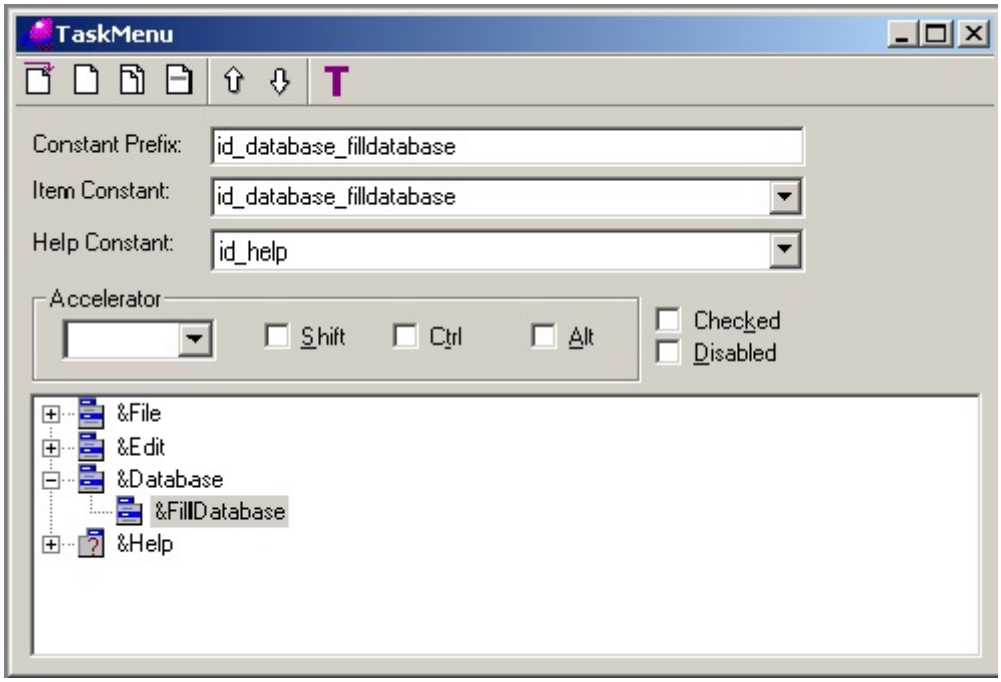


Figure 7.3 Create menu items

So now we have a menu-item to fill the database, but of course this only will work if we add the appropriate code somewhere in our project. I decided to put all the code that has to do with the database in the class (the module) “person”. So now go the file “person.pro” and open it in an editor by double clicking the name in the Project Tree. After the piece of code:

```
clauses
   classInfo(className, classVersion).
```

Enter the following:

```
clauses
        fillDatabase() :-
                assert(person("1", "John", 85, 185, 110)),
                assert(person("2", "Thomas", 75, 190, 150)),
                assert(person("3","Anna", 65, 175, 123)),
                assert(person("4", "Lidy", 55, 160, 135)),
                stdIO::write("Database has been filled"), stdIO::nl.
```

With this code for “assert” you tell the computer to add four records to the database. “Assert” is one of the standard predicates that you can use to manipulate facts in a database.

Now the code is there, but still there is a little problem. This code is now known inside the class person, but not outside. That is a problem, because we want to connect the menu-item in the Taskwindow with this piece of code in class “person”. So we need to state the existence of this code to the other parts of the project. The place to do this is the file ‘person.cl”. Think of it as a counter or a window where you can see what is available in the module “person”. Please open the file “person.cl” and after the lines:

```
predicates
   classInfo : core::classInfo.
   % @short Class information  predicate.
   % @detail This predicate represents information predicate of this class.
   % @end
```

add the line:

```
   fillDatabase : () procedure.
```

Take note of the dot at the end of the line. This line in “person.cl” informs the the other parts of the program outside class “person” of the existence of a predicate “fillDatabase” so it can be used. In the predicate “fillDatabase()” I use the predicate “write()” from the module stdIO. You may have to change in person.pro the line

```
        implement person
           open core
into
        implement person
           open core, stdio
```

Now let us connect the menu-item “FillDatabase” with the predicate “fillDatabase”. We’ll do it in the usual way. Go to the TaskWindow.win, right click and choose the Code Expert. In the Code Expert (actually the Dialog and Window Expert) open Menu / TaskMenu / id_database, highlight id_database_fillDatabase and <Add> code for this option. See figure 7.4

Then go to the code by double clicking the line with “id_database_filldatabase->...” and change the inserted code into:

```
predicates
    onDatabaseFilldatabase : window::menuItemListener.
clauses
    onDatabaseFilldatabase(_Source, _MenuTag) :-
        person::filldatabase().
```

This code tells that when you click on menu option “FillDatabase” the program should call the predicate “fillDatabase” that resides in the class “person”. The names of the class and the predicate are connected by a double colon. It indicates that “person” is a class name and “fillDatabase” is a predicate name.

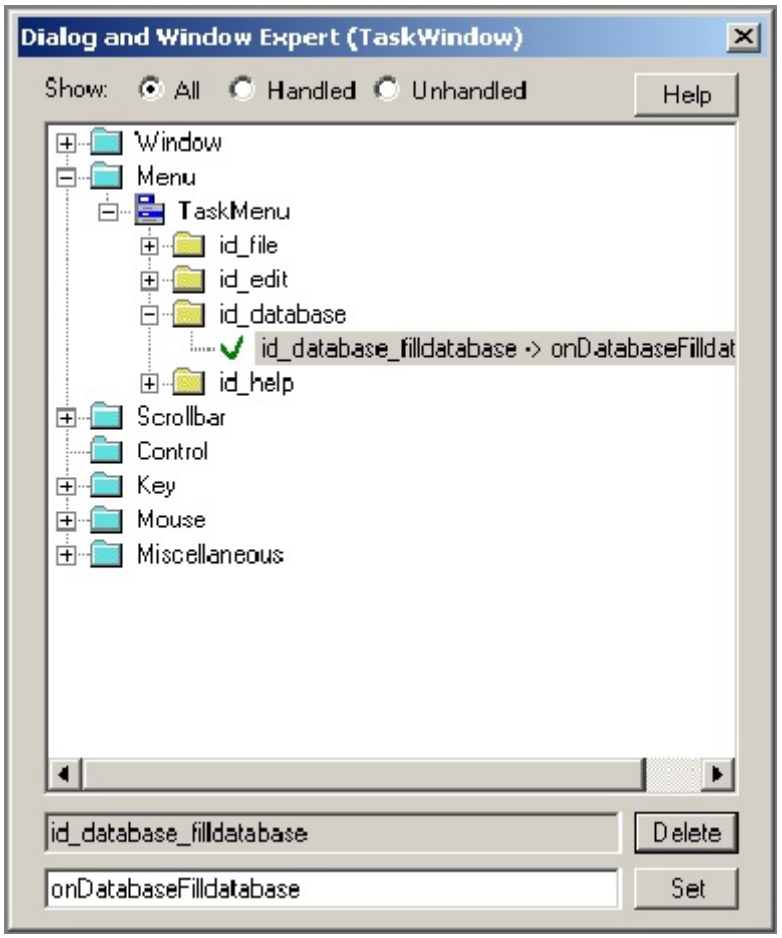


Figure 7.4 adding code for “fillDatabase”

When you now run the program and click on the menu item *Database/FillDatabase*, then the code for “fillDatabase” in the class (module) “person” will be executed and the four records will be added to the database. Go ahead and try it. Close the files that are open and execute the program. If you get some questions if you want to include one or more files, answer “yes”. When you click the option Database/FillDatabase, the records are added to the (empty) facts database and a message is shown in

the Messages Window. By the way, in this chapter I'll do all the talking via the Message Window. That's because it takes the least amount of typing, but of course you can add nice notes and other popup windows if you like.

The database is filled, but you don't see any record. You only get a lousy message, but who believes a computer? So let's add a predicate to list the records in the database. The steps that you need to take are very similar to the ones you took to fill the database, so I'll give the necessary actions and code without much explanation.

First enter the code for a predicate "listDatabase" in the file "person.pro". In my project I entered the following code right after the code for "fillDatabase".

```
 listDatabase() :-
     person(ID, Name, Weight, Length, IQ),
     stdIO::write("ID: ", ID, ", Name: ", Name, ", Weight: ", Weight, ", Length: ", Length, ", IQ: ", IQ),
     stdIO::nl,
     fail.
listDatabase().
```

Here I apply a trick you learned in chapter 5. When we call predicate "listDatabase, the first record in the facts database is matched and the write predicate in stdIO is called. As a side effect the values are written to the Message Window. Then the predicate fails. The inference engine backtracks and the next record in the database is matched and printed. Then the predicates fails again, et cetera. When there are no more records to match, the predicates backtracks to the second clause en there nothing happens, so the predicate call ends in silence. But in the mean time the records have been written to the Message Window.

In version 7.1 another way is introduced instead of using the trick with "fail". A set of three connected keywords was introduced: foreach - do - end foreach. Take a look at the following code:

```
 listDatabase() :-
   foreach
     person(ID, Name, Weight, Length, IQ),
   do
     stdIO::write("ID: ", ID, ", Name: ", Name, ", Weight: ", Weight, ", Length: ", Length, ", IQ: ", IQ),
     stdIO::nl,
   end foreach.
```

The keyword searches the clauses for "person" and takes one at a time. When it has found one, that is used in the lines after the word "do". When the search of "foreach" fails because there are no more clauses for "person", then the program jumps to "end foreach" and the predicate succeeds. The code with foreach-do-end foreach should prints the data of every person in the database.

Now go to the file "person.cl" to add a declaration for the predicate "listDatabase" so other parts of the project may use it. Add

```
        listDatabase : () procedure.
```

right after the declaration of “fillDatabase”. Now it is time to add a menu-item to the TaskMenu. Go to the TaskMenu.mnu and add the option “ListDatabase” in the submenu under Database. Then add the standard code for this menu-item. Go to “TaskWindow.win”, right click and open the Code Expert. In the Dialog and Window Expert click Menu, click TaskMenu, click id_database, highlight id_database_listDatabase and <Add> the code. Change it into:

```
predicates
   onDatabaseListdatabase : window::menuItemListener.
clauses
   onDatabaseListdatabase(_Source, _MenuTag) :-
      person::listDatabase().
```

This should do the trick. Build and execute the program. When you click in the program on Database/Listdatabase, you will see the four records appear in the Message Window. Don’t forget first to fill the database, otherwise you will see nothing.

7.3 Manipulating the data: add a record

So far so good. But the fun of a database is to manipulate the data. Adding records, deleting records and changing the content of records. In this section we will add the necessary code to do that. We’ll start with the functionality to add another record.

To enter a record, we need a form. In VIP there are two possibilities that are almost the same. You can create a Dialog or you can create a Form. In this project I shall use Dialogs, but you are free to use Forms, there is hardly any difference. To be honest I even don’t know the difference. In this Dialog we gather the data of the new person (for the new record) and when the user clicks <OK> we shall enter the record in the database.

To keep the structure of the project clear, I shall put the Dialogs in another package, that I shall call “forms” just for fun. So please, go to the Project Tree, click once on the root to highlight it and then add another package called “forms”. Now highlight this new package and add a Dialog. Click *File/NewInExistingPackage*, choose Dialog in the leftmost window and create a Dialog with the name “addPerson”. Be sure to place it in the existing package “forms.pack”.

When you click <Create> the IDE opens the Dialog Editor. Here you see the familiar grid of the new Dialog (or Form) with three buttons <OK>, <Cancel> and <Help> already placed on the grid. Now we are going to enter four Text Fields and four Edit fields so that the Dialog looks like the one in figure 7.5

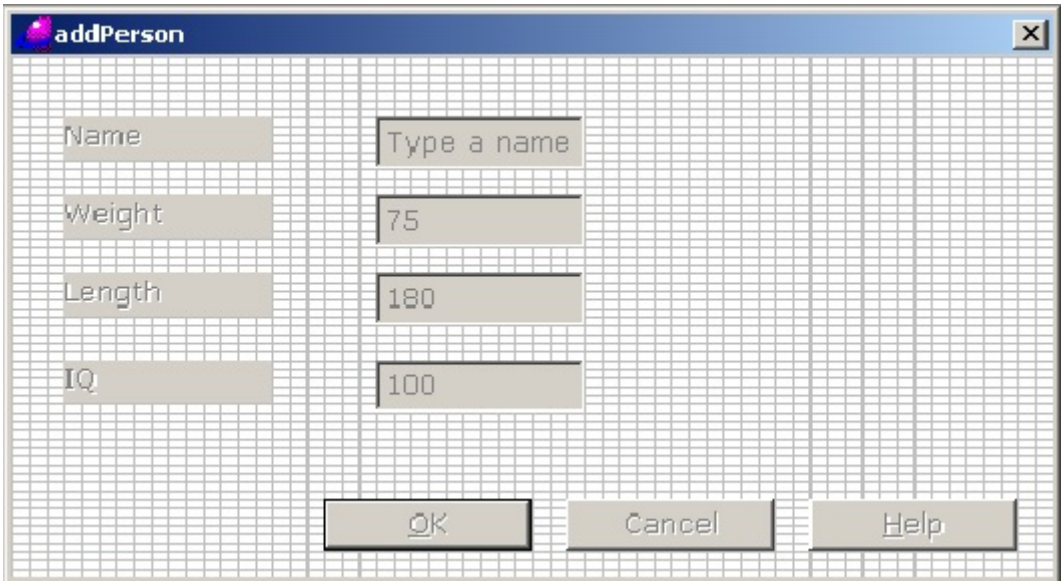


Figure 7.5 the Dialog “addPerson

To enter a text field like the one with the label “Name” in the upper left corner, go to the Controls toolbar and click on the icon upper case letter “T”. The cursor changes into a plus sign and the word “Text”. Go to the Dialog and click the place where you want to drop the text field. The text field gets the focus (if not, click once on the text field) and in the properties list the properties of the text field are shown. In the left column on the third line you will find the word “Text” and in the right column “StaticText”. Change “Static Text” into “Name” and when you hit <Enter> you will see that the label of the text field will have changed. Now enter the other three text fields.

To enter Edit Fields is a little bit more work. To place an Edit Field, go to the Controls Toolbar and select the box with the word “Edit” (top row, most right symbol). The cursor changes in a plus sign and the word “Edit”. Go to the Dialog, place the cursor where you want the Edit Field and click once. The Control gets focus (if not select it) and in the properties list you see the properties of the control. Two properties are important: the Name of the control and the Text. You find the Name on the first row in the properties list. There it says something like:

Name edit_ctl

Now change the Name of this (edit)control into:

Name name_ctl

This edit field is going to contain the name of the person in the new record. Take care that the name of the control “name_ctl” begins with a lower case letter and keep in the end of the name the “_ctl”. It is good practice to change the names of controls into something that has meaning for human beings. Now add the other three controls and give them the names weight_ctl, length_ctl and iq_ctl. You can also change the standard Text “Edit” of the Edit fields, I chose “Type a name” for the Name Edit Field and for the other ones I chose average values. You enter the new text in the Text field in the properties list.

When you insert the controls, you will find out that it is not easy to get them aligned. I think this is the moment to play a little with the Layout Tools in the Layout toolbar. Select a group of controls and align them.

Maybe you wonder why there is no field for the ID in the Dialog. The reason is that the ID should be a unique code. When you enter an ID yourself, you may make a mistake and give the same ID to two different persons. To avoid that, I shall make the program “compute” a unique ID.

Back to the Dialog. When everything is like in figure 7.5 you save and close the dialog editor. Build the project (yes again ...) And when the IDE asks you if you want to include something, say “yes”. Now you have a form but two things are lacking. First we need a menu-item to open the form and then we need code so that when you click <OK> on the form, a new record is added.

First the menu-item, that is rather simple and should be familiar to you. In TaskMenu.mnu enter a new sub-option under the option Database. I called this option “Add Person”. It is not too critical what you enter here, but take care. The text of this option is used to generate code, so when you enter another text here, your code will deviate slightly from mine. When the TaskMenu is changed, go to TaskWindow.win and open the Code Expert. Now choose Menu, choose TaskMenu, choose id_database, and enter the standard code for id_Database_Add_Person. Then go to the code and change it into:

```
predicates
   onDatabaseAddPerson : window::menuItemListener.
clauses
   onDatabaseAddPerson(Source, _MenuTag) :-
      NewDialog = addPerson::new(Source),
      NewDialog:show().
```

When you now click on Database/AddPerson, the Dialog will open and you can enter the data for the new person. Mind you that this does not mean that the record is actually entered in the database. For that we need additional code. We will make a predicate in the module “person” that adds a (new) person to the database. Let us now create the code for this predicate.

All the code to manipulate the database will be gathered in the module “person”. So go to the file “person.pro” (that is the file where the Prolog code is) and enter the following code right after the code for “listDatabase”.

```
addPerson(Name, Weight, Length, IQ) :-
   findAll(ID, person(ID, _, _, _, _), IDList),
   NewID = toString(toTerm(maximum(IDList)) + 1),
   assertz(person(NewID, Name, Weight, Length, IQ)),
   stdIO::write("Person ", Name, " has been added with ID: ", NewID), stdIO::nl.
```

The essential part of the predicate is the middle. It says:

```
      assertz(person(NewID, Name, Weight, Length, IQ)),
```

This is the code to add the record (including the newly computed unique ID) to the database. The standard predicate “assertz” instructs the computer to place the record at the end of the facts list. When you use “asserta” instead, the record is placed at the beginning of the facts list. You can see the difference when you list the database after you entered a new record. (Of course you have to change

the code!). Here I use “assertz”, it is not important here. Also you can use “assert”. This predicate works just like “asserta”.

Before the record is added, we have to find a new unique ID. Here I use two standard predicates to “compute” the new ID. The first one is in the first line. It says:

```
findAll(ID, person(ID, _, _, _, _), IDList),
```

This standard predicate finds all the occurrences (records) that are in the database, gathers the value of a specified field (argument of a functor) and puts the values found in a list. Phew, this needs a lot of explaining.

A list in Prolog is a set of values. A list specifies the elements between square brackets and separates the elements with comma’s. The values can be of any kind, but here I will only use strings and integers. Examples of lists are:

```
[ 1, 2, 3, 4, 5, 6 ]
[“John”, “Thomas”, “Anna”, “Lidy”]
```

The number of elements is not restricted. In fact you can also have a list with no elements. It is called the empty list and looks like this: []. One restriction however, the elements in a list must be of the same type. When I call the predicate “findall”:

```
findAll(ID, person(ID, _, _, _, _), IDList),
```

then the first argument “ID” states that I am looking for the variable ID, the second argument “person(ID, _, _, _, _)” states that this variable can be found on the first position in the functor “person” and the third argument states that I expect findall/3 to bind the list of the values found to the variable IDList. So after the call

```
findAll(ID, person(ID, _, _, _, _), IDList),
```

I know that all the values of ID in the database are stored in the list IDList. Then, in the next statement, I use this list to find the maximum value of ID in the database. You should read this statement inside out. The innermost expression is

```
(maximum(IDList))
```

Maximum/1 is a standard predicate that computes the maximum of the specified list, in this case IDList. The next partial statement is:

```
(toTerm(maximum(IDList)) + 1)
```

Toterm/1 changes the type of the maximum of IDList into an integer and I add 1 to it. So this value is an integer that is 1 more than the maximum of the database. This is a unique number. Finally I bind this number to the variable NewID, but because ID is of type string, I have to change the type from integer to string with the predicate toString/1.

```
NewID = toString(toTerm(maximum(IDList)) + 1),
```

It may look complicated (and frankly I think it is) but it works fine. The variable NewID is used in the assert statement. The predicate is finished with some output to the message Window that states that the record is added and that shows the new ID.

```
stdIO::write("Person ", Name, " has been added with ID: ", NewID), stdIO::nl.
```

We have entered the code for the predicate but we still have to declare this new predicate so other parts of the program can use it. Please go to the file “person.cl” and add the code

```
addPerson : (string Name, integer Weight, integer Length, integer IQ) procedure.
```

Right under the declaration for “listDatabase”.We are now almost done, except for one detail. The standard predicate “maximum()” is in a separate module (in fact it is a class) with the name “list”. We have to tell the program to open this module. Please go to the file ”person.pro” and change the code at the beginning into:

```
implement person
    open core, stdIO, list
```

This completes the code for “person”. But the program misses yet some code to connect the form that gathers the data with the predicate that adds the record to the database. It seems a good idea to relate that code to clicking the <OK> button in the form. So first insert some standard code for clicking the <OK> button. Go to the Project Tree and double click “addPerson.dlg” (or “addPerson.frm” if you made a Form). The Dialog editor is opened. Select the OK-button and in the properties window click the <Events> tab. In the list of events choose “Clickresponder” and in the value field choose “onOkClick”. Now double click on “onOkClick” to go to the code editor. Change the inserted standard code into:

```
predicates
    onOkClick : button::clickResponder.
clauses
    onOkClick(_Source) = button::defaultAction() :-
        Name = name_ctl:gettext(),
        Weight = toTerm(weight_ctl:gettext()),
        Length = toTerm(length_ctl:gettext()),
        IQ = toTerm(iq_ctl:gettext()),
        person::addPerson(Name, Weight, Length, IQ).
```

What is happening here is in fact quite simple. I extract in four subsequent statements the values that are in the four Edit Fields in the form and put the values in four variables. With these four variables I call the predicate

```
person::addPerson(Name, Weight, Length, IQ).
```

There is a little problem here. The value in an Edit Field is by definition a string. As the variable Name is also a string, the value produced by gettext() can be directly bound to Name. But for the variables Weight, Length and IQ I need integers. So before I can bind the values to the variables I first have to convert them with toterm/1.

Now we are ready. Please build and execute the project and then enter one or more new persons. But before you do that, fill the database with fillDatabase. If you don’t, you will get an error. Remember that when you add a new record, the existing records are used to compute an ID for the new record. But when the database is empty, the function “maximum” gets nervous because it cannot find any record. So please first fill the database, then add a record and check the results with “listDatabase”. Whenever you find this bug a nuisance, here’s how to change the code for “addPerson” to prevent the problem

```
addPerson(Name, Weight, Length, IQ) :-
        findAll(ID, person(ID, _, _, _, _), IDList),
        IDList <> [], !,
        NewID = toString(toTerm(maximum(IDList)) + 1),
        assertz(person(NewID, Name, Weight, Length, IQ)),
        stdIO::write("Person ", Name, " has been added with ID: ", NewID),
        stdIO::nl.
addPerson(Name, Weight, Length, IQ) :-
        assertz(person("1", Name, Weight, Length, IQ)),
        stdIO::write("Person ", Name, " has been added with ID: ", "1"),
        stdIO::nl.
```

Don't forget the exclamation mark! Now the program detects it when there are no facts and the new record gets ID number 1. Of course when you then use "fillDatabase", there will be two records with ID=1 because the ID's are hard-coded in the four records that are used in the program. You may want to change the code to prevent this error. But if you think that the program gets too complicated for your cognitive capabilities at this moment, leave it and live with it.

A last remark. When you change this program to better understand it (or just for fun), take care of the fact that the predicate for the <OK> click responder must be a procedure. That means that there must be only one reaction to the click on <OK>. More on procedures in chapter 9. Here it is to sufficient to remember that possible backtracking in your Prolog code must not lead to more than one solution when someone hits the <OK> button or another button.

7.4 Manipulating the data: delete a record

Next we will turn to another database activity, delete a record. How to change a record will be treated in the next section. To delete a record, you first have to find it. In a person database it is convenient to search for persons by name, but there may be a problem when there are persons with the same name in the database. So it is wiser to search for a record by ID as this is a unique identifier. But most people do not know their IDnumber by heart. I shall use a compromise: when we want to find a record we first search by name. The program then will display the ID's that go with this name. You can then click the ID and that determines the record to be deleted.

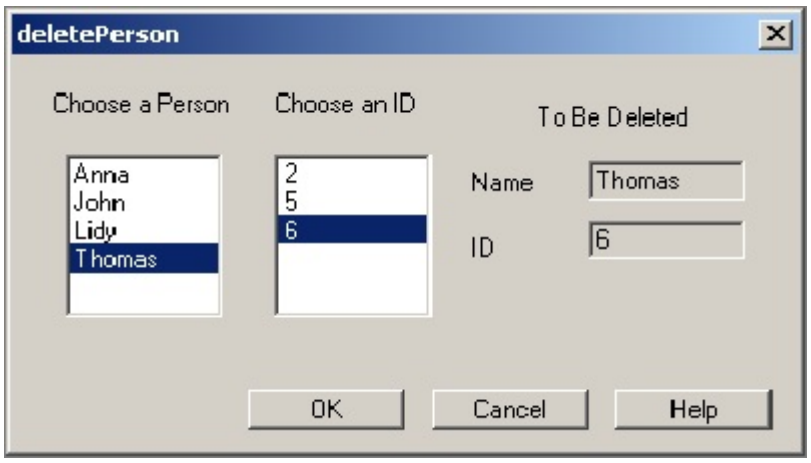


Figure 7.6: Dialog to delete a record

So let's start with creating a Dialog that will do the job. I want it to look to the user like the form in figure 7.6.

In the box below "Choose a Person" the list of available names should appear. When the user selects a name, then in the box next to it, under "Choose an ID", the list of available IDs should appear. If the name is unique, then there appears one ID, when the name is not unique, then all the IDs that go with the name should appear. When the user selects an ID, the corresponding ID and Name appear in the fields on the right side. The user then can click <OK> or <Cancel>. When he clicks <OK>, the record is deleted. When he clicks <Cancel>, the record is not deleted. In both cases the window is closed. I shall not use the <help>-button, but you may connect it to an appropriate help text in a message box.

Now create a new Dialog called "deletePerson" in the Forms package. We will fill it with controls as is shown in figure 7.7 . Most controls should be familiar to you, except the two boxes beneath "Choose a Person" and "Choose an ID". Both are a listbox, you will find the symbol in the Controls Toolbar at the second row on the most left place. The Name of the Listbox under "Choose a Name" is simply "listbox_ctl", that is by default given by the IDE. The Name of the Listbox under "Choose an ID" cannot be the same, so I renamed that one to "idBox_ctl". I also changed the name of the two edit fields at right. The one after "ID" will contain the ID of the chosen record, so I changed the Name of that edit field into "id_ctl". The other edit field will contain the name from the record, so I changed the Name of it into "name_ctl". Both edit fields are only to check the data in the record_to_be_deleted. The user should not be able to change the contents. So I changed the property "ReadOnly" of both from False into True. You will find this property in the Properties List. Please complete the dialog, save it and build the project.

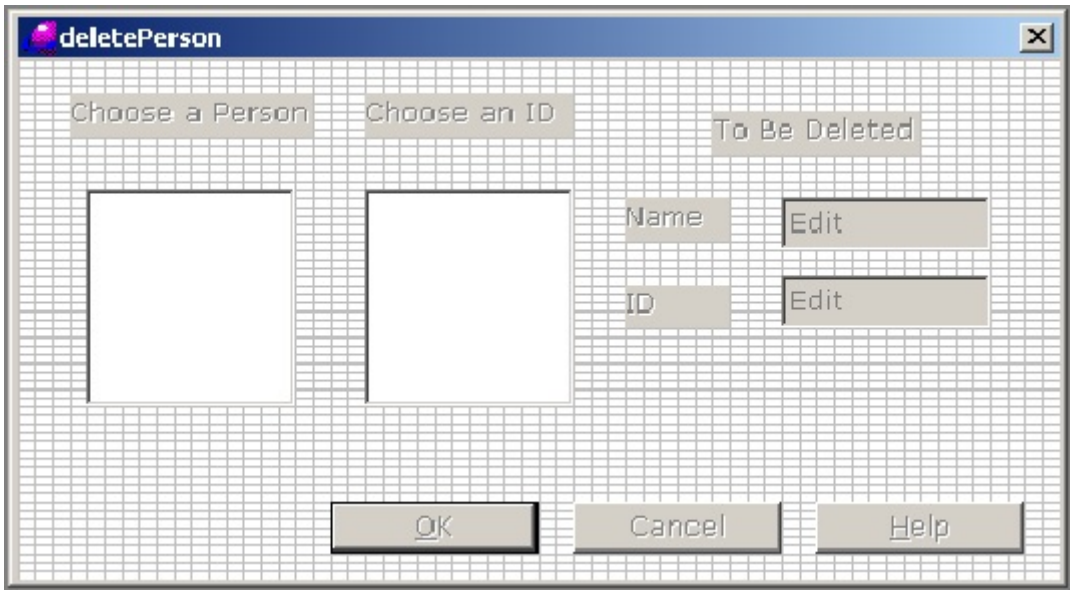


Figure 7.7 Dialog to delete a person

Now we will take care of the menu item that triggers this form. I added a menu-item in the Task Menu right under the option "Add Person". I gave it not surprisingly the name "Delete Person". Clicking this menu item should trigger the appropriate dialog. So please go to TaskWindow.win, right click and open the Code Expert. Click Menu, click TaskMenu, click id_database and <Add> the standard code for the new menu item id_delete_person. Then go to the code and change the standard code into:

```
predicates
   onDatabaseDeletePerson : window::menuItemListener.
clauses
   onDatabaseDeletePerson(Source, _MenuTag) :-
      NewDialog = deletePerson::new(Source),
      NewDialog:show().
```

When you now execute the program the form should appear when you click Database/DeletePerson. But nothing can be done yet with this dialog, because we didn't add functionality.

The first thing we should take care of is to fill the listbox with the names of the people in the database. Remember that we want to keep all the predicates that do something with the database in class (module) person. To fill the listbox so the user can choose a name, we need a list of all the names. So we make a predicate like this:

```
   getAllNames(NameList) :-
      findAll(Name, person(_,Name,_,_,_), NameList).
```

I put it in the file person.pro, right after the predicate addPerson(), but before "end implement person" When we call this predicate, the standard predicate "findall" creates the wanted list. We use it in the same way as in the preceding section. The variable Name is used to collect the strings that appear as the second argument in the functor "person" and these Names are collected in the list NameList. This is the clause for this predicate, the declaration ...

```
   getAllNames : (string* NameList) procedure (o).
```

... must be placed in the file "person.cl" where the other declarations reside. The asterix means that in this case we use a list. More on lists in chapter 10.

Now we are ready to use this predicate to fill the listbox in dialog deletePerson. Dialog deletePerson is created by calling the predicate new() in "deletePerson.pro". Please look for that file (you may have to build the project first) and open it in an editor. Then find the following clauses:

```
clauses
   new(Parent) :-
      dialog::new(Parent),
      generatedInitialize().
```

This is the code that is used to create the dialog. We will extend the code so that at creation time the listbox will be filled with names. Change the existing code into:

```
clauses
   new(Parent) :-
      dialog::new(Parent),
      generatedInitialize(),
      person::getAllNames(NameList),
      SingletonList = removeDuplicates(NameList),
```

```
    listBox_ctl:addList(SingletonList).
```

Take care to change the dot after

```
    generatedInitialize()
```

into a comma!

The three lines of code perform the following

- person::getAllNames(NameList),
  Calls the predicate that generates a list of all names in the database
- SingletonList = removeDuplicates(NameList),
  removeDuplicates/1 removes the duplicate names in the list. The cleaned list is bound to the variable SingletonList
- listBox_ctl:addList(SingletonList).
  This fills the listbox with the names in the list. Actually it adds the names in the list to the names already in the listbox, but as that is empty, we only see the names in the list.

The predicate removeDuplicates/1 is a standard predicate. It resides in the class “list”, so we must add it to the project. Go to deletePerson.pro and add at the begin of the code:

```
implement deletePerson
   inherits dialog
   open core, vpiDomains, list
```

Actually you only have to add the word “list” after vpiDomains. Mind the comma! And don’t forget to fill the database before you try the program, otherwise you will see nothing, that is you see an empty list.

The listbox should be filled with the names in your database. You can select any of them, but of course nothing happens. We didn’t implement the code yet. Maybe it is a good idea to try and execute the program.

What we want to happen is that when you select a name in the listbox, the program searches the database for records with that name and shows all the ID’s that it finds with the selected name. These actions should be triggered when you select a name. Luckily there is a standard predicate that is helpful here. It is one of the predicates that come with the listbox. Open the “deletePerson.dlg” in the Dialog Editor and click on the listbox where the names should appear, to give it the focus. Then open the Events list in the properties window. In the Events list select “SelectionChangedListener” and give it the value “onListboxSelectionChanged”. Then double click on it, the standard code is inserted and the editor is opened where the standard code is inserted. This code is executed when you change the selection in the listbox. Now please change the standard code into:

```
predicates
   onListboxSelectionChanged : listControl::selectionChangedListener.
clauses
   onListboxSelectionChanged(Source) :-
      [Name | _T] = Source:getSelectedItems(),
      person::getSomeIDs(Name, IDList),
```

```
    idBox_ctl:clearAll(),
    id_ctl:setText(" "),
    name_ctl:setText(" "),
    idBox_ctl:addList(IDList), !.
onListBoxSelectionChanged(_Source).
```

Don't forget to remove the underline before Source in the first clause, but do insert an underline in the second clause! Let me explain in plain words what is happening here.

[Name | _T] = Source:getSelectedItems(),
GetSelectedItems is a standard predicate that returns the selected items. Because VIP allows you to select more than one item in a listbox, the selected items are returned as a list. This list is bound to [Name | _T]. This last expression is a typical Prolog expression. It represents a list, but at the same time cuts the list in two parts. The first element is bound to the variable Name, the rest of the list is bound to the variable _T. The underscore is a warning for the compiler that we will not use this rest. Because in this case we allow the user only a choice of one element, it suffices to bind the first (and only) element of the list to Name.

person::getSomeIDs(Name, IDList),
We use Name in the calling of the predicate "getSomeIDs()". This predicate (that we yet have to make) will be placed in the class "person" and we will turn to the code later. For now we assume that we give Name as input and get back IDList as a list of IDs that are found with Name in the database.

idBox_ctl:clearAll(),
id_ctl:setText(" "),
name_ctl:setText(" "),
Because a new name was selected, we clear the idBox and the edit fields for ID and Name.

idBox_ctl:addList(IDList), !.
We now add the list of IDs to the idBox_ctl.

onListBoxSelectionChanged(_Source).
Don't think too much about this line. For the moment it is not too important.

Above I introduced the predicate "getSomeIDs()". It resides in the class "person" and it finds all the IDs that go with a given Name. It is straight forward:

```
getSomeIDs(Name,IDList) :-
    findall(ID, person(ID,Name,_,_,_), IDList).
```

I put it in "person.pro" right after "getAllNames. This predicate call shows you another use of "findall/3". Here we call "findall/3" and tell it (in the first argument) to find values for ID. In the third argument we tell findall/3 to put the values found in a list called IDList. But at the same time we tell findall/3 in the second argument that it should only take into account the records with the value Name as second argument. Remember that in Name we store the name selected in the listbox. So findall restricts the search to the records with the specified Name. Finally we have to declare getSomeIDs() in "person.cl". This is the declaration:

getSomeIDs : (string Name, string* IDList) procedure (i,o) .

Put it in “person.cl” right after the declaration of “getAllNames()”. When you now build and execute the program, it shows the ID’s when you select a name. That is to say, only when you first filled the database.

But still the fields with name and ID of the record_to_be_deleted are not shown. This should be done after you select an ID. Because this looks much like what happens when you choose a name, I give the code without explanation.

Go to “deletePerson.dlg” in the Project Tree and open it in the Forms editor. Select idBox_ctl (click once on it) and open the events list in the properties window. Select the event “SelectionChangedListener and give it the value “onIDBoxSelectionChanged”. Add the standard code (double click) and open the editor. In the editor change the standard code into

```
predicates
   onIdBoxSelectionChanged : listControl::selectionChangedListener.
clauses
   onIdBoxSelectionChanged(Source) :-
      [ID | _T] = Source:getSelectedItems(),
      person::getPerson(ID, Name, _, _, _),
      id_ctl:setText(ID),
      name_ctl:setText(Name), !.
 onIdboxSelectionChanged(_Source).
```

When the selection in idBox is changed, we get the chosen ID and then ask the module “person” to return the data for the person with this ID. The module “person” returns the wanted value for Name and we use it to fill the Edit Field. To make this work we need to add the predicate “getPerson()” to “person.pro”. Please go there and enter after the code for “getSomeIDs()”the following code:

```
getPerson(ID, Name, Weight, Length, IQ) :-
   person(ID, Name, Weight, Length, IQ), !.
getPerson(_ID, "0", 0, 0, 0).
```

The second clause is probably never used, but the compiler likes it to be there. Finally enter the declaration for “getPerson(ID, Name, Weight, Length, IQ)” in “person.cl”:

```
getPerson : (string ID, string Name, integer Weight, integer Length, integer IQ) procedure (i,o,o,o,o) .
```

Now we are almost done. We only have to insert the code to delete the record when the user has selected a record and then hits <OK>. To insert the code open “deletePerson.dlg” in the editor, select the <OK> button and open the Events list in the properties window. Select ClickResponder and give it the value “onOkClick”. Add the standard code, in the editor go to the code and change it into:

```
predicates
   onOkClick : button::clickResponder.
clauses
```

```
onOkClick(_Source) = button::defaultAction() :-
    ID = id_ctl:getText(),
    person::deletePerson(ID).
```

What we do is that when the user clicks <OK> we get the ID from the ID Edit Field and tell “person” to execute the predicate “deletePerson”. This predicate should delete the indicated record. Let’s see how it works. In “person.pro” add the following code just before “end implementation”

```
deletePerson(ID) :-
    retract(person(ID, Name, _, _, _)), !,
    stdIO::write("Person ", Name, " with ID: ", ID, " has been deleted"), stdIO::nl.
deletePerson(_).
```

This predicates tells the program to retract the record with the indicated ID. To retract a record means to discard it. To make sure, the Name in the retracted record is retrieved and shown to the user. Finally enter the declaration into the file “person.cl”. It is:

```
    deletePerson : (string ID) procedure (i) .
```

Now save, build and execute the program. Fill the database, add some persons (preferably with the same name or with existing names) and delete persons. Check the results now and then with “List Database”. Don’t bother too much about the possibility that the user clicks <Cancel>. We didn’t insert code for that option, so when the user hits that button, nothing happens. That’s exactly what the user wants. VIP simply closes the Dialog. IF you want, you can insert a note that says that no record is deleted. Here is the code and where to place it. Bring “deletePerson.dlg” into the editor and highlight the <Cancel> button. In the Events list go to the event clickresponder en give it the value “onCancelClick”. Double click on the value and change the generated code into:

```
predicates
    onCancelClick : button::clickResponder.
clauses
    onCancelClick(_Source) = button::defaultAction :-
        vpiCommonDialogs::note("No deletions").
```

Also there is the bug that when you want to delete a record from an empty database, the program shows simply an empty listbox. You may want to change the code for “getAllNames” to show a message. Here’s how:

```
    getAllNames(NameList) :-
        findAll(Name, person(_,Name,_,_,_), NameList),
        NameList <> [], !.
    getAllNames([]) :-
        vpiCommonDialogs::error("database is empty").
```

Note again the exclamation mark.

7.5 Manipulate the data: change the contents of a record.

To change the contents of a record in fact you do a number of (by now) familiar things.

- You retrieve the contents of the wanted record
- You show the contents to the user
- The user changes the contents
- You discard the old record
- You insert a record with the new values.

Concerning these last two steps: what to do when the contents are not changed by the user? Well, I'm a lazy programmer, I don't care. When nothing has changed, I still remove the old record and insert the new (but unchanged) values.

The steps we take in this section are very much like the ones we took in the previous sections. So I will give the necessary code without much explanation. First create a dialog. I made it look like the one in figure 7.8. It has the same listboxes as the dialog for deleting a person and I gave them the same names listbox_ctl and idBox_ctl. To the right you see five Edit Fields. These contain the contents of the record. The Field after ID is grayed out. We don't want the user to change the ID as it is used for unique identification. So I set the property "read-only" of the ID field to True. Besides I changed the names of the five edit fields. Not surprisingly I called them id_ctl, name_ctl, length_ctl, weight_ctl and iq_ctl respectively. Now please go ahead and create this dialog. I named it "changePerson".

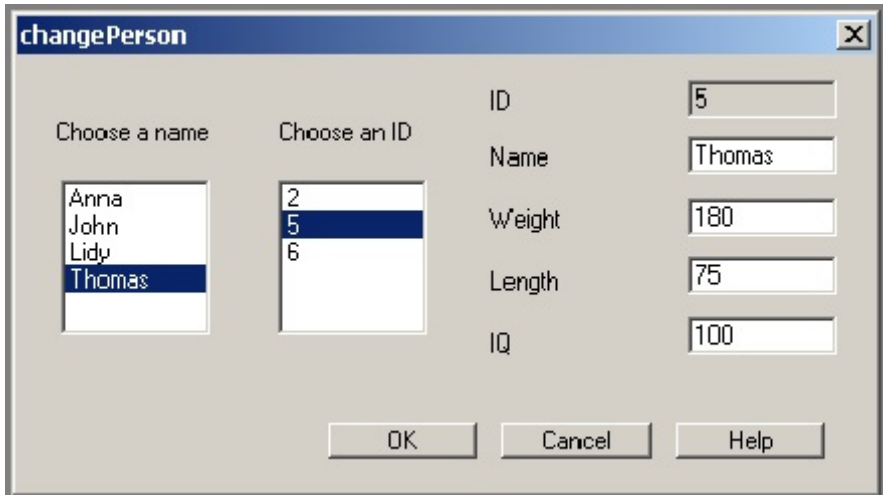


Figure 7.8 Dialog for change person

In figure 7.8 you see that the dialog is filled with the names from the database. I filled the database with the four records from section 7.1 and then added two records with the name Thomas. In the dialog I selected the name Thomas, the program responded with showing the three ID's, I selected ID=5 and the program shows the contents of the record with ID=5.

The code for this dialog will be quite familiar if you understood the previous section. First, when the dialog is generated, we need to fill the listbox with names. Build the project, go to the file "changePerson.pro" find the code:

```
clauses
   new(Parent) :-
      dialog::new(Parent),
      generatedInitialize().
```

Change it into:

```
clauses
   new(Parent) :-
      dialog::new(Parent),
      generatedInitialize(),
      person::getAllNames(NameList),
      SingletonList = removeDuplicates(NameList),
      listBox_ctl:addList(SingletonList).
```

Then go to the top of the file and change

```
   open core, vpiDomains
```

into

```
   open core, vpiDomains, list
```

We need code for the triggers SelectionChanged for the listbox_ctl (with names) and the idbox_ctl with the ID's. Please insert the standard code for both (refer to the previous section if you forgot how to do that) and then change the standard code into :

```
predicates
   onListboxSelectionChanged : listControl::selectionChangedListener.
clauses
   onListboxSelectionChanged(Source) :-
      [Name | _T] = Source:getSelectedItems(),
      person::getSomeIDs(Name, IDList),
      idBox_ctl:clearAll(),
      id_ctl:setText(" "),
      name_ctl:setText(" "),
      weight_ctl:setText(" "),
      length_ctl:setText(" "),
      iq_ctl:setText(" "),
      idBox_ctl:addList(IDList), !.
   onListBoxSelectionChanged(_Source).

predicates
   onIdBoxSelectionChanged : listControl::selectionChangedListener.
clauses
   onIdboxSelectionChanged(Source) :-
      [ID | _T] = Source:getSelectedItems(),
      person::getPerson(ID, Name,Length,Weight,IQ),
```

```
        id_ctl:setText(ID),
        name_ctl:setText(Name),
        length_ctl:setText(toString(Length)),
        weight_ctl:setText(toString(Weight)),
        iq_ctl:setText(toString(IQ)), !.
    onIdboxSelectionChanged(_Source).
```

Then insert the standard code for the <OK>-button and change in into:

```
predicates
    onOkClick : button::clickResponder.
clauses
    onOkClick(_Source) = button::defaultAction() :-
        ID = id_ctl:getText(),
        Name = name_ctl:gettext(),
        Weight = toTerm(weight_ctl:gettext()),
        Length = toTerm(length_ctl:gettext()),
        IQ = toTerm(iq_ctl:gettext()),
        person::changePerson(ID, Name, Weight, Length, IQ).
```

The only new predicate that we need in “person.pro” is “changePerson()”. Here it is.

```
changePerson(ID, Name, Weight, Length, IQ) :-
    retract(person(ID, _, _, _, _)),
    assertz(person(ID, Name, Weight, Length, IQ)), !,
    stdIO::write("Person ", Name, " with ID: ", ID, " has been changed"), stdIO::nl.
changePerson(_, _, _, _, _).
```

It simply removes the old record and inserts the a new one with the content that comes from the edit fields in the dialog. Because I use here “assertz()”, the new record is placed at the end of the database. The declaration for this predicate should be entered in “person.cl”. Please insert:

```
changePerson : (string ID, string Name, integer Weight, integer Length, integer IQ) procedure (i, i, i, i, i) .
```

Finally enter a new item in the task menu. I called it “Change Person” and put it between “Add Person” and “Delete Person”. Then in the Code Expert insert the standard code and then change it into:

```
predicates
    onDatabaseChangePerson : window::menuItemListener.
clauses
    onDatabaseChangePerson(Source, _MenuTag) :-
        NewDialog = changePerson::new(Source),
        NewDialog:show().
```

Don’t forget to delete the underscore in front of Source! Now you are ready to change a record. Please, go ahead and try it.

7.6 Saving and consulting the database

So far so good. You can add records, change their content and delete them. But when you exit the program, the database is lost. This is the fate of every internal database that is kept in facts in a program. The reason to use an internal database is that it is faster than a database on disk. In practice it is of course unacceptable that you lose data. So let's add two more predicates: one to save the database to a file on disk and another one to retrieve the saved data. Luckily there are standard predicates for these actions. You can find them in the class "file".

The standard predicate to save an internal facts database to disk is "save/2". It takes two predicates, a filename to write the data to and a name for the internal facts database. Until now we haven't given the internal facts database a name. That wasn't necessary because there was only one database in the class person. But now that we are going to use the predicate save/2, we first must give the facts database an name. I chose the name "personDB". To give that name to the facts database, change the line

```
        class facts
```

in "person.pro" into

```
        class facts - personDB
```

Don't forget the "-" in front of the name. You don't have to add a period at the end of this line

Now that the facts database has a name, we can implement the predicates for saving and retrieving. First add two new menu-items to the TaskMenu. I suggest that you give the new options the names "SaveDatabase" and "ConsultDatabase" and place them within the option "Database". Then add standard code for both options in TaskWindow.win and change the standard code into:

```
predicates
   onDatabaseSavedatabase : window::menuItemListener.
clauses
   onDatabaseSavedatabase(_Source, _MenuTag) :-
      person::saveDatabase().

predicates
   onDatabaseConsultdatabase : window::menuItemListener.
clauses
   onDatabaseConsultdatabase(_Source, _MenuTag) :-
      person::consultDatabase().
```

As you can see I keep the predicates for manipulating the data in the facts database in the class person. So now go to the file "person.pro" and add the code for the two new predicates. Here it is.

```
   saveDatabase() :-
      file::existFile("personDB.txt"), !,
      file::delete("personDB.txt"),
      file::save("personDB.txt", personDB),
      stdIO::write("The database has been saved to file personDB.txt"), stdIO::nl.
   saveDatabase() :-
```

```
    file::save("personDB.txt", personDB),
    stdIO::write("The database has been saved to file personDB.txt"), stdIO::nl.

  consultDatabase() :-
    file::consult("personDB.txt", personDB),
    stdIO::write("Records have been added from file personDB.txt"), stdIO::nl.
```

In both predicates the central clause is to call the standard predicate from the class “file” and to give it the arguments it needs. The first argument is the external filename that will be used to write the data to or to read from, the second argument is the name of the internal facts database in the program. In both cases we also write a message to the Message Window. The clauses for the consultDatabase/0 predicate are fairly straightforward, the use of saveDatabase/0 is a little bit more complicated. That is because save/2 adds the records to the contents of the file when it already exists. When you take no precautions this will result in the same record appearing more then once when you save more then once. So in the first clause of saveDatabase/0 we look if the file already exists. This is done by the standard predicate existFile/1. If the named file exists, this predicate succeeds, otherwise it fails. If the file exists, we use delete/1 to delete the existing file and then save/2 creates the file a new and adds the records. If the file doesn’t exist, the predicate existFile/1 fails and the engine goes to the second clause. There it simply writes the data to a newly created file “personDB.txt”.

Finally add two declaration lines to “person.cl”. Here they are:

```
  saveDatabase : () procedure .
  consultDatabase : () procedure .
```

Now the program is capable of saving and retrieving data from an external file on disk. Take care that this implementation of saving and retrieving is error prone. You must first save data to disk before you can consult the file - otherwise the program will crash. Also the name of the file on disk is preset, the data are always written to “personDB.txt”. You will find this file in the same folder as the program “ch07db.exe”. It is the folder with name “Exe” in the project folder “ch07db”. Another nuisance is that when you “consult” the data on the external file, they are added to the existing data in the internal file. So consulting two times in a row results in two times the number of records. If you want to avoid this, you must add a clause that deletes all records in the internal facts database before consulting. The best way to do that is to ask the user if she wants to keep or to delete the existing data before adding from the file. When you replace the clauses for consultDatabase/0 with the following code, the program will behave better. In the same code I repaired the bug that the program crashed when there was no file named “personDB.txt”. To repair I used the predicate existFile/1 that resides in the module (class) “file”. To be able to reach that module, change in “person.pro” the clauses

```
      implement person
        open core, list
into
      implement person
        open core, list, file
```

The other changes are shown below.

```
class predicates
  getData : () procedure .
```

```
clauses
    consultDatabase() :-
        file::existFile("personDB.txt"), !,
        getData().
    consultDatabase() :-
        vpiCommonDialogs::error("Sorry, can't do what you want", "No file called personDB.txt ").

    getData() :-
        Answer = vpiCommonDialogs::ask("What do you want me to do?",
                                       "Do you want to delete the existing records \n before consulting?",
                                       ["Yes", "No"] ),
        Answer = 0, !,
        retractFactDB(personDB),
        file::consult("personDB.txt", personDB),
        stdIO::write("Existing records have been deleted \n
                      New records have been added from file personDB.txt"), stdIO::nl.
    getData() :-
        file::consult("personDB.txt", personDB),
        stdIO::write("New records have been added from file personDB.txt"), stdIO::nl.
```

The database is now more or less complete. But take care! There is hardly any security in this program. E.g. When you exit the program, it should ask if it must save the data base. Also the user is free to do what he wants, there is no authorization et cetera. But the program works like a database.

7.7 Conclusion

In this chapter we created a database program, or better, a datafile program. You learned to use a listbox and a few predicates that come with it. You learned to work with lists and how to assert and retract facts in a database. Quite a job for one chapter. Congratulations!

By the way, in VIP there is a library of predicates to manipulate a database. Take a look in the VIP Wikipedia at http://wiki.visual-prolog.com and locate the article on “Visual Prolog External Database System”. For a short introduction take a look at chapter 20 in “Prolog for Tyros” by Eduardo Costa.

In this program we used classes as modules. In fact we neglected an important aspect of Visual Prolog, namely that it is an Object Oriented Programming (OOP) language. You could have done the same in a non-object oriented language. Now it is time to explore the OOP character of VIP. So up to the next chapter on Object Oriented Programming.

Chapter 8. Object oriented programming - classes and objects.

Visual Prolog is an object oriented programming language. When you are going to program in Prolog, you must know about the principles behind object oriented programming. The first thing to know is that object oriented programming is shortened to OOP. The other things to know are explained in this chapter. Of course I cannot explain everything that is to know about OOP in a single chapter in a introductory text. But the basics that you need are here, I think.

8.1 An OO-view at the world

Everywhere in and outside science people use models to describe the world around them. Models are very diverse, there seems to be hardly any restriction to how people describe the world. In that view, OOP is just another way of modeling. But it has turned out to be a very powerful way of modeling. Euclid used 5 axioms to create geometry. Let me explain to you the fundamentals of OOP in 5 basic ideas.

1. The first basic idea behind OO modeling and programming is that the world consists of objects. An object can be anything, literally any thing. The only condition is that you must be able to identify an object, that is you must be able to distinguish an object from its environment. So a drop of rainwater on your window is an object, a drop of water in the sea is not an object. But you will understand that this is hardly a serious restriction.
2. The second basic idea is that an object has certain characteristics. Characteristics are called attributes. An object can have many attributes . Think of a human being as an object and how many attributes you can imagine. Length, Weight, IQ, ... you name it. An attribute is a characteristic, but it is also a point of measurement. You can link a value to an attribute. In this way you create facts. A fact is an Object-Attribute-Value tuple that is true in a certain place at a certain time.
3. The third basic idea is that objects can do something, they perform activities, also called tasks. So every object has, next to its attributes, some activities that it can perform. In OOP we call these activities operations or methods.
4. The fourth basic idea is that objects can be brought together in groups. Of course you can do that in many ways, but in OOP the most important way is to bring together those objects that share the same attributes and the same operations. The set of objects that have the same operations and the same attributes is called the class of those objects.
5. The fifth idea is that objects are related to each other. There are several kinds of relations, but for now I skip this.

What is the use of all this? Well, basically it is a way to cope with the complexity of programming computers. When you use a program, you want it to do some job, but at the same time you want it to interact with other programs, you want it to have help files, maybe you want it to check your spelling, to give you access to internet, and so on. At the same time you want to be free in choosing what the program should do at a certain moment. You don’t like programs that allow you only one option at a time. To create such a program is quite a job. OOP is a way to create programs that are built from, say, modules. Each module takes care of a part of the job and therefore is less complex than the total program.

Let me give an example. Imagine that you have a database system of students, teachers and courses. When you program this information system in, say, Ms Access, you create the necessary tables and the user interface. Then you write the code that is necessary to operate the system. When the user clicks an option in the interface, the program recognizes the option, gets the data from the tables and performs the task that the user indicated. Every option in the menu is handled by the same program. In an OOP environment we would use a class of students. For every living student the class would generate a different object that contains the data (the attributes and the values) of that particular student. Also there would be a class of teachers and a class of courses that generate the particular objects for teachers and courses. Now think of the classes as modules in the information system. Add two more modules: a user interface and a report generator. When a user wants to have a list of students, she tells the user interface what she wants. The user interface asks the report generator to produce the report. The report generator starts to create the output, it creates headers et cetera and then asks each student object to provide the data of that student. The student objects provide the student data, the report generator finishes the report and sends it to the user interface. This is more or less how an OO program works.

The advantage of this way of programming is that you split up the job in simple activities that are performed by different objects and that are relatively independent. The user interface only has to take care of what the user enters and to send the necessary message to the other parts of the program. The student objects only have to take care of the data of one student and to provide these when asked. This architecture may seem to be more complex than necessary, but there are advantages. Imagine that some day the user interface needs a change. Then you only have to change that module. Or imagine that extra attributes are added to the student class. Then it is enough to change the class of students. Only when you want to allow the user to use these new attributes, then you'll have to change the user interface and the report generator accordingly.

Another advantage is that a module only has to know how to ask for an operation by another module. It doesn't have to know how the other module does the job. When you ask an object how old it is, you don't need to know how the age is calculated by the object. So when you decide that calculating the age should be changed, you can simply change it within the objects in a class without changing other classes. This is known as *encapsulation* and it is one of the magic points of OOP.

In the example I used the words modules and classes interchangeable. In fact, that is not correct, but it won't harm you. To understand OOP you should think of your computer program as a simulation model of the real world. To understand that, we have to take a closer look at classes.

8.2 More on classes

Classes are the core concept of OOP. A class is more than a set of objects. To begin, a class is a declaration. When you create a class, you give a specification of the attributes and methods that the objects in the class will posses.

When you declare a class, there are no objects. The objects are created when necessary when you execute the program. The class generates its own new objects. Objects of a class are called *instances* of that class. Let us elaborate a little bit more on the example above. When you enter the data of a new student, your OO information system literally creates a new object that represents the new student. Every class in a OO program has a method to create an instance. This method (in Prolog it is a predicate) is called the constructor of the class. Most of the time the constructor is called

“new()”. In the examples in the previous chapters we have made classes. When creating a cass, there was a checkbox called “creates object”.See e.g. figure 7.2. When you don’t check that box, then the class is considered a module. When you do check that box, then the class is supposed to have a predicate new/0 that creates objects, even when you don’t define such a predicate. Of course it is not necessary to give the constructor the name new/0. You are free to use another name. But keep in mind that a constructor must always have the type “procedure”. More on that in chapter 9.

In the example the new object will contain the data of the new student. To store the data in the attributes of an object, each object has methods of which the name begins with *set...*. When a student object has attributes Name, Address, Place, then there are methods called setName(), setAddress() and setPlace(). When you want to retrieve those data, you will ask that object after its Name or Address. For retrieval there are methods that begin with *get...*. So in this case student objects will also have a method getName(), getAddress() and getPlace(). An object can contain many attributes and many get and set methods. These methods are sometimes called the getters and setters of a class. In Visual Prolog these getters and setters take the form of predicates. You have to define them yourself in the prolog code for the class.

Communication between objects is performed by calling methods. In the example above the user interface calls a method in the report generator that creates the report. The report generator calls a method in object student to get the wanted data. Objects from other classes that interact with a specific object have to know what methods are available. Methods that are available to other objects (that are callable from within other objects) are called *public*. Now to make the classes as independent as possible, you don’t want every method to be public. Because when you change a method, you could be forced to go to every other class that calls this method to change the call. So we prefer to keep methods out of sight as much as possible. This is called *loosely coupling* and it is an important principle in OOP. So there are methods that are not callable from outside an object. These methods, that are only callable from within the object itself are called *private* methods.

Summarizing: an OO program consists among others of classes. Classes create objects and objects perform activities because they get messages (procedure calls) from the user interface or from other objects.

Visual Prolog is an OO programming language. That means that when you program, you will have to think in classes and objects. Every operation is performed by a method in an object. When you use PIE, this is not the case. PIE resembles Prolog from before the OO era. That is why PIE is a good place to learn to understand the inference engine in Prolog because you are freed from the OO and GUI hullabaloo. But when you are going to write a program in VIP that will be used by other people, you will have to learn and apply OO thinking and GUI practices.

8.3 Classes and objects in Visual Prolog

To see how classes, objects and messages go in VIP, we shall create a project to generate objects of a class person. You will create new objects and delete objects. Please create a new project (I call it “ch08class”), go to the Project Tree and highlight the root directory ch08class. Then click *File/New in New Package* and create a new class. Highlight “Class” in the left pane. Then enter in the right pane in the Name field the name of the class (“person”) and check the radio button for New Package. Leave

In Namespace unchecked and check Create Objects (that is what we are going to do). When it looks like figure 8.1, hit <Create>.

The IDE creates three files and presents them to you in three different editors. The files are named “person.i”, “person.cl” and “person.pro”. In these three files the class “person” is specified. Everything you want to know about class “person” is in these files. You already are familiar with files like “person.pro”. In this file we put the Prolog code for the predicates. In the Prolog helpfile this file is called the implementation of the class person. The code for the predicates is called the definition of the predicates. In the file “person.cl” we give the declarations of the class. More on this in the rest of this chapter.

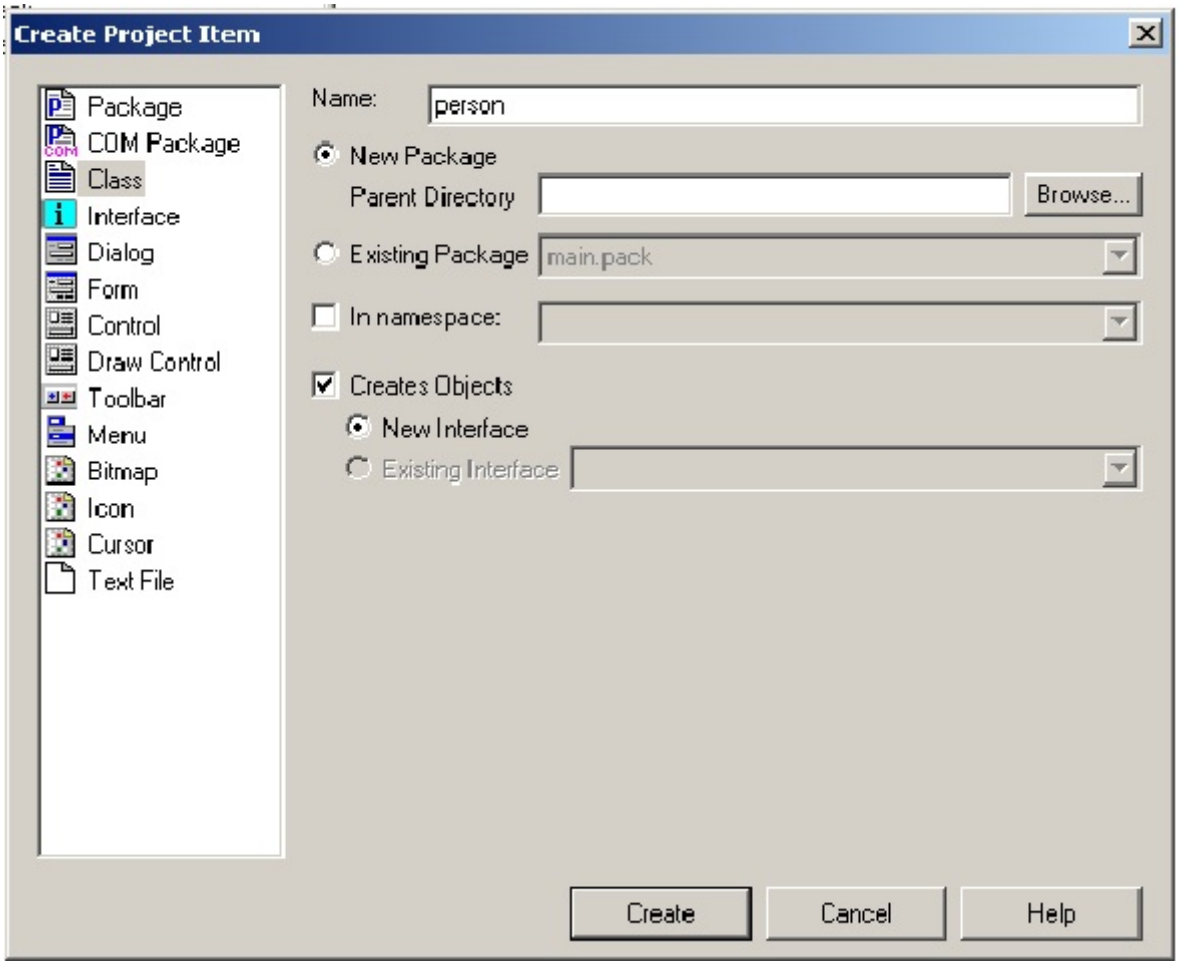


Figure 8.1 Create class “person

The third file, that you don’t know yet, is the file “person.i”. It is the so-called interface to the class person. The interface is like a desk where you can get information. In this case, the interface is what the outside world sees of the *objects* that are generated. The public *class* predicates are in “person.cl”. Now you can imagine what that means: in the files “person.cl” and “person.i” are the declarations of public methods. In Prolog methods are performed by predicates. Predicates that are declared here, are callable by other objects and classes. The IDE has already put some code in the file “person.i”. Here it is.

```
interface person
```

```
    open core

end interface person
```

As you can see, there are not yet any public predicates in the class person. Next to methods, the objects of class person will have attributes. Here I shall confine the attributes to a single one, the name of the object. As said in the previous section, when an object has an attribute, it can (and often will) have a get- and set-method for it that are callable publicly. So in the class person we want two predicates, getName and setName. When you call the first one, the object sends its name to you, with the second one you can enter (or change) the name of the object. So now change the contents of "person.i" into:

```
interface person
    open core

predicates
    getName: () -> string Name.
    setName : (string Name).

end interface person
```

You now have declared that every object that is generated by the class person will know two predicates that operate as procedures. Procedure getName needs no arguments and returns the name of the object as a string. The variable Name is used later on. The other procedure setName needs one argument of type string and sets the name of the object. This is all you (as an outsider) need to know of the object. You ask for the name and you get it. You give it a name and it is set. What is going on inside the object is none of your business as long as you get the name or the name is set. Of course you should realize that there are a lot of conventions behind this all. GetName and setName are familiar procedures in OO programming. When you create other procedures, be sure to make clear to the outside world what they do. By the way, you have declared the procedures as predicates. This is normal in Prolog, a procedure is a kind of predicate. In the next chapter I shall elaborate on these declarations. Also you should realize that you only have declared the predicates. What they do and how they do it, say the Prolog code for it, is another matter.

Now close the file "person.i" and take a look at "person.cl". It is the class declaration of the class person. In it is the following code:

```
class person : person
    open core

predicates
    classInfo : core::classInfo.
    % @short Class information  predicate.
    % @detail This predicate represents information predicate of this class.
    % @end

end class person
```

Don’t bother about the lines that start with a “%”-sign. This sign marks a comment. A comment is some insertion in a program that explains what is happening at that place. It is meant for human readers of the code, the compiler skips it. The “%”-sign indicates that the rest of a line is comment.

The first line in “person.cl” says

```
class person : person
```

In fact this is the declaration of the class “person”. It declares that there is a class with the name “person” (that’s what we wanted!) and that the type of this class is “person”. What does that mean? Well, think back of the what is in the interface “person.i”. In that interface you stated how to access objects of the class person. In VIP this is taken as a description of the type of the class. Being of type person tells the world that you can access objects generated by classes of this type via the interface person. In other words, in our program objects from the class person will have two callable procedures, getName and setName. Next to these two procedures there must be a constructor, because when you created the class, you checked the box “creates objects”. Objects are created by constructors. To declare the constructor, change the code in “person.cl” into:

```
class person : person
   open core

predicates
   classInfo : core::classInfo.
   % @short Class information  predicate.
   % @detail This predicate represents information predicate of this class.
   % @end

constructors
   new : (string Name).

end class person
```

In this class the predicate new/1 is the constructor. When you call this predicate it will create a new instance of person. To be able to do so, the constructor needs a name. You provide that name as an argument for new/1. The name of the new object is provided as a string.

Close and save the file “person.cl” and finally take a look at “person.pro”. This is the file where the code for the procedures will be placed. That’s why this file is called the implementation of the class person. Here you enter the code that is needed to make the objects behave as expected, that is the code for the predicates that were declared in person.i and person.cl. Please change the existing code into:

```
implement person
   open core

facts
   name : string.

constants
   className = "person/person".
```

```
    classVersion = "".

clauses
    classInfo(className, classVersion).

clauses
    new(Name) :-
        name := Name.

clauses
    getName() = name.

clauses
    setName(Name) :-
        name := Name.

end implement person
```

Let's take a closer look at the code. The first line of interest is

```
facts
    name : string.
```

This states that the objects of this class have a so-called fact of the type string. A fact is a variable, in this case you will recognize that this fact will contain the name of the object. But notice that the "variable" name begins with lower case. Facts like this one belong to an object, with every object the fact can (and will) have a different value.

Next you find the constructor.

```
clauses
    new(Name) :-
        name := Name.
```

The constructor new/1 was declared in "person.cl". Here you find the code for this predicate. It says that when new/1 is called, an object is cerated and that the fact "name" must get the value of the variable "Name" (notice upper and lower case!) that is implied in the call from outside. When you call the predicate new, you should give a name for the new object. Notice the assingment symbol ":=".
The expression

```
a := B
```

means in words: "assign to a the value of B".

The next clause is for the predicate getName. The code says:

```
clauses
    getName() = name.
```

When this predicate is called it should return the name that is contained in the fact name. And finally there is the predicate setName:

```
clauses
    setName(Name) :-
        name := Name.
```

When this predicate is called, internally in the object the value of the fact name is changed into whatever Name is provided by the call.

Now the picture is complete. We have declared a class person with one attribute that is put in the fact “name”. We have declared a constructor for this class and (in the interface “person.i”) two predicates to get and set the name. Finally we have put the necessary code into the implementation file “person.pro”.

Please take some time to digest the following. We did declare a class, but when your program runs, the activities are performed by objects. When you don’t generate an object, nothing will happen. Also, the fact name is a distinct fact in each object and it has a distinct value within each object that is created. Also you may have noticced that there are two places to declare public predicates. Predicates declared in person.cl are called class predicates. They focus on the class as a whole. The predicates declared in person.i are object predicates. They focus on individual objects. More on this later.

Now let us try to create objects. For that we need an event. Create a new option in the Task Menu and call it “CreateObjects”. Go to the Code expert and <Add> code for the new menu option. Then go to the code and enter the following

```
predicates
   oncreateobject : window::menuItemListener.
clauses
   oncreateobject(_Source, _MenuTag) :-
      NewPerson = person::new("John"),
      NameGiven = NewPerson:getName(),
      stdIO::write("New person with name ", Namegiven), stdIO::nl(),
      NewPerson:setName("Thomas"),
      stdIO::write("Name is changed. New name is ", NewPerson:getName()), stdIO::nl().
```

What happens when you click “CreateObjects”? First the variable NewPerson is bound to a new object. It is an instance of class person and it is given the name “John”. Remember that the constructor new(Name) is a predicate that is declared in the class. You call a predicate in a class by specifying the name of the class, followed by two colons followed by the name of the predicate and the arguments. In the next line the name of the created object is asked (by calling “getName()”) and bound to the variable NameGiven. This name is written to the message window. Not surprisingly it is “John”. Then we change the name of Newperson into “Thomas”. Again we ask the name with getName and now it turns out to be “Thomas”. Please notice that a predicate in an object is called by naming the object followed by a single colon followed by the name of the predicate and its arguments. This is a possible way to use the constructor and getName and setName, although it is a bit stupid to create only objects with the same name and to change the name immediately. As you will understand, this is only a simple example to show the use of the creator and the other predicates.

In a serious application you will probably ask the user for a name and then create the object. Let’s do that. Please create a dialog or a form like the one in figure 8.2.

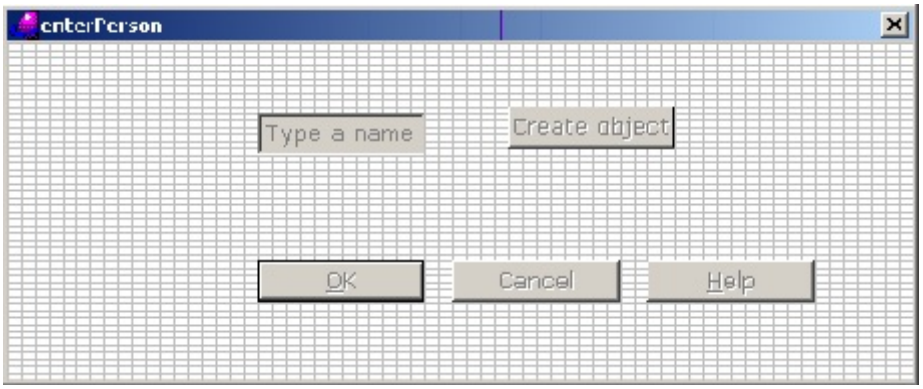


Figure 8.2 a form to enter a name and create an object

I named this form “createPerson”. Next to the familiar buttons it contains an edit field and a button with the label “Create Object”. The idea is that the user enters a name in the edit field and then clicks the CreateObject button and then the program creates an object with that name. To make the form work like that, you have to enter code for the CreateObject button. In the property list I changed the name of the button into createObject_ctl. The edit field I gave the name “name_ctl”. Do the same and close the form editor. Now reopen the form in the editor (this seems silly, but if you don’t, the code generator doesn’t seem to use the new name). Then generate the standard code for the clickresponder of the createObject_ctl. Go to the code and change it into:

```
predicates
    onCreateObjectClick : button::clickResponder.
clauses
    onCreateObjectclick(_Source) = button::defaultAction() :-
        Name = name_ctl:getText(),
        _NewPerson = person::new(Name).
```

To be able to use the form it needs to be displayed. Therefore change the code for the menu option CreateObject in the Taskmenu into:

```
predicates
    oncreateobject : window::menuItemListener.
clauses
    oncreateobject(Source, _MenuTag) :-
        NewForm = createPerson::new(Source),
        NewForm:show().
```

When the user chooses the option CreateObject, the form createPerson is opened. When the user clicks the CreateObject button, the variable Name is bound to the text in the edit field. “Name_ctl” is the name of the edit field (I gave it to the field), “name_ctl:getText()” returns the name that the user has typed in that field. The variable Name is then used to call the predicate “new” in class “person” and an object is created with whatever name the user entered. Unluckely you don’t see much of this new object. It is created, and that’s all. To see that it is really there, change the code in the clause for the constructor predicate new/1 in person.pro into:

```
clauses
    new(Name) :-
        name := Name,
        stdIO::write("Hello world, I'm a new object. My name is ", name),
```

```
        stdIO::nl.
```

Every time you create an object, it writes a greeting and it's name to the message window.

8.4 Classes and objects are different

Classes and objects are different things. The division of labor in an OO program dictates that a class creates objects and objects perform activities that are specified in the predicates. So clearly there are two groups of predicates: predicates that are performed by objects and predicates that are performed by the class. The first are called object predicates. They are focused on a specific object. The getName() predicate will only give the name of the object that it belongs to, it never gives the name of another object. The second group, the predicates that are performed by the class, are called class predicates. They are not focused on a particular object but on the class as a whole. Both object and class predicates are public predicates when they are declared in file *.i or *.cl.. The difference is that class predicates concern the class as a whole, they are focused on the set of all objects created by that class. Class predicates can also use facts that are not object attributes. These are called class facts and they are attributes of the class as a whole; they are shared among all objects of the class. In the syntax of VIP you recognize the difference also. To call a class predicate you write something like:

```
        <className> :: <predicateName>().
```

With the class you use a double colon. To call an object predicate you write:

```
        <objectName> : <predicateName>().
```

Here you use a single colon. In fact you did this from the beginning. Remember how in chapter 2 we created code to show a form. It said:

```
clauses
        onFileNew(Source, _MenuTag) :- NewForm= form1::new(Source), NewForm:show().
```

When you clicked the menu-option *File/New*, this predicate was called. It consists of two sub-calls. The first is:

```
        NewForm= form1::new(Source),
```

This says that we call the predicate "new()" in the class "form1". By now you know that this predicate is the constructor of the class. It does what it should do, it creates a new object from the class "form1" Not surprisingly that happens to be a form. This form (this object) is bound to the variable "Newform". From now on we can use this form object by using the variable as a reference. That is exactly what we are doing in the second sub-call:

```
         NewForm:show().
```

We refer to the object by the variable Newform and tell it to "show()" itself and bingo it is there on your screen. Please notice the different use of the colon and the double colon.

To illustrate the difference between class and object predicates, let us extend the example above with code that counts the number of person objects that are created. This count will increase every time an object is created and never decrease. I will extend the declaration of the class with a predicate (i.e. getCreatedCount) that can return the current count. So please now change the file person.cl into:

```
class person : person
   open core
```

```
predicates
   classInfo : core::classInfo.
   % @short Class information  predicate.
   % @detail This predicate represents information predicate of this class.
   % @end

constructors
   new : (string Name).

predicates
   getCreatedCount : () -> unsigned Count.

end class person
```

Why do we declare this predicate in the class declaration “person.cl”? Well, think of this. Each object that is generated, knows everything about itself. But it doesn’t know anything about other objects. Especially in this case, it doesn’t know if it is the latest object generated or even if there are other objects at all. But the class knows! Because the class generates the objects, it can count them and retain the number. You may think of a class as an umbrella covering the objects (or maybe as an mother duck covering little ducks). So when you want to know the number of objects created, you have to ask the class, not any object. So this is a class predicate and it **must** be declared in the class declaration “person.cl”. This is a strict rule with no exceptions. Public predicates are declared in both the interface and in the class declaration. The class predicates (performed by the class) are declared in the class declaration. The object predicates (performed by any of the objects) are declared in the interface declaration. It is not allowed to declare object predicates in a class declaration, and it is not allowed to declare class predicates in an interface.

You have declared “getCreatedCount()” as a public class predicate. Next it will need a specification in the implementation of the class and it will also need a fact for storing the count. This fact must be a ***class fact***, because it does not belong to each object, but it belongs to the class as a whole. It is shared among all the objects. Facts are declared in the implementation file “person.pro”. In the **implementation** of a class you can declare and define private object items (facts and predicates) as well as private class items. To declare class items you prefix the corresponding declaration section with the keyword “class”. All in all our class person can now be implemented like this:

```
implement person
   open core

class facts
   createdCount : unsigned := 0.

clauses
   getCreatedCount() = createdCount.

facts
   name : string.
```

```
constants
    className = "person/person".
    classVersion = "".

clauses
    classInfo(className, classVersion).

clauses
    new(Name) :-
        name := Name,
        createdCount := createdCount + 1,
        stdIO::write("Hello world, I'm a new object. My name is ", name),
        stdIO::nl,
        stdIO::write("I am object number ", createdCount),
        stdIO::nl.

clauses
    getName() = name.

clauses
    setName(Name) :-
        name := Name.

end implement person
```

I have added a class fact “createdCount”, which is initialized to zero. I hope you will take the precise syntax for granted, I will explain more in the next chapter. I have also added a clause for the predicate getCreatedCount, which returns the current value of createdCount. Finally, I have added the code in the constructor, which increments the createdCount.

Notice that in the constructor “new” there are two assignment statements:

```
        name := Name,
        createdCount := createdCount+1.
```

They have the same shape, but the first one assigns a value to the object fact “name”, the other one assigns a value to the class fact “createdCount”. In OO language this is called: one updates the object state, whereas the other updates the class state. A state of an object is the set of values that are assigned to its attributes. When one or more of the values change, the state is changed. The same goes for the class. When you create a new object, the value of “createdCount” changes and in OO language we say that the state of class person has changed. The other clauses in the constructor new/1 are to show you what is happening in the Message Window.

Now build and execute the program and enjoy the birth of new objects.

8.5 Classes and modules

You can think of a class or of objects as a set of predicates that perform tasks that are closely related. Sometimes you want to group some predicates but don't need a class with them that generates objects. In Prolog you can declare a class as non-object constructing. Such a class, that is like a module, is created with with File/New, option "class". In the Create Project Item Dialog you uncheck the "creates objects" check box. A class that doesn't create objects is described in two files. The interface file is not needed. There are no objects, so there are no object predicates publicly available, so there is no need for an interface file. As there is no interface file, this kind of class is not "typed". You can see that in the .cl file. The first line doesn't contain a colon and a type. Also these classes don't have a constructor. And finally, as there are no objects, these classes don't have object predicates or object facts. They only can have class predicates and class facts.

Let's extend the program with a class that contains code for output. Create the class "output". We are going to use this class only for generating output, it will not generate objects. So please uncheck the "creates objects" check box. Be sure to put the class in the same pack as "person". Then <Create> the class. The IDE now only presents two files, "output.cl" and "output.pro". The file "output.i" is not needed, for you declared that this class will not generate objects, so there is no need to declare objects predicates so there is no need for "output.i".

In the file "output.cl" enter this code (the IDE already has placed the standard code)

```
class output
   open core
predicates
   classInfo : core::classInfo.
   % @short Class information  predicate.
   % @detail This predicate represents information predicate of this class.
   % @end
   myWrite : (string ToWrite).
   myWrite : (unsigned ToWrite).
end class output
```

Save and close this file and in "output.pro" enter this code

```
implement output
   open core

constants
   className = "person/output".
   classVersion = "".

clauses
   classInfo(className, classVersion).

clauses
   myWrite(ThingToWrite) :-
```

```
    stdIO::write(ThingToWrite).

end implement output
```

As you can see in the class “output” I use the standard write-statement to implement the predicate myWrite(). In a serious program you would probably never do this (I mean create a class to do what is standard available). Also notice that in output.cl we declare the predicate myWrite/1 twice. This gives the possibility to use the predicate to print strings and to print unsigned integers. In OOP this is called polymorphism: the same predicate can be used in several ways.

Now go to the file person.pro and change the code for new/1 to use the new predicate myWrite/1.

```
clauses
  new(Name) :-
    name := Name,
    createdCount := createdCount + 1,
    output::myWrite("Hello world, I am a new object. My name is "),
    output::myWrite(getName()),
    output::myWrite(". I am person number "),
    Number = person::getCreatedCount(),
    output::myWrite(Number),
    stdIO::nl().
```

Now build and execute the program. It should perform as before.

All in all you have created a program that resembles a real object oriented system[9]. There is a user interface (createPerson.frm), there are objects that perform tasks (they tell their name and number) and there is a module that takes care of the output for the user.

What we did here is a general principle that is applied in OO programming. The information about the name is stored in the object. That is the place to get it, when you want to write the name, the object is the one that should provide it. So you ask the name with a call to “getName()”. The value of the number of objects is stored in a class fact, so the class is the one to ask the number of objects created (with a call to “person::getCreatedCount()” - note the double colon). By the way, of course you don’t have to use the code:

```
    output::myWrite(getName()),
```

You can use

```
    output::myWrite(Name),
```

but I thought it to be more elegant to use consistently the get-predicate to retrieve a value.

[9] Although the program doesn’t do much, it only generates objects.

8.6 Keeping track of the objects: a simple OO database

In the previous sections the programs created objects and that was it. In this section I like to extend this. We are going to use objects to store data and retrieve these data when we need it by asking the objects. This is in short the principle behind an object-oriented database.

By now you know how to create objects and how to assign data to their attributes. The next question is how to retrieve these data. For retrieving the data you need to retrieve the object. How to retrieve an object? There are many answers to this question. E.g. one could use the fact that when an object is created, it is bound to a variable. This variable acts as a pointer to the object. One way to find the object is to keep the bound variables in a list. When you need to know about an object, simply look in the list of variables and trace the object that you need and then ask your question about its data. In this section I shall use another method. I simply put the created objects in a list and when I need a certain object, I'll search this list till I find the wanted object. By the way, as I'm going to use a list, you may want to read the first section of chapter 9 on lists first.

Let's assume that we want a database of people and that we want to know the name and a number of everyone. The name can be chosen freely, the number will be computed by the program. To show you how this works I created a program, target GUI, with the name ch08classDB. It looks a little like the program ch08class, but I decided to make it another project and not to change ch08class. In the project I created a package "person" with in it a class person. Here I am going to store the data. The data are stored in an internal facts database in the class person. Please create the project and the package "person" and the class "person". Take care that the class "person" does create objects!

As we are going to make a database, we have to implement the basic functionality for a database. That is we must be able to create objects, show one or more objects and delete objects. So I decided to add four options to the TaskMenu:

- CreatePerson to create an object
- ShowAll to show the data of all persons
- ShowSingle to show the data of a specific person
- DeletePerson to delete a person.

Please add these options to the Taskmenu and have the Code Expert generate the standard code for each of them. We 'll come back to these menu items later on.

The data will be stored in the objects that are generated by the class "person". In the definition file "person.pro" I add two kinds of facts. There are the object facts "name" and 'number" that contain the data of each object. Next to these there are the class facts "createdCount" and "createdObjects". The class fact "createdCount" takes care of counting the number of objects that have been created. The class fact "createdObjects" is a list that contains the created objects. Here are the declarations that you should insert in "person.pro".

```
class facts
   createdCount : unsigned := 0.
   createdObjects : person* :=[] .

facts
   name : string.
```

```
number : unsigned :=0.
```

Pay attention to the class fact “createdObjects”. The declaration says that the type of it is a “list of person”, but how does the compiler know what type that is? The answer is shown in the class declaration. The first line in “person.cl” says

```
class person : person
```

It tells the compiler that this class “person” is of type “person”. This means that each object that is created in this class can and will react when you call one of the object predicates that are mentioned in the interface “person.i”. In this interface we declare in general the predicates among others to set and get the attributes of an object. In this case the object has two attributes (facts), name and number. The name is given by the user, so there need to be two predicates: setName() to set the attribute and getName() to retrieve it. The number will be calculated by the program, so for this attribute we only need a predicate getNumber(). These are the three predicates to declare in “person.i”. So please go there and add this code:

```
predicates
   getName : () -> string Name.
   setName : (string Name) procedure (i).
   getNumber : () -> unsigned Number .
```

Next we need to add the code for these predicates in “person.pro”. Here it is:

```
clauses
   getName() = name.

   setName(Name) :-
      name := Name.

   getNumber() = number.
```

To create an object, we need a so-called constructor. It is a class predicate that creates the objects, that is, it reserves some memory for the newly created object and generates a kind of pointer to the object. This pointer is bound to a variable and we can reach the object by way of this variable. For the construction of the objects I declared the constructor “new()”. As the constructor is by definition a class predicate (the class creates objects, objects cannot directly create objects) I put the declaration in the file “person.cl”. The declaration looks like this:

```
constructors
   new : (string Name).
```

The constructor “new()” has one argument, the name of the new object. We shall ask the user to provide the name. Thereto I created a form with the name “createPerson.frm”. It looks like in figure 8.3. It is quite simple. It contains some text, an edit field “name_ctl” and two buttons. The button with the word “create” on it, is called “createButton_ctl”. It is newly created and it is clicked to create an object. The other button with the word “Close” is called “close_ctl”. It is the standard button <OK> that the IDE provides when a form is newly created. By using this former <OK>-button, I produce the effect that when this button is clicked, the form closes. In this way I don’t need to give code for this button. I discarded the other buttons <Cancel> and <Help> as I don’t use them.

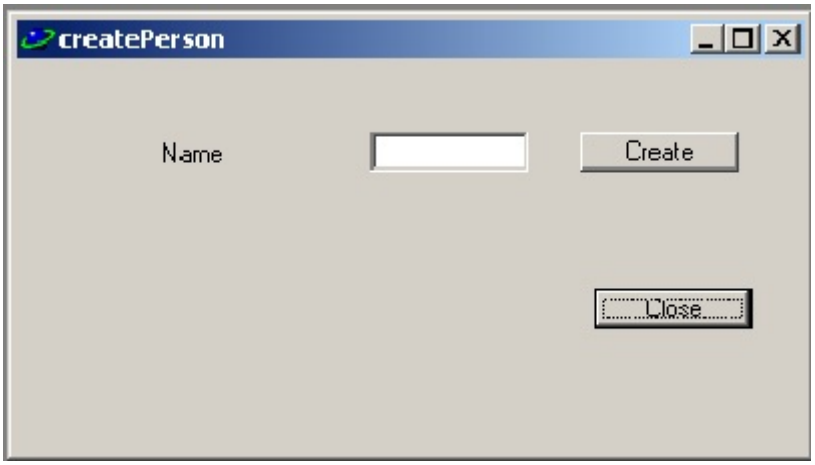


Figure 8.3 form to create a person

So there remains only to design some code for the event that the user clicks the <Create>button. In the form editor highlight the <Create>-button and in the properties window choose the Events tab. Go to the option ClickResponder and choose the value “onCreateButtonClick”. Double click the opton and in createperson.pro add the code

```
        predicates
            onCreateButtonClick : button::clickResponder.
        clauses
            onCreateButtonClick(_Source) = button::defaultAction :-
                Name = name_ctl:getText(),
                _NewPerson = person::new(Name).
```

When the user clicks the <Create>-button, the constructor predicate “new()” of class person is called. This predicate should reside in “person.pro” and it looks like this:

```
clauses
   new(Name) :-
      name := Name,
      createdCount := createdCount + 1,
      stdIO::write("Hello World, I am a new object person. My name is: ", getName()),
      number := getCreatedCount(),
      stdIO::write(". My number is: ", number),
      addToObjectsList(This),
      stdIO::nl.
```

The first line gives the Name provided by the user to the attribute ”name”. Then the counter “createdCount” is raised by one, and a message is shown. In the next line the attribute “number” is set to the value of the (raised) “createdCount”. In this way every object will have an unique number. The predicate getcreatedcount/0 needs a declaration and definition. The declaration is inserted in “person.cl” and looks like:

```
        predicates
                getCreatedCount : () -> unsigned Count.
```

The definition is inserted in “person.pro”and goes like this:

```
        clauses
            getCreatedCount() = createdCount.
```

After the number is written, there is a call to the predicate

```
addToObjectsList(This)
```

This predicates is a private class predicate. It can only be called from within the class, so it must be declared within “person.pro”. Here are the declaration and the clauses

```
predicates
   addToObjectsList : (person NewPerson) procedure (i).
clauses
   addToObjectsList(NewPerson) :-
      createdObjects := [NewPerson | createdObjects].
```

When we call this predicate, we use the argument “This”. This is a special variable. “This” can be used in predicates. It refers to the object that is “in control” at that moment. So when we ask an object for its name, “This” refers to the object that is asked to give its name. In this case “This” refers to the object that is created by the constructor. The code puts the newly created object at the head of the list of created objects.

The only thing left to do is to add code to the option in the TaskMenu. Use the Code Expert to reach the standard code for “CreatePerson” and change it into:

```
predicates
   oncreateperson : window::menuItemListener.
clauses
   oncreateperson(Source, _MenuTag) :-
      NewForm = createperson::new(Source),
      NewForm:show().
```

This completes the code for creating person objects. It is a good moment to build and execute the project and to generate some objects. If you want to make things a bit more interesting, you may add some more attributes.

Now that we can create objects, it is time to show them to the world. To achieve that I created a predicate showAllPersons/0 and placed it in class “person”. As this predicate is to work on all objects, it is a class predicate. Besides it is to be called from outside the class “person”, so it is a public class predicate. That’s why I declare it in “person.cl”. This is the declaration:

```
showAllPersons : () procedure.
```

Next I entered the definition in “person.pro”. Here is it.

```
showAllPersons() :-
   stdIO::write("These are the persons in the database \n"),
   showAllObjects(createdObjects).
```

This predicates announces that the objects will be printed and then calls another predicate showAllObjects/1 to do the job. ShowAllObjects/1 takes the list of objects and one by one takes the head off the list, asks the object for its name and number and then proceeds with the tail. This continues till the tail is empty. The predicate showAllObjects/1 is a private class predicate, so I declare it in the file “person.pro”.

```
class predicates
```

```
    showAllObjects : (person* CreatedObjects).
clauses
    showAllObjects( [Head | Tail]) :-
        stdIO::write("Name: ", Head:getName(), " with number: ", Head:getNumber() ),
        stdIO::nl, !,
        showAllObjects(Tail).
    showAllObjects( [ ] ).
```

Finally we have to attach the code to the taskmenu item “ShowAll”. So go to the standard code that was generated by the IDE in “TaskWindow.pro” and change it into:

```
predicates
    onshowall : window::menuItemListener.
clauses
    onshowall(_Source, _MenuTag) :-
        person::showAllPersons().
```

The program now will show a list of the objects in the Messages Window.

A list is nice, but in a database you often only retrieve data of a single object. So let’s now design code for the next taskmenu item “ShowSingle”. This option should show a single object. I decided to first show a list of the names to select an object and then to use the name to find the object and the other data, in this case the number. Let’s enter the code for this. I changed the standard code that was generated by the IDE for the Taskmenu option “ShowSingle” into:

```
predicates
    onshowsingle : window::menuItemListener.
clauses
    onshowsingle(_Source, _MenuTag) :-
        person::getAllNames(Nameslist),
        b_true = vpiCommonDialogs::listSelect("Select a name", NamesList, 0, Name, _Index), !,
        person::showSinglePerson(Name).
    onshowsingle(_Source, _MenuTag) :- !.
```

Then I declared the new predicates in “person.cl”.

```
  getAllNames : (string* Nameslist) procedure (o).
  showSinglePerson : (string Name) procedure (i).
```

The predicate getAllNames/1 produces a list of the names of the objects. This is the code. Please insert it in “person.pro”.

```
clauses
    getAllNames(NamesList) :-
        getNames(createdObjects, NamesList).

class predicates
    getNames : (person* CreatedObjects, string* NamesList) procedure (i,o).
clauses
    getNames(ObjectsList, NamesList) :-
        [Head | Tail] = ObjectsList, !,
        Name = Head:getName(),
```

```
        getNames(Tail, TailNamesList),
        NamesList = [Name | TailNamesList].
    getNames([], []) :- !.
```

The predicate getAllNames/1 uses a private class predicate “getNames/2" to produce the list of names. Just like it was done before, the list is given as an argument and getNames/2 one by one takes the head off the list and asks the object for its name. The gotten names are collected in a list and upon termination of the predicate returned to the calling predicate in “Taskmenu.pro”. The list of names is then offered to the user with the help of the standard predicate”listSelect” from the class “vpiCommonDialogs” that was introduced to you in chapter 3. The user chooses a name from the list and then the predicate showSinglePerson/1 in “person.pro” is called. It shows the data of the person wanted. Here is the code.

```
showSinglePerson(Name) :-
    getData(Name, createdObjects, Number), !,
    stdIO::write("You have selected: ", Name, " with number: ", Number),
    stdIO::nl.
showSinglePerson(_Name).

class predicates
  getData : (string Name, person* ObjectList, unsigned Number) determ (i,i,o).
clauses
  getData(Name, ObjectsList, Number) :-
    [Head | Tail] = ObjectsList,
    Name <> Head:getName(), !,
    getData(Name, Tail, Number).
  getData(_Name, ObjectList, Number) :-
    [Head | _ ] = ObjectList,
    Number = Head:getNumber().
```

The predicate showSinglePersson/1 gets the name and then calls another predicate getData/3 to retrieve the data. Predicate getdata/3 shows a pattern that should be familiar by now; It takes the list, retrieves the name of the first object in the list and compares it with the name given as argument. If the two names differ, getData/3 continues with the rest of the list. If the names are the same, then the second clause is invoked. Here we take the first object in the list (that is the one we were looking for - the names are equal) and we retrieve the other data, i.c. the number. Then showSinglePerson/1 writes these data to the Messages Window. Build and execute the program to verify that it really does what it says.

Finally we need code to delete a person. I assume that by now you are getting familiar with the program so I will give the code without much explanation. Change the standard code for the TaskMenu item “DeletePerson” into

```
predicates
  ondeleteperson : window::menuItemListener.
clauses
  ondeleteperson(_Source, _MenuTag) :-
    person::getAllNames(Nameslist),
    b_true = vpiCommonDialogs::listSelect("Select a name", NamesList, 0, Name, _Index), !,
```

```
        person::deletePerson(Name).
    ondeleteperson(_,_).
```

The pattern looks like the way we coded onShowSingle(). First we get the names, then show it to the user. She chooses a name and the object is removed from the list. Deletion is performed by the predicate deletePerson/1 in the class “person”. This predicate is a public class predicate so I declare it in “person.cl”

```
    deletePerson : (string Name) procedure (i).
```

The code for this predicate looks like this:

```
  deletePerson(Name) :-
    [Object | Tail] = createdObjects,
    Name = Object:getName(), !,
    createdObjects := Tail,
    stdIO::write("Person ", Name, " has been deleted \n").
  deletePerson(Name) :-
    [Object | Tail] = createdObjects,
    createdobjects := Tail,
    deleteperson(Name), !,
    createdObjects := [Object | createdObjects].
  deletePerson(_).
```

The pattern is like other predicates: you take the head of the list of objects, compare the name of it with the name chosen by the user and when they match you discard it. If the name is not the one that is given, then in the second clause we retain the object, continue with the tail of the list and upon returning from the that call, combine the retained object with the tail without the deleted object. Take a closer look at the way I use the list “createdObjects”. In the other predicates I gave this list as an argument. Here I use the fact that it is a fact database. Thaat means that I can use the list from anywhere in the file “person.pro”. Here is how this works. I take the full list createdObjects, peel the head off. If this object has the right name, then createdObjects gets the value of the tail. So the named object is deleted from the list. If the name is not the one wanted, then in the second clause I again peel off the head, but this time that object is retained in the variable Object. Then deleteperson/1 is recursively called, it uses again the list createdObjects, but that is now the list without the (previous) first element. The wanted object is deleted and then the list is reunited with its former first element. If this sounds or looks like magic to you, read more about lists in the next chapter.

The program can be built and executed now. It should work properly. But maybe you are worried about a detail. When we delete an object, we remove it from the list. But when it was created there was some memory reserved for it. We did not remove somehow that piece of memory. You might say that the object is still there, but we cannot reach it anymore. So next to the constructor, one would expect something like a destructor. But... there are no destructors in Visual Prolog. There is something else, it is called the garbage collector. This is a program that works almost like a demon. It checks the objects that are created. When an object cannot be referenced anymore, the garbage collector removes it from memory. So when we removed the reference to the object from the list, it was removed from memory by the garbage collector.

Chapter 9 Declarations in Visual Prolog[10]

In Visual Prolog, the program that you write is compiled before it is executed. Compilation is the translation of your Prolog code into machine code that is executed by the processor in your computer. Compilation is quite a job. It is done by the so-called compiler. In order that the compiler can do its work properly, it is necessary that you provide it with all the information it needs. This chapter is about that. In your program you provide the code. Part of the code is not about the "things to do" but is about what the code is. Is it a predicate, is it a clause, is it a fact? For every piece of code in your program you have to provide the compiler with information about it. This is done in the so-called declarations in the program. Or to say it in another way, your program will consist of several sections. Each section is preceded with a keyword that informs the compiler what it can expect in that section. This chapter is about these keywords, their meaning and the contents of the various sections in a program. But first we have to go back to basics. What are the building bricks of your program? After that we will see what keywords there are and what their meaning is and how you can/should use them.

In this chapter I have to be quite formal. That's the way computers work. To make the formal syntax better readable, I shall use now and then a kind of notation that places special words between hook brackets "<" and ">". A word between these brackets has a special meaning. I shall try to make these words easy understandable. So I shall use the word <className> to indicate the name of a class. This opens the possibility to describe the name of the file with the code of a class as <className>.pro. When I use a word between "<" and ">" I shall explain its meaning. But most of the time the meaning will be obvious I think.

I think that it is possible to skip this chapter at first reading. But I'm not sure about it and reading this chapter gives you a good feeling about the richness of Visual Prolog. So chew it!

9.1 Declarations and compilation

In classic Prolog, e.g. in PIE, when we use a predicate we simply do so without any prior intimation to the Prolog engine about our intention. For example; in the earlier chapters we used PIE and a program about a family. The clause for the grandFather predicate was directly written down using the traditional Prolog predicate head and body construction. We did not bother to inform the engine explicitly in the code that such a predicate clause was to be expected nor did we tell anything about what it would look like. Similarly, when a functor is to be used in traditional Prolog, we can use it without first forewarning the Prolog engine about our intention to use it. We simply use a functor, the moment we feel the need for it.

However, in Visual Prolog, we first need to declare the existence of a predicate to the compiler before writing down the code for the clause body of it. Similarly, before using any functors, they need to be declared and their presence is to be informed to the compiler. The reason a forewarning is required in Visual Prolog is essentially to ensure that runtime exceptions (errors, bugs) are converted into

[10]The text in this chapter is partly from the tutorials by Sabu Francis, partly from the VIP helpfile and partly from the Visual Prolog Language Tutorial that came by Visual Prolog Version 4.0. It describes only a part of the declarations that are possible - for beginners :-).

compile time errors as far as possible. By *"runtime exceptions",* we mean the errors that turn up when you run the program. For example, if you intend to use an integer as the argument of a functor, and instead of the integer you erroneously use a real number, things will go wrong when you run the program. We say that it is a *runtime* error and the program will fail there. This is the case in many other programming languages that have a so-called interpreter. In these languages the program is stored as a text and only at runtime the necessary program code is translated statement by statement into machine code by an interpreter and executed. In VIP there is a compiler instead of an interpreter. A compiler compiles the whole program into machine code before execution and at runtime the processor executes the machine code. So a compiler can check the code while it is compiling, before you run the program.

Because of this feature in Visual Prolog, the overall efficiency of a programmer improves. The programmer does not have to wait till the program is actually executed to detect a bug. In fact, those of you, who are experienced in programming, would understand what a life-saver this can be; for often, the particular sequence of events that is causing a runtime exception to happen at run-time may be so elusive that the bug may actually turn up after several years, or it may manifest itself in some crucially critical or other embarrassing situations!
All this automatically implies that the compiler **has** to be given explicit instructions regarding the predicates and other things that exist in the code. For this instructions there are two distinct concepts. In the *declaration* you describe in general what e.g. a predicate will look like and what kind of arguments it will use, in the *definition* (or implementation) you enter the specific clauses for that predicate. Please take care not to mix up these two notions. They are used throughout this chapter and the VIP documentation. Sometimes you can combine the two in one statement, as you can with domains and constants. Sometime you have to declare and define in different statements, as you do with predicates.

This chapter is about what to declare , how to declare it and where to put the declarations in your program. It is not complete. I will restrict the myself to the declarations that are used most and that are important to a beginning programmer.

9.2 Basic notions and overview of keywords

In programming a computer, you are using a kind of language. In this book the language is Prolog. To be able to talk about that language we need a number of basic concepts. This section is about these basic notions.

Identifier
An identifier is a name. It could be the name of a predicate, of a class, of a variable, of anything. An identifier must conform to some rules. There are four kinds of identifiers in VIP. They are called:

- lowercaseIdentifier
- uppercaseIdentifier
- anonymousIdentifier
- ellipsis

A lowercaseIdentifier is a sequence of letters, digits, and underscores that starts with a lowercase letter. An uppercaseIdentifier is a sequence of letters, digits, and underscores that starts either with a capital letter or with an underscore. The anonymousIdentifier is the underscore "_". You use the

anonymousIdentifier in places where a predicate needs an argument but you don't use this argument. E.g. when a functor (describing a person like in chapter 7) contains Name and Age and you only want to know the Name. In that case you can call the predicate with the anonymousIdentifier in the place of the Age. You met the anonymousIdentifier already in chapter 7 in the simple database. As this is a book for beginners I shall leave out the ellipsis.

Keyword
Keywords are special words that have a specific meaning. They are used by the compiler. You can only use a keyword in the way and with the meaning as it is intended by the makers of VIP. You cannot use a keyword in any other way, e.g. as a variable name. VIP knows the following keywords. The meaning of some of them will be explained in this chapter.

align and anyflow as
bitsize
class clauses constants constructors
div delegate domains determ digits do
else erroneous externally end
facts from failure foreach
goal guards
implement inherits interface if
language
multi mod monitor
nondeterm
or
open
predicates procedure
quot
reference rem resolve
supports single
then to

For now, don't bother too much about the meaning of these keywords, just remember never to create classes, predicates or variables that have one of these keywords as their name. All keywords except "as" and "language" are reserved words. Notice that "div", "mod", "rem" and "quot" are also reserved words, but these are not keywords but arithmetic operators. See below. The keywords "guards" and "monitor" are at the moment not used in the language, but are reserved for future usage.

Comment
A comment is a part of your program code that is skipped by the compiler. You use it to clarify parts of your program code for the human reader. A Visual Prolog comment is written in one of the following ways:

- The /* (a slash followed by an asterisk) characters, followed by any sequence of characters (including new lines) and terminated by the */ (asterisk, slash) characters. These comments can be multi-lined. They can also be nested.
- The % (percent sign) character, followed by any sequence of characters. Comments that begin with character % (percent sign) continue until the end of the line. Therefore, they are commonly called "single-line comments."

Examples of comments are:

```
/*This is a comment ...
   ... that continues on the next line */
person(Thomas, 61,190).          % the rest of this line is a comment
```

In the editor in VIP, the text in comments is colored blue.

Declaration
A declaration is an informative formal specification of the code that you are going to use in your programs. E.g. when you use a predicate, you first have to declare it. A predicate “square” that expects an integer as input argument and that returns the square of the number (so it is a function) is declared:

```
square : (integer Input) -> integer Answer procedure (i).
```

Statements like these are declarations. A declaration informs the compiler of the general form of e.g. predicates. They are used at compile time. As such they are not executed at run time. Declarations will be treated at length in this chapter.

Definition
A definition is the specification of the program code in your program. E.g. the definition of the predicate “square” could be:

```
square(Input) = Answer :-
        Answer = Input * Input
```

This clause defines (specifies) the declared predicate.

To separate the declaraton and the definition of a predicate, we precede the declaration with the word “predicates” and we precede the definition with the word “clauses”. The declaration must precede the definition. So for the predicate “square()” we get:

```
predicates
        square : (integer Input) -> integer Answer procedure (i).
clauses
        square(Input) = Answer :-
                Answer = Input * Input
```

The definition is executed at run time. At compile time the compiler uses the declaration to check the use of the predicate in the definition. In this case the compiler could find out that somewhere in your program you erroneously use a string as input when you call the predicate “square”.

Punctuation marks
Punctuation marks in Visual Prolog have syntactic and semantic meaning to the compiler, most of them do not specify by themselves an operation that yields a value. They are used to separate clauses, to end a declaration et cetera. Some punctuation marks, either alone or in combinations, can also be Visual Prolog operators. In VIP these punctuation marks are accepted by the compiler:

```
  ;  !  ,  .  #  [  ]  |  (  ) :-  :  ::
```

The meaning and use will become clear in the remainder of this chapter.

Operators
Operators specify an evaluation to be performed on involved operands. As an example take a look at

```
        1 + 2
```

The numbers 1 and 2 are called the operands, the “+” (“plus” in words) is the operator. The arithmetic operator “+” specifies that the number 1 and 2 should be added. In VIP these operators are known:

```
    +  -  /  * ^  = div mod quot rem <  >  <>  ><  <=  >=  :=
```

All operators are binary operators, that means that they need two operands. But - and + also exist as unary operators. That means that to the compiler “-3" is meaningful. Although these operators are quite familiar, there are slight differences between their use in Prolog and other languages.

- The operator “=” is not an assignment operator, it is a comparison operator. Essentially this operator compares the two operands. Its use depends on whether the operands are bound or not. If both operands are bound, then the comparison succeeds if both operands are the same, otherwise it fails. If one of the operands is bound and the other is not, then the unbound one is bound to the value of the bound one. If both are unbound, then they are bound to each other. If later on one gets a value, the other one will “automatically” get the same value.
- The operators “div” and “mod” are reserved words. The operators “div” and “quot” give as a result an integer division, but both do it slightly different. Please take a look in the help file to find out the difference. The operators “mod” and “rem” give the remainder with an integer division. Please read the helpfile on the difference.

Literals

Literals are to be taken literally. It means that they do not represent a variable or a predicate or whatever, they are just what they are. The number 12345 is just this number, nothing else. Literals fall into following categories: integer, character, floating-point, string, binary and list.

Integer literal

An integer literal (or integer for short) is a number that has no decimal part, it is an integral value. It may be preceded by the unary plus or minus sign. You can use octal integers in your program. They must be preceded by “0o” (zero and lowercase “o”). So 0o12 (octal 12) is decimal 10. Also hexadecimal numbers are possible. They must be preceded by “0x” (zero and lowercase “x”). So 0x22 (hexadecimal 22) is decimal 34. There are two standard domains for the integer literal. It can belong the domain “integer” or to the domain “unsigned”. Take care that an integer does not exceed the maximum and minimum values for the domain. See the next section on domains.

Real Literal

A real literal (or real for short) is a number that has a decimal part or at least a decimal dot, or that contains an exponential part. If a number has no decimal dot, a decimal part, or an exponential part, then VIP takes it to be a integer. There are two ways to write a real

- with a decimal dot or part, like 1234.56 or “543.”.
- with an exponent part, like 12345.6789E2. In this case the part “E2" denotes that the number 12345.6789 must be multiplied with 10^2, so in fact 12345.6789E2 equals 1234567.89.

A real may be preceded by a unary plus or unary minus. Also the number in the exponential part may be preceded by a unary plus or minus. A real literal is also called a floating literal. It should not exceed maximum and minimum values for reals.

Character Literals

A character literal (or character for short) is a single character within single quotes. An example of a character is ‘w’. A character can be any printable character or an escape sequence.

String Literals

A string literal (or string for short) is a sequence of characters between double quotes (apostrophes). Examples are “1234$%^&” and “Hello world”.

List Literals

A list literal (or list for short) denotes a list of elements. The elements can be of any type, but the elements in the same list must be of the same type. A list can be written in many ways, but the two most important ones are:

- write the elements between square brackets and separated by comma's. Examples are [1,6,3,2] (a list of integers) and ['w', 'd', 'a', 'q', 'w'] (a list of characters).
- partition the list in a Head (that is the first element) and the Tail (the other elements in the list). Head and Tail are partitioned by a vertical line "|". The partition is written [3 | 4,2,1,3] where the first "3" is the Head and [4,2,1,3] is the Tail. More on this in chapter 10.

Section

A Visual Prolog program consists of several files and each file is divided into different **sections** by appropriate keywords. The keywords inform the compiler what to expect in the section. I have given the example of predicates and clauses. The keyword "predicates" tells the compiler that in the section that begins with this keyword, it can expect the declaration of predicates. On the other hand, the definitions of predicates are to be found in sections that start with the keyword "clauses". Usually, each section starts with a keyword. There is normally no keyword which signifies the ending of a particular section. The presence of a keyword indicates the start of a section and the end of the preceding section. The exceptions to this rule are the keywords "implement" and "end implement" The code contained between these two keywords indicates the code to be used for a particular class.

Scope

Your program is divided into several parts, think of it as the files that make up your project. Everything that you declare, can be used freely within the file in which it is declared. But most of the time it cannot be used outside that part of your program. This is what is meant by the scope. The scope indicates where e.g. a predicate can be used. If you want to use a predicate outside its scope, then you must take precautions. E.g. when you use a predicate outside its scope, then you can indicate where to find it by preceding it with the name of the class or the object.

9.3 Overview section keywords

In this section you will find a short introduction to some of the keywords for sections in VIP. We give a short overview, the next sections give a comprehensive description. Not every keyword is described here, there are other keywords in Visual Prolog, but the ones mentioned here are for now the most important ones. No matter how I write them here (I have a stubborn word processor that loves to use UpperCase), keywords in your program must always be lowerCaseIdentifiers.

Constants

The keyword "constants" is used to mark a section of the code that defines commonly used values in the program code. For example, if the string literal "PDC Prolog" is to be used in multiple locations throughout the code, then you can declare and define a mnemonic (a short, easily remembered word) for this string thus:

```
constants
    pdc="PDC Prolog".
```

Note that the definition of a constant ends in a period (.). Unlike a Prolog variable, the name of a constant should be a lowerCaseIdentifier, that is a word starting with a lower case letter.

Domains
The keyword “domains” is used to mark a section declaring the domains that will be used in the code. A domain describes e.g. a range of values for variables or a kind of predicates. Well known domains are integer, real and string. There are many variations for the syntax of such domain declarations, and they cater to all possible kinds of domains that will be used later on in the code. As this book is a basic one, we shall only partly describe the domains declarations that are possible.

Facts
The keyword “facts”designates a section, which declares the facts that would be used later on in the code of the program. A fact can be a single variable, but it can also be a functor that holds many values. Each fact is declared with its name and in case of a functor, the arguments that are used for the respective facts along with the domains that those arguments belong to.

Predicates
The section that starts with the keyword “predicates” contains the declarations of the predicates that will later be defined in the clauses section of the code. Just like the facts declaration, the names that would be used for these predicates along with the arguments and the domains, to which the arguments belong to, are indicated in this section.

Clauses
Of all the sections that are present in a Visual Prolog code, the section that starts with the keyword “clauses” is the one that closely mimics a traditional Prolog program. It contains the actual *definitions* of the previously declared predicates. And you will find that the predicates used here will follow the syntax as declared in the *predicates* section.

Class facts
A fact denotes a kind of attribute. Most facts will be attributes of an object. In that case the fact will have probably a different value for different objects. But sometimes a fact is an attribute of a class as a whole. E.g. when you want to keep track of the number of objects that have been created, then this fact is a fact that belongs to the class, not to an individual object. In that case you have to declare the fact in the section “class facts”.

Class predicates
In the section “class predicates” you declare local predicates, that are predicates that can only be called from within the scope where they are defined. Local predicates are used for actions within a class (or object) that are not interesting for other objects. An example could be the sorting of a list before that list is printed.

Open
The keyword “open” is used to extend the *scope visibility* of the class. In this section you specify the names of other classes. When the compiler compiles a class (say: a module) and it encounters a predicate that is not in this class, then it will look in the classes mentioned in the “open” section. The keyword “open” is to be used right after the “implement” keyword.

Implement and end implement
Among all the keywords discussed here, these are the only ones, which exists as a pair. Visual Prolog treats the code written between the two keywords “implement” and “end implement” as the code that

belongs to one class. The name of the class MUST be given after the “implement” keyword. The name MAY be given after the “end implement”, but then it must be exactly the same as the name that is given after “implement”.

Now that I introduced the most used keywords, the next sections will give a detailed description of each keyword, its meaning and how to use it.

9.4 Section domains

Let me introduce the “domains” section in an informal way. Every variable in Prolog is of a certain *type*. The type of a variable indicates what values are acceptable for a variable. E.g. an variable of type integer can only have an integral value, that is a number like 123 or -456. In Prolog there are a few standard types. The ones to know were already presented in the previous section. They are: unsigned, integer, real, and string. When you use only these standard types in your program, you can omit the section “domains”.

The standard types are used to create your own types. An example. In the previous chapter we used a functor with the length and weight of persons. Length and weight cannot be negative. So we could want to indicate this to the compiler. To do it, we create a domain of positive integers. Let’s assume that length cannot be more than 300 (what is quite tall, I think) and weight cannot be more than 250 (and that is quite heavy :-)). Now we declare three domains, one for length, one for weight and one for name. It will look like:

```
domains
        length = integer [0..300].
        weight = integer [0..250].
        name = string.
```

In this declaration we state that length is an integral value within the limits 0 and 300. For the new domain “name” we only state that is is a string. Now turn to the functor “person”. The functor was named “person” and it had three arguments for Name, Length and Weight. In chapter 7 we declared the functor “person” as a fact:

```
facts
        person : (string, integer, integer)
```

But now, with the new domains at hand, we can make the declaration more meaningful:

```
facts
        person : (name, length, weight)
```

By using domains you reach two goals: the code of your program becomes more understandable and the compiler can do more precise checks. In the declaration in the previous chapter the second and third argument were both of type integer and could easily be confused. But with this declaration, the compiler knows that the second argument must be of type length and that is something different than type height.

When you declare a domain, you can use the name to create another domain. E.g. you could use the domain “weight” to declare a new domain “lightWeight”:

```
domains
        lightWeight = weight [0..50].
```

to indicate light weight people. And so on.

Now let's take a look at the formal syntax. For a start you may see a domain as a value range for a variable, but in this section we will see more kinds of domains. A domains section starts with the word

```
domains
```

that is followed by one or more domain declarations. A domain declaration is at the same time a domain definition. A domain definition defines a domain in the current scope, that is in the current file (class, module), that is the file in which the domain is defined. You cannot use the domain outside the current scope, unless you make an explicit reference to the scope where the domain is declared and defined. The syntax of a domain declaration in its simplest form is:

```
domains
    <domainName> = <typeExpression>.
```

A <domainName> is simply the name of the new domain. You are free to choose a meaningful name, but it must be a lowercaseIdentifier. The <typeExpression> is the name of a standard type (e.g. real) or a domain that you declared/defined before. In general the <domainName> is called the child domain and the <typeExpression> is the parent domain. But take care, there are exceptions to this rule.

<typeExpression>

There are a few basic types predefined in VIP. They are integer, unsigned, real, character and string[11]. These basic types are used to create any type that you want to use in your program. The wanted types are declared in the domains section and defined by one or more <typeExpression>. A <typeExpression> is an expression that denotes a type. A <typeExpression> can be

- integralDomain
- realDomain
- typeName
- compoundDomain
- listDomain
- predicateDomain
- typeVariable
- typeApplication

Because this is a book for beginners, I shall skip the predicateDomain, the typeVariable and the typeApplication.

Integral Domain

Integral domains are used for representing ranges of integral numbers, or integers for short. There are two predefined domains for integers: "integer" and "unsigned", they represent signed and unsigned numbers respectively with natural representation length for the processor architecture (i.e. 32bit on a 32bit machine, etc). Probably you will not bother much about the representation :-).

To declare a new integer domain you should declare:

```
domains
myDomain = integer.
```

After the word "integer" you can state two parameters, <sizeDescription> and <rangeDescription>. The syntax then becomes:

[11] There is also the type symbol, but I was advised not to mention it.

```
domains
   <domainName> = <typeExpression> <sizeDescription> <rangeDescription>.
```

The <sizeDescription> indicates the number of bits that should internally in memory be used for the values in <domainName>. For <sizedescription> you use the word "bitsize" followed by the (integral) number of bits. The <rangeDescription> denotes the value range of the values in <domainName>, it consists of a minimum and a maximum value. The <rangeDescription> is indicated by two (integral) numbers between square brackets, separated by two dots. So to declare that a domain with the Name "newDomain" contains integer values, has a range from 0 to 10 and should be stored in 8 bits we declare (and define):

```
domains
   newDomain = integer bitsize 8 [0..10].
```

If <sizeDescription> is omitted, then the compiler will assume it to be the same as the parent domain. If there is no parent domain, it will become the natural size for the processor. If a <rangeDescription> is omitted, the range of the parent domain is used. If there is no parent domain, then the range will depend on the <SizeDescription>. E.g. an unsigned num ber that is stored in 8 bits cannot be larger than 255 (decimal). Of course you must not specify a range where the minimum exceeds the maximum. Also minimal and maximal limits should satisfy the limits implied by the specified <SizeDescription>.

If you want to, you can use arithmetic expressions for <sizeDescripton> and <rangeDescripton>, but they must result in a number at compile time. An example:

```
constants
   three = 3.
domains
   subint = integer [0..three].
```

Real Domains

Real domains are used to represent numbers with fractional parts (i.e. floating point numbers). Real domains can be used to represent very large and very small numbers. The built-in domain "real" has the natural precision for the processor architecture (or the precision given by the compiler).

Just like it is done with integers, you declare a real domain by stating:

```
domains
   <realDomainName> = real <realPrecisionDescription> <realRangeDescription>.
```

After the word "real" you can specify two special properties of the new domain: <realPrecisionDescription> and <realRangeDescription>. The <realPrecisionDescription> declares precision of the domain, measured in number of decimal digits, that is the number of digits right of the decimal point. You specify the <realPrecisionDescription> by declaring the keyword "digits" followed by the (integral) number of digits that should be used in the decimal part. If <realPrecisionDescription> is omitted then it will become the same as for the parent domain. If there is no parent domain, then it will be the natural precision for the processor or given by the compiler (in Visual Prolog v.6 the compiler limit is 15 digits). The precision has upper and lower limits given by the compiler, if the precision that you specify is larger than that limit, then numbers will only obtain the processor (compiler) specified precision anyway.

The <realRangeDescription> is the same as for integral value ranges: two number between square brackets and separated by two dots. As this is a range for real numbers, the minimum and maximum are real numbers. You can use expressions for these number, but they must evaluate to a floating point value at compile time. That is the compiler must be able to calculate the real domain precision and limits at compiling time.
The real range description declares minimal and maximal limits for the real domain. If a limit is omitted then it will be the same as for the parent domain. If there is no parent domain then the largest possible range for the declared precision will be used.

Type Names
A type name is a <domainName> that is defined in the current scope or elsewhere. When you have defined a <domainName>, you can use that name in other declarations. This is called using a typeName.

Using typeNames for declarations is fairly straightforward. An example:

```
domains
   <domainName> = <aPreviousDeclaredDomain>
```

with optional parameters for e.g. bitsize, precision and/or range. If you want to use a range from another scope (say: another class), then you must precede <aPreviousDeclaredDomain> with the name of the class and a double colon. An example:

```
domains
   myNewDomain = anotherClass::myOld Domain.
```

In this example, “myNewDomain” will be a subtype of “myOldDomain” and you must take care that the properties (bitsize, range, precision) of “myNewDomain” do not violate the properties imposed by “myOldDomain”. E.g. the range of “myNewDomain” must not be greater than the range of “myOldDomain”. Also the bitsize or precision cannot be greater.

Compound Domains
The domains mentioned till here are simple domains. There are also compound domains (also known as algebraic data types). They are used to represent lists, trees, and other tree structured values. In its simple forms compound domains are used to represent structures that you have seen before as functors, and enumeration values. An example is:

```
domains
   person = person(string, integer, integer).
```

Although the declaration is correct, it is not very clear. An alternative is:

```
domains
   person = person(string Name, integer Length, integer Weight).
```

In this case the compiler ignores the variables Name, Length and Weight. But your program is better understandable.

This can become more complex. Let’s assume that we want to make a database of persons and their belongings. Suppose that a person can have a book (with title and author) or a guitar (with a brand name). Now we can declare:

```
domains
   personName = string.
   author = string.
```

```
    title = string.
    brandName = string.
    possession = book(author, title); guitar(brandName).    % notice the semicolon, a logical "or"

class facts
    person : (personName, possession).
```

Facts will be treated in the next section. Now the declaration says that the functor "person" contains two arguments, a personName and a possession. A personName is from the domain string (or for short: is a string), a possession can be from the domain book or from guitar. The semicolon is read as a logical "or" . In your program you can initialize the facts with clauses like:

```
    person("John", book( "Eduardo Costa","Prolog for Tyros")).
    person("John", guitar("Fender")).
```

And you can use statements like:

```
    ...,
    person(Name, Possession),       %The two variables are bound here
    writePossession(Possesion),
    ...
```

And then have the clauses for "writePossesion":

```
writePossession(book(X,Y)) :-
    write(X,Y).
writePossession(guitar(X)) :-
    write(X).
```

Compound domains have no subtype relations to any other domain. If a domain is defined as being equal to a compound domain, then these two domains are synonym types rather than subtypes. Meaning that they are just two different names for the same type.

Compound domains can have a recursive definition. They can also be mutually/indirectly recursive.

Example I

```
domains
    t1 = ff(); gg(integer, t1).
```

t1 is a compound domain with two alternatives. The first alternative is the null-ary functor ff, while the second alternative is the two-ary functor gg, which takes an integer and a term of the domain t1 itself as arguments. So the domain t1 is recursively defined. The following expressions are terms of the domain t1:

```
ff()
gg(77, ff())
gg(33, gg(44, gg(55, ff())))
```

Example II

```
domains
    t1 = ff(); gg(t2).
    t2 = hh(t1, t1).
```

t1 is a compound domain with two alternatives. The first alternative is the null-ary functor ff, while the second alternative is the unary functor gg, which takes a term of the domain t2 as argument. t2 is a compound domain with one alternative the functor hh, which takes two t1 terms as argument. So the domains t1 and t2 are mutually recursive. The following are terms in the domain t1:

```
ff()
gg(hh(ff(), ff()))
gg(hh(gg(hh(ff(), ff())), ff()))
ggg(hh(ff(), g(hh(ff(), ff()))))
gg(hh(gg(hh(ff(), ff())), gg(hh(ff(), ff()))))
```

In chapter 11 we will see more compound domains like trees.

List Domains

List domains represent sequences of values of a certain domain. Normally we think of a list as a series of elements between square brackets and separated by comma's. The elements must be of the same type. An example: [1, 2, 3, 2, 1] is a list of integers. You declare a list by naming the domain of the elements, followed by an asterisk.

```
domains
  realList = real*.
```

9.5 Section constants

A constant is a variable that has the same value throughout your program. One uses a constant to make the program more understandable. It makes more sense to use in the code the variable "VATpercentage" than the value 0.20. Besides when you use a constant and the value changes (taxes do change!), then you only have to change the constant declaration. This will prevent mistakes.

A constants section defines one or more constants in the current scope. It starts with the keyword

```
constants
```

For every constant you create a line that states the name of a constant, followed by a colon, the type of the constant, an "="sign and its value. The line ends with a dot. So in general the declaration of a constant looks like:

```
constants
   <constantName> : <constantType> = <constantValue>.
```

The <constantName> is a lowerCaseIdentifier. The <constantValue> should be conform the <constantType>. If it is an expression then the compiler must be able to evaluate it at compile time. Examples:

```
constants
   vat : real = 0.19.
   constVIP : string = "Visual Prolog".
```

If the domain of the value is one of the standard domains, then the name of the domain and the colon ' : ' can be omitted, giving the following abbreviation:

```
<constantName> = <constantValue>
```

E.g.

```
vat = 0.19.
```

Constants defined in this way can be used in all contexts where a literal of the same kind could be used.
If the <constantType> is omitted, then the constant domain must be unambiguously determined by the <constantValue> expression. Take care of the way VIP distinguishes between integers and reals. The number “123" is an integer, the number “123.” is a real.

9.6 Section facts

A fact describes the values for one or more attributes of an object or of a class. Facts in Prolog are stored in a database. Every fact has its own database. There are two kinds of facts. When the fact represents the value of a single attribute, then we call it a “factVariable”. When a fact describes the value of two or more attributes, then we call it a “factFunctor”.

A facts section declares the facts that are contained in a fact database. The fact database and the facts belong to the current scope. As explained before, a fact database can exist on a class level as well as on an object level. When it exists on an object level, then facts will be different for different objects and you have to add, change or remove facts by calling an object predicate. When a fact exists on class level, then you must add, change or delete a fact by calling a class predicate. In the latter case you must declare the fact to be a “class fact”. See the end of this section.

Facts sections can be declared only in class implementations, that is in the file “<className>.pro”. Fact data bases can be given a name. If the fact database is named, an additional compound domain is implicitly defined. This domain has the same name as the fact database and has functors corresponding to the facts in the fact section.

A facts declaration starts with the word

```
facts
```

followed by the one or more facts declarations. When you want to give the facts database a name, the declaration looks like:

```
facts - name_of_the_database
```

The name is an lowerCaseIdentifier. The word “facts” and the name are separated by a dash “-“. The facts database is now called a named facts database.

Each declaration after the word “facts” is ended with a period. The declaration of a factVariable differs slightly from the declaration of a factFunctor. The simplest fact declaration is the declaration of a factVariable. It states the name of the fact, its type and an initial value:

```
<factVariableName> : <factType> := <initialValue>
```

The <factVariableName> is a lowerCaseIdentifier. The <factType> can be any declared domain. The <initialvalue> must of course be in the declared domain and, when it is an expression, must evaluate to a value at compile time. The <initialValue> may be omitted, but then the factVariable must be initialized in the constructor, that is the fact must be initialized at the moment when the object that it belongs to, is created. The factVariable cannot exist without a value.

To change the value of a factVariable, you must use the assignment sign “:=”. So when you declare the factVariable

```
facts
   myFact : integer := 0
```
then the factVariable myFact initially gets the value zero. To change the value, you must use a statement like:

```
...,
myFact := 3,
...
```

The declaration of a factFunctor consists of the name of the factFunctor, followed by a colon, the arguments of the functor between round brackets and optionally the mode of the factFunctor. It looks like this:

<factFunctorName> : (<argumentlist>) <factFunctorMode>

The <factFunctorName> is a lowerCaseIdentifier. The <argumentlist> contains the types of the arguments and an optional variable name for each argument. The types are separated by comma's. When you add variable names, there must NOT be a comma between the type and the name. An example:

```
facts
   person : (string, integer)
```
defines a factFunctor with two arguments. The declaration

```
   person : (string Name, integer Length)
```
defines the same factFunctor. The compiler neglects the names of the variables.

The <factFunctorMode> is optional. It can only be used with a factFunctor and it indicates the number of facts that can be in the database. (Obviously a factVariable can have only one value). The mode can be determ, nondeterm or single.

- If the mode = single, then a fact always has one and only one value. If you assert a new value to the facts database, then the old values are overwritten. You cannot retract a single fact. If a factFunctor has mode = single, then a call of this fact will always succeed.
- If the mode = determ, then the fact can have zero or one value. If a fact has zero values, then any read access to it gives fail. If a factFunctor has mode = determ, then a call of this fact can fail (when there are no facts) or it can succeed.
- If the mode = nondeterm, then the fact can have zero, one, or any other number of values. If a factFunctor has mode = nondetrm, then a call of this fact can fail, succeed and it can return with a backtrackpoint that can give more solutions.
- When you omit the <factFunctorMode>, the mode = nondeterm is assumed.

A factFunctor is initialized via the clauses section. When you declare the factFunctor:

```
facts
   person : (string Name, integer Length)
```
then in the clauses section (see below) you can add clauses like:

```
person("John", 185).
```
Initially the data base is filled with these facts. In this case take care that the expressions in the clauses can be evaluated at compile time.

Facts declared in the section that begins with the keyword "facts", are by definition object facts. The section in which you declare class facts starts with the keyword

class facts

The section “class facts” is syntactically equal to “facts”section. So a class facts database can contain factVariables and factFunctors. Et cetera.

Facts can only be declared in a class implementation (the file with the name “<className>.pro”) and subsequently they can only be referenced from within this implementation file. So the scope of facts is the implementation in which they are declared. But the lifetime of object facts is the lifetime of the object to which they belong. Likewise the lifetime of class facts are from program start to program termination.

9.7 Section predicates

The section “predicates” declares a set of object or class predicates in the current scope. The section begins with the word

predicates

that is followed by one or more predicate declarations. A file can contain more than one section “predicates”. In its simplest form a predicate declaration consists of a predicate name, followed by a colon and a list of arguments between round brackets and it ends with a period.

<predicateName> : (<argumentlist>) .

The <predicateName> is a lowerCaseIdentifier. The <argumentlist> states the types of the arguments. Types can be standard types or declared domains. The argumenttypes are separated by commas. An argument may be followed by a variable name; in that case there is no comma between type and variable name. An example:

myPredicate : (integer, real, string).

When using variable names, the declaration becomes:

myPredicate : (integer FirstNumber, real SecondNumber, string TheString).

The variable names are neglected by the compiler.

There are several options that can (and sometimes must) be added to the declaration.

- When the predicate is a function, then the argument that contains the value_to_be_returned must be declared.
- Optionally you can state the mode of the predicate.
- Optionally you can state the flow pattern of the predicate.

When you omit mode or flow pattern, the compiler assumes values for them.
When these options are added, the declaration looks like:

<predicateName> : (<argumentlist>) -> <returnArgument> <predicateMode> <predicateFlow>

These options have the following meaning.

-> <returnArgument>

The option <returnArgument> is preceded by the symbol “->”. It consists of the type of the return value (the domain where the value is from) and a Variable Name. When you add <returnArgument> to the declaration, the predicate is called a function. You have to take precautions that the clauses return the right value. Let’s take a closer look at functions.

When you call a predicate, some of the arguments can be used as output arguments. Upon return, you can retrieve their values and in this way you can make a predicate provide you with the values that you want. Another way to retrieve values when calling a predicate, is the function. A function is a special kind of predicate. It returns one value simply by calling, without using an explicit output argument. Let me give an example. Take a look at the predicate squarePred/2 that computes the square of an integer. This is the predicate declaration

```
predicates
   squarePred : (integer InputVar, integer OutputSquare) procedure (i,o).
```

And this is the clause[12]

```
clauses
   squarePred(InputNumber, OutputSquare) :-
   OutputSquare = InputNumber * InputNumber.
```

When this predicate is called , the InputNumber is used to compute the square, that is bound to OutputSquare. So when you call it with

```
   squarePred(2, Square),
```

then Square will be bound to the number 4. Now take a look at this predicate

```
predicates
   squareFunc : (integer InputVar) -> integer OutputSquare procedure (i).
clauses
   SquareFunc(InputNumber) = OutputSquare :-
   OutputSquare = InputNumber * InputNumber.
```

This predicate takes the InputNumber to compute the square and binds OutputSquare to it. It then returns the value of OutputSquare to the calling program. When you call this predicate, you must provide a variable to catch the returned value like this:

```
...,
Square = squareFunc(3),
...
```

This will bind Square to what the functon squareFunc/1 returns. In the declaration the part:

```
-> integer OutputSquare
```

indicates that this predicates is a function that will upon termination return the value that is bound to the variable OutputSquare.

<predicateMode>

The option <predicateMode> indicates the behavior of the predicate. That is, it tells the compiler if the predicate can fail, can succeed or can backtrack (or both or all three options). There are six different modi possible.

- <predicateMode> = fail
  the mode “fail” indicates that the predicate will always fail. There is one standard predicate that has this mode: it is the predicate fail/0.
- <predicateMode> = procedure
  the mode “procedure” indicates that the predicate will always succeed and that it will succeed with only one solution. No alternative solutions are possible, no backtracking will take place.
- <predicateMode> = determ
  the mode “determ” indicates that the predicate can fail or succeed, but that it never returns with a backtrackpoint. That implies that when the Prolog engine backtracks, it will skip calls to this

[12]Clauses are treated in the next section.

predicate as it can never have a backtrackpoint. When it succeeds, there will be only one solution.

- <predicateMode> = multi
  the mode “multi” indicates that the predicate never fails, it will always succeed and it may come back with a backtrackpoint so a call to this predicate may result in multiple solutions.
- <predicateMode> = nondeterm
  the mode “nondeterm” indicates that the predicate can fail, can succeed and can return with a backtrackpoint.
- <predicateMode> = erroneous
  the mode “erroneous” is a very special one. It does not fail or succeed, but it raises a so-called exeption. An exception is a deviation of a program from its "normal" execution path, for example, when a file cannot be found, memory overflow occurred, etc. If an exception occurs, then a specified exception handling predicate should be executed. An exception handler somehow analyses what kind of exception occurred and then can do some fixing action, for example, allow the user to enter a correct file name and recall the predicate, or just show a message with information about an exception. I shall skip this mode here.

When you omit the <predicateMode>, then the compiler assumes that the predicate has mode “procedure”. It is illegal to state a predicate mode for constructors, they always have the mode procedure. Table 9.1 gives an overview

predicate mode	succeeds or fails	# solutions	backtrackpoint?
fail	always fails	none	no
procedure	always succeeds	one	no
determ	can fail or succeed	none or one	no
multi	always succeeds	one or more	yes
nondeterm	can fail or succeed	none, one or more	yes

Table 9.1 Predicate modi

With the <predicateMode> you give the inference engine in fact an indication how you intend to use the predicate.

In the sequence: procedure, determ, multi and nondeterm we say that a mode is stronger (more stringent) than an other mode when it permits less options and that it is weaker when it permits more options. So “procedure” is stronger than “determ” as determ not only allows succeed, but also fail. When declaring predicates you must take care that the modes are compliant with each other. E.g. when you declare a predicate to be “procedure” and in the clauses of this predicate you call another predicate that is nondeterm, then in fact you change the mode of the calling predicate. Luckily you can change the behavior of a predicate by using the cut “!” and the predicate fail. Let me give an example. Take a look at the following clauses.

```
child(Name) :-
   father(Name, Father), write(“Father is: “, Father).
child(Name) :-
   mother(Name, Mother), write(“Mother is: “, Mother).
```

This predicate tells us that someone with the name Name is a child when it has a father or a mother. Assume that we have a database with several facts for the functors “father” and “mother”. Let’s call this predicate with a certain name, say

child(“John”).

Now this call can obviously fail when no fact is found with the name “John”. When on the other hand it finds a fact for father(“John”, Father) then it succeeds and it returns the first solution. But it also returns a backtrackpoint because there is another clause in the database it can try. So this predicate is nondeterm, it can come back with one of {fail, succeed, backtrackpoint}. Now let’s add a cut to the first clause:

```
child(Name) :-
   father(Name, Father), !, write(“Father is: “, Father).
child(Name) :-
   mother(Name, Mother), write(“Mother is: “, Mother).
```

When the first claus succeeds, the cut will prevent backtracking. So the predicate can come back with one of {fail, succeed}, that is mode = determ. Finally you can also change the predicate into the mode procedure. Take a look at this:

```
child(Name) :-
   father(Name, Father), !, write(“Father is: “, Father).
child(Name) :-
   mother(Name, Mother), !, write(“Mother is: “, Mother).
child(_) :-
   write(“No known parents”).
```

The last clause plays the role of a catch_all clause. When the other two clauses fail, the last one will succeed, no matter the Name used in the call. In this definition because of the cuts only one clause will succeed and because of the catch-all clause, at least one will succeed. So mode = procedure.

<predicateFlow>

The option <predicateFlow> indicates whether an argument is an input argument or output argument. When you call a predicate, the input arguments must have a value. On return you use the output arguments to retrieve the values that you need from the predicate. The <predicateFlow> is given by a sequence of the characters ‘i’ and ‘o’ between round brackets and separated by commas. The character ‘i’ indicates an inputargument, the character ‘o’ indicates an outputargument. The characters are written without the quotes. They must be given in the order in which the arguments are given in the <argumentlist>. When a predicate has five arguments and the first three are input and the last two are output then the <predicateFlow> looks like

(i, i, i, o, o)

So a predicate declaration can take the form

myPredicate : (string Title, integer NrOfPages) procedure (i,o).

The declaration says that myPredicate has two arguments, one string and one integer, the mode is procedure (that means it always succeeds, no backtrackpoints), that you have to input the string and that it will return the number of pages in the second argument.

It is possible to declare more than one <predicateFlow>. The declared <predicateMode> applies to all flows that follow it. When different flows have diffferent modes, you must declare it. It can lead to a sequence like:

... ... procedure (i, o) nondeterm (o, o).

This declaration says that with the flow (i,o) the predicate has mode = procedure and with the flow (o,o) the mode = nondeterm.

When declaring a predicate, the flow can be omitted. Inside an implementation (i.e. for a private predicate) the needed flows are derived from the usage of the predicate. Inside an interface or a class declaration (i.e. for a public predicate) omitting flows means that all arguments are input. The special flow pattern “anyflow” can be stated only in declarations of local predicates (i.e. in predicate declarations inside the implementation of a class). It means that the exact flow pattern will be evaluated during the compilation. If the optional preceding predicate mode is omitted for the flow pattern “anyflow”, then it is assumed to be a procedure.

Flows can also be applied to functors and lists, but I’ll leave that out.

9.8 Section clauses

A section “clauses” begins with the keyword

```
clauses
```

and consists not surprisingly of a set of clauses. A file can have more than one section “clauses”. The clauses section is meant to define (or specify) implementations of predicates or initial values of facts. A single clause section can have clauses for several predicates and facts. On the other hand, all clauses for one predicate or fact (that is clauses or facts with the same name and arity) must be grouped together in one clauses section and without intervening clauses of other predicates or facts.

Clauses are used to define predicates. A single predicate is defined by one or more clauses. Each clause is executed in turn until there are no more clauses left to execute. If no clause succeeds the predicate fails.

If a clause succeeds and there are more relevant clauses in a predicate left, the program control can later backtrack to the clauses of this predicate to search for other solutions. This is called setting a backtrackpoint. Thus, in principle a predicate can fail, succeed, and even succeed multiple times.

A clause consists of at least a head and optionally a body. The head of a clause consists of a predicate name followed by the right number of arguments between round brackets, e.g.

```
person(“John”, 185).
```

This is a clause that states that John has a length of 185. You can think of it as a statement that is unconditionally true. That’s why it is also called a fact.

When you add a body to the head, the head must be separated from the body by the “:-“sign. E.g.

```
grandFather(GrandChild, GrandFather) :-
    parent(GrandChild, Person),
    father(person, GrandFather).
```

The “:-“sign reads as a logical IF. The body can be seen as one or more conditions. This is the logical interpretation of a clause. When the conditions are met, the head is true. The body consists of statements, separated by commas. The comma “,” reads as a logical AND. When you separate statements with a semicolon “;”, this reads as a logical OR. The use of the semicolon in clauses is not recommended, as it easily leads to confusion. So in stead of

```
child(Name) :- father(Name, Father) ; mother(Name, Mother).   % note the semicolon
```
we prefer to use two separate clauses that have exactly the same effect.
```
child(Name) :- father(Name, Father).
child(Name) :- mother(Name, Mother).
```

There is also the procedural interpretation of the clause. In this interpretation the head of the clause is the procedure head that is to be called, and the statements after ":-" will be executed when the predicate is called. Which interpretation you need depends on your program and the side effects of the statements.

When a predicate is called the clauses are tried in turn (from top to bottom). For each clause the head is unified with the arguments from the call, that is the input arguments are bound to the values that are provided by the calling predicate. If this unification succeeds then the body of the clause (if such one exists) is executed. The clause succeeds if the match of the head succeeds and the body succeeds. Otherwise it fails.

When a predicate is a function, then you must declare right after the heading the "="sign and the variable that is bound to the value that must be returned by the function. E.g. when you declare
```
square : (integer) -> integer Square procedure (i).
```
to declare a function that squares an integer, then the clauses section should contain the necessary clauses, e.g.
```
square(Number) = Square :-
   Square = Number*Number.
```

The expression "= Square" tells the compiler that upon termination of the clause for the predicate square/1, the value of the variable Square must be returned to the calling predicate. To call a function you must have a variable ready to be bound to the returning value. E.g.
```
...,
NewSquare = square(3),
...
```
This binds Newsquare to the value 9.

9.9 Section goal

The goal section defines the main entry point into a Visual Prolog program.

In traditional Prolog, you provide the inference engine with the predicate it should start with. That can be any predicate. Remember the "family"-program you used in PIE. It was up to you to enter the wanted goal and the program used the named predicate to start with. However, this is not the case with Visual Prolog. In Visual Prolog programs are compiled and the compiler has the responsibility to produce efficiently executing code for the program you write. It is important to notice that compiling the program and executing the (compiled) program are two different things. First the program is compiled and then (later) the compiled program is executed. Hence, the *compiler* needs to know beforehand the exact predicate from which the code execution will start, so that later on when the program is called to perform, it can do so from the correct starting point. As you may have guessed, the compiled program can run independently without the Visual Prolog compiler or the IDE itself.

In order to obtain a compiled program, there is a special section that starts with the keyword “goal”. This section contains a special predicate “run” that is used without arguments. That predicate is the one from which the entire program execution will start. So the goal section is the entry to a program. It starts with the word “goal”, followed by a clause body. An example:

```
goal
   mainExe::run(main::run).
```

I made a program called ”procfunc” that writes to “console”. The goal says that the engine should look for the predicate “run” in the class “main”. This class was created by the IDE when I created the new program (project). The clause for “run” in “main” looks like this

```
clauses
   run():-
      Kwad = func::squareFunc(3),          % I added this line for my program
      stdIO::write(Kwad),                  % I added this line for my program
      _X = stdIO::readchar(),              % I added this line for my program
      succeed(). % place your own code here
end implement procfunc
```

You don’t have to bother much about this as the IDE takes care of it. But one thing may embarrass you. Normally, the predicate run/0 in the goal must have procedure mode. It means that the predicate “run/0" has mode = procedure. Now when you insert code (as I did in the example above), it may happen that you call a predicate that has a less strong mode, e.g. nondeterm. Then the compiler will complain that by calling this predicate, you in fact change the mode of run/0 to nondeterm. This is annoying, but you will have to solve this problem in some way.

As a last remark, the goal section defines its own scope, therefore all calls to other predicates (invocations) should contain class qualifications. That is you must give the class name where the called predicate resides.

9.10 Section open and scope access issues

Visual Prolog divides the total program code into separate parts, each part defining one class. In object oriented languages, a class is a package of code and its associated data which are put together. Classes were introduced in chapter 8. As noted earlier, for those of you who are not familiar with object oriented programs, you can think of a *class* loosely as being synonymous to a *module.* Normally, Visual Prolog defines each class in its own separate files.
During the program execution, it often so happens that the program may need to invoke a predicate that is actually defined in another class (file). Similarly, data (constants) or domains defined in a class may need to be accessed from a different file.
Visual Prolog allows such code and data references across classes using a concept called scope access. This can be understood by an example. Suppose there is a predicate called pred1 defined in a class called class1 (which is written in the file “class1" using the IDE), and we need to call that predicate in the clause body of some predicate, pred2 in a different file (say “class2") then this is how pred1 would be called within the clause body of pred2 :

/* this is a part of the clauses in “class2" */

```
    pred3:-
       ...,
       ... .
    pred2:-
            class1::pred1, %pred1 is not known in this file.
                           %It is defined in some other file,
                           %Hence a class qualifier “class1::” is needed
            pred3,
            ...
/* end of the part from “class2" */
```

In the above example, you can see that the clause body of pred2 calls two predicates pred1 and pred3. As pred1 is defined in another file (which defines “class1"), the word class1 with the token **::** (two colons) precedes the word pred1. This is called a class qualifier.
But the predicate pred3 is defined within the same file as pred2 itself. Hence there is no need to indicate class2:: prior to the predicate call for pred3.

This behavior is technically explained thus: The *scope visibility* of pred3 is within the same scope of pred2. Hence there is no need to clarify that pred3 is from the same class as pred2. The compiler will automatically look for the definition of pred3 within the scope area as defined for class2.
The scope area of a particular class definition is restricted to the class that is implemented in a particular file (i.e. code that is written within the “implement” - “end implement” keywords). The predicates defined therein can call each other without the class qualifier and the double colon (::) token preceding it.
The scope area of a class can be extended by using the keyword “open”. This keyword informs the compiler to bring in names of predicates, constants or domains that are defined in other files. If a scope area is extended, then we need not write the class qualifier with the double colon.

```
/* this is a part of the clauses in “class2" */
    open class1
        ...
    pred3:-
             ...
             !.

    pred2:-
            pred1, % Note: "class1::" qualifier is not needed
                   % anymore, as  the scope area
                    % is extended using the 'open' keyword
            pred3,
            ...
/* end of the part from “class2" */
```

The section “open” does not look like a section, it looks more like a line that starts with the keyword “open” followed by a comma separated list of names (of classes). An example

```
    open another_class, yet_another_class.
```

It ends not with a period but with a hard return.

Open qualifications are used to make references to class level entities more convenient. The open section brings the names of one scope into another scope, so that these can be referenced without qualification.
Open has no effect on the names of object members as these can only be accessed by means of an object anyway. But names of class members, domains, functors, and constants can be accessed without qualification.
When names are brought into a scope in this way it may happen that some names becomes ambiguous

Open sections have only effect in the scope in which they occur. Especially this means that an open section in a class declaration has no effect on the class implementation.

9.11 Class predicates, class facts and where to declare them

When a class with name <className> can create objects, the code for it is divided in VIP over three files. They are

- The file "<className>.cl" is called the class declaration.
- the file "<className>.i" is called the interface.
- the file "<className>.pro" is called the implementation.

These files were described in chapter 8. To repeat in short: the class interface contains the declarations that are public (can be called from within other files) and that are connected to the objects that are generated by the class. The class interface can be seen as a description of what (predicates, facts) the outside world sees of the objects of a class. So you can think of an interface as a declaration of the type of objects that the class generates. That why we sometimes say that an interface defines a named object type. All predicates declared in an interface are object members in objects of the interface type.
When a class cannot generate objects (that is up to you to decide), then the class interface file is omitted and the code for the class is spread over two files. The class declaration file contains the declarations of the predicates that are also public but that are connected to the class as a whole. A class declaration file can contain constant and domain definitions and predicate declarations. The other parts of the program can see and use exactly those entities mentioned in the class declaration. We say that the declaration of a class specifies the public part of the class. Finally the class implementation file contains the definitions of the predicates, that is the clauses of the predicates and facts that are declared in the class declaration and interface. A class implementation is used to provide the definitions of the predicates and constructors declared in the class declaration, as well as the definitions of any predicates supported by its constructed objects.

In each of these files you can enter other sections besides sections for predicates and facts. But take good notice of the scope of them. See section 9.10.

For predicates and facts things are a bit more subtle as there can be predicates, class predicates, facts and class facts. I'll try to explain what to take care about, but things may get confusing (or confused :-)).

Facts belong to objects, that is with each object from the same class there can (and normally will) be different facts. These kind of facts must be declared (and defined) in the class interface file. But when a fact belongs to the class as a whole, then it is a class fact and must be declared (and defined) in the

class declaration file. But take care. When you declare a fact in the interface file, then it is by definition an object fact. When you declare a fact in the class declaration file, then it is by definition a class fact. Only in the class implementation file confusion is possible. Declaring facts in the implementation file means that the fact is by definition private. It cannot be seen from outside the class implementation. But it can be an object fact or a class fact. So when you declare a fact in the implementation file, you must indicate whether it is an object fact or a class fact. When a fact is a class fact, you must declare it in a section that begins with the keywords

class facts

Table 9.2 gives an overview.

fact declaration is ...	public fact	private fact
for an object member	in class interface “facts”	in implementation “facts”
for a class member	in class declaration “facts”	in implementation “class facts”

Table 9.2 declarations of facts

The predicate declaration is used to declare the predicate in scopes in which the predicate declaration can be seen. When predicates are declared in an interface file, this means that the predicate is an object predicate and publicly available and objects of the corresponding class must support these predicates. When predicates are declared in a class declaration this means that the class publicly provides the declared predicates. And if predicates are declared in a class implementation, this means that the predicates are only available locally (private). In the latter case, a predicate can be an object member or a class member. When it is an object member, then it must be declared in a section that starts with the keyword “predicates”. When it is a class predicate, then it must be declared in a section that starts with the keyword “class predicates”. In all cases a corresponding definition of the predicates must exist. That is to say that there must be clauses for the declared predicate in the implementation file. Table 9.2 gives an overview.

Predicate declaration is ...	public predicate	private predicate
for an object member	in class interface “predicates”	in implementation “predicates”
for a class member	in class declaration “predicates”	in implementation “class predicate”

Table 9.3 declarations of predicates

The keyword “class” can be used only inside a class implementations. Outside the implementation it has no use, as e.g. predicates, declared in an interface, are always object predicates and predicates, declared in a class declaration, are always class predicates already.

So a class can privately (i.e. inside the implementation file) declare and define more entities than those mentioned in the declaration and/or the interface files. Especially, an implementation can declare fact databases that can be used to carry class and object state. An implementation is a mixed scope, in the sense that it both contains the implementation of the class and of the objects produced by

the class. The class part of a class is shared among all objects of the class, as opposed to the object part, which is individual for each object. Both the class part and the object part can contain facts and predicates, whereas domains, functors, and constants always belong to the class part, i.e. they do not belong to individual objects.

By default all predicate and fact members declared in the implementation of a class are object members. To declare class members the section keyword (i.e. predicates and facts) must be prefixed with the keyword class. All members declared in such sections are class members. Class members can reference the class part of a class, but not the object part. Object members, on the other hand, can access both the class part and the object part of the class.

An interface declares how the outside world sees the objects of that type. But take care! When you declare constants and domains in the class declaration, then you can reach them as part of an object:

<objectIdentifier>:<constantName>

with a colon between <objectIdentifier> and <constantName>. But when you declare constants and domains in the interface file, they are not seen as object members. In this case the interface is also a global scope, in which constants and domains can be defined, but these constants and domains defined in an interface are not part of the type that the interface denotes (or of objects having this type). Such domains and constants can be referenced from other scopes by qualification with the interface name:

<interfaceName>::<constantName>

or by using an “open” qualification. (See section 9.10)

Chapter 10 Recursion, lists and sorting

In this chapter I introduce the concept of recursion and the list. Both are very much used in Prolog programming. As a matter of fact you have seen both in the previous chapters, but here I will bring out the details that you should know. In the first section I turn to recursion. The second section is on lists and in the third section the two will meet to show some powerful possibilities. In section 10.4 I introduce the class "list" that contains many handy predicates for list manipulation The fifth section is on sorting, as sorting shows you some nice applications of recursion and lists.

I should warn you beforehand though, that in a certain way this chapter is superfluous. Every example program that you find in this chapter is also available as a standard predicate. They are available in the class "list". When you add this class to the "open" declaration you get access to all these predicates. I give an overview in appendix A2. But I think that when you go through the examples in this chapter, your understanding of Prolog will improve. Besides it is highly probable that you will use lists and recursion in your own programs.

10.1 Recursion

Remember what you did in elementary school when you started to learn to count from 1 to 10? You started with the number 1, then added 1 to make 2, then added 1 to make 3, then added 1 ... et cetera until you reached 10. What you did was that you started with 1 and then over and over again called the same function ("add 1") to get to the next number. This is the basic idea behind recursion. In general we say that recursion takes place when a function calls itself, thereby generating an endless series of calls. As functions in Prolog are predicates, in Prolog we say that recursion takes place when a predicate calls itself. As an example look at the predicate to count:

```
predicates
        count : (integer Number) procedure.
clauses
        count (Number) :-
                stdIO::write(Number),
                NextNumber = Number + 1,
                count(Nextnumber).
```

When we call this predicate with "count(1)" it starts counting like you did. First it writes the number 1, then it adds 1 to make 2, then it calls itself with "count(2)" so it writes the number 2, then it adds 1 to make 3, then ... ad infinitum. The predicate "count" is a recursive predicate.

It may bother you that when you call "count/1", it never stops. So let's add a clause to stop the recursive calling. Suppose that we want to stop the counting when the number 10 is written. That means that we want the predicate to stop when it reaches 11. We can do that with this clause:

```
        count(11) :-
        stdIO::write("That's all folks!"), !.
```

When we put this *in front* of the other clause then every time the predicate is recursively called, it first will check if Number has got the value 11. If not, then it continues counting. If yes, then it prints the message and stops. Because of the cut also the backtracking is stopped. Now let's put everything into a program. In this chapter I shall use the strategy "console" for the projects, because that minimizes the amount of typing. The focus in this chapter is on Prolog, not on the GUI.

So please create a new project. I called it "ch10count". For the UI strategy choose "Console". This means that the program will run in a separate window that looks like like (and behaves like) good old MsDos. In this window you don't have to bother about creating forms and other dialogs as the communication with the program goes line by line. This window is also called the console or the MsDos console. More about the console in chapter 11.

In the project tree create a new class with the name "counter". Put it in the existing package and uncheck the box that says "creates object" as we will only use this class as a module. The IDE opens the files "counter.cl" and "counter.pro" as usual. In the file "counter.cl" enter the declaration for the predicate "count". It should look like:

```
class counter
   open core

predicates
   classInfo : core::classInfo.
   % @short Class information  predicate.
   % @detail This predicate represents information predicate of this class.
   % @end

   count : (integer Number) procedure .
end class counter
```

Then in "counter.pro" add the clauses for the predicate count/1. It should look like:

```
clauses
   classInfo(className, classVersion).

   count(11) :-
      stdIO::write("That's all folks!"), !.
   count(Number) :-
      stdIO::write(Number),
      stdIO::nl,
      NextNumber = Number + 1,
      count(NextNumber).

end implement count
```

I added a line "stdIO::nl" to make the program put every number on a new line. Now we are almost ready to run the program. Because we use the strategy "console" we have to insert the right code into the goal. First build the project and then go to the file "main.pro". There you will find the clause:

```
clauses
   run():-
      console::init(),
      succeed(). % place your own code here
end implement main
```

Change it into:

```
clauses
   run():-
      console::init(),
      counter::count(1),
      _X = stdIO::readChar().

end implement main
```

With the clause

```
      counter::count(1),
```

we start the counting. The clause

```
      _X = stdIO::readChar().
```

Forces the program to wait till you hit <Enter> after it has counted till 10. If you remove this statement, the MsDOS console window will vanish before you can read what is written.

When you run the program, an MsDOS window opens and the program shows you that it can count.

I put the rule to stop the program in a separate clause. There are other possibilities. One could also write:

```
   count(Number) :-
      Number < 11,
      stdIO::write(Number),
      stdIO::nl,
      NextNumber = Number + 1,
      count(NextNumber).
   count(_) :-
      stdIO::write("That's all folks!"), !.
```

Recursion comes in very handy, but at the same time you should be very careful. Imagine what happens when you change the order of the two clauses:

```
   count(Number) :-
      stdIO::write(Number),
      stdIO::nl,
      NextNumber = Number + 1,
      count(NextNumber).
   count(11) :-
      stdIO::write("That's all folks!"), !.
```

In this case the clause “count(11)” is never reached, so the program never stops! Also the place of the write-statement is quite critical. See what happens when we change the program into this one:

```
count(11) :-
    stdIO::write("That's all folks!"), !.
count(Number) :-
    NextNumber = Number + 1,
    count(NextNumber),
    stdIO::write(Number),
    stdIO::nl.
```

The numbers are shown in reversed order. That is because now the program first goes recursively from 1 to 10 and then returning from the calls, starts to write the numbers. So now it first writes the final message, then the number that was generated last, that is 10, then the 9 and so on. Note that when the program calls the recursive predicate, it retains the variables and their values with every call. Each recursive invocation of a clause has it’s own set of variables. Variables are local to the clause where they are used.

In the predicate

```
count(Number) :-
    stdIO::write(Number),
    stdIO::nl,
    NextNumber = Number + 1,
    count(NextNumber).
```

you see that the recursive call is the last statement. That implies that the program does not have to return here when the called predicate has been handled. This is called tail recursion and it is important to look for when you write a program. Tail recursion means that when the program is executed the compiler can replace recursion with iteration. That reduces the needed space for the stack. Recursion leads very easily to a huge number of calls and it saves space when these calls don’t have to remember a return address.

I like to end this section with a classic example, the factorial. The formula n! (Say: n factorial) stands for:

```
    n! = 1 * 2 * 3 * 4 * ...* n
```

A program to calculate the factorial of a number could contain these clauses:

```
fac(1,1) :- !.
fac(Number, Factorial) :-
    NumberMinus1 = Number - 1,
    fac(NumberMinus1, FactorialMinus1),
    Factorial = Number * FactorialMinus1.
```

When the number is 1, the factorial is also 1. There I place a cut, because at this point the program should stop the recursion. Otherwise the next clause will be repeated ad infinitum and Number will

become negative. When the number is greater than one, the factorial of the lesser numbers are calculated and finally the factorial of Number is calculated.

Let's write a program for this. Create a new project (or use the project from the previous section). I created a new program and called it "ch10factorial" with strategy "console". In the project I created a new module (class that does not create objects) "factorial". In "factorial.cl" I declared the predicate "fac".

```
predicates
    classInfo : core::classInfo.
    % @short Class information  predicate.
    % @detail This predicate represents information predicate of this class.
    % @end

    fac : (integer Number, integer Factorial) procedure (i,o).
end class factorial
```

Then in file "factorial.pro" I entered the clauses:

```
clauses
    classInfo(className, classVersion).

    fac(1,1) :- !.
    fac(Number, Factorial) :-
        NumberMinus1 = Number - 1,
        fac(NumberMinus1, FactorialMinus1),
        Factorial = Number * FactorialMinus1.

end implement factorial
```

Then I built the project and finally in the file "main.pro" I changed the "run"-predicate into:

```
clauses
    run():-
        console::init(),
        stdIO::write("Type a number (0 to end) -> "),
        hasdomain(integer, Number),
        Number = stdIO::read(),
        Number <> 0, !,
        factorial::fac(Number, Factorial),
        stdIO::write("Factorial of ", Number, " is ", Factorial),
        stdIO::nl,
        _ = stdIO::readchar(),
        run().
    run().
```

When you run this program it will ask you for a number and then give the factorial of that number. It is not a foolproof program. When you input a negative number or a number that is too large, then things will go wrong. For the arguments of the predicate “fac” we use integers. Integers are restricted in their maximum value, The permitted range is from -2147483648 to 2147483647. When you choose a too large number, on my machine the Number 13 generates already too large factorials, the factorial becomes bigger than the limit and you will get an error message, that will probably vanish from you screen before you can read it.

To make things a bit more interesting we can keep track of what is happening. Let’s introduce a write statement that shows the intermediate steps in the recursion. Please go to “factorial.pro” and change the clause for “fac(Number, Factorial)” into:

```
fac(Number, Factorial) :-
        NumberMinus1 = Number - 1,
        fac(NumberMinus1, FactorialMinus1),
        Factorial = Number * FactorialMinus1,
        stdIO::write("Factorial of ", Number, " is ", Factorial),
        stdIO::nl.
```

When you now run the program it shows you every factorial up to the given number. Notice how fast the factorial grows. It grows faster than squares.

Just as an illustration let us now define a *function* to compute factorials. I give it the name “facfunc” and the declaration that I give in “factorial.cl” looks as follows:

```
   facfunc: (integer Number) -> integer Factorial procedure.
```

In “factorial.pro” enter the clauses for facfunc()

```
clauses
   facfunc(1) = 1 :- !.
   facfunc(Number) = Factorial :-
      NumberMinus1 = Number - 1,
      FactorialMinus1 = facfunc(NumberMinus1),
      Factorial = Number * FactorialMinus1,
      stdIO::write("Factorial of ", Number, " is ", Factorial),
      stdIO::nl.
```

And finally change the code for the “run” predicate in “main.pro” into:

```
clauses
   run():-
      console::init(),
      stdIO::write("Type a number (0 to end) -> "),
      hasdomain(integer, Number),
      Number = stdIO::read(),
      Number <> 0, !,
```

```
    stdIO::write("Factorial by predicate"), stdIO::nl,
    factorial::facpred(Number, Factorial),
    stdIO::write("Factorial of ", Number, " is ", Factorial),
    stdIO::nl,
    stdIO::write("hit <Enter>"),
    _ = stdIO::readchar(),
    _ = stdIO::readchar(),
    stdIO::write("Factorial by function"), stdIO::nl,
    FuncFactorial = factorial::facfunc(Number),
    stdIO::write("Factorial of ", Number, " is ", FuncFactorial),
    stdIO::nl,
    run().
run().
```

This last piece of code is quite horrible, so you'd better copy and paste it from the pdf file of the book.

Now build and run the program. The results of both the predicate and the function are the same, but they tackle the problem in different ways. It is up to you to decide which one you like best.

10.2 Lists[13].

A list is a sequence of elements. Lists and list processing is a powerful technique in Prolog. In this section we explain what lists are and how to declare them, and then give several examples that show how to do list processing in your own applications. We also define several well known Prolog predicates and take a look at list processing from both a recursive and a procedural standpoint. In the end we look at compound lists, that is a list with different types of elements

What Is a List?
In Prolog, a list is an object that contains an arbitrary number of other objects. Lists correspond roughly to arrays in other languages, but, unlike an array, a list does not require you to declare how big it will be before you use it. Not does it have indices to point to elements.

A list that contains the numbers 1, 2, and 3 is written as

```
[ 1, 2, 3 ]
```

The order of the elements in this list matters: number "1" is the first element, "2" - the second, "3" - the third. List [1, 2, 3] is different from the list [1, 3, 2].
Each item contained in the list is known as an element. To form a list data structure, you separate the elements of a list with commas and then enclose them in square brackets. Here are some other examples:

```
["dog", "cat", "canary"]
["valerie ann", "jennifer caitlin", "benjamin thomas"]
```

[13]This section is based on a tutorial by Thomas Linder Puls (PDC Prolog)

The same element can be present in the list several times, for example:

[1, 2, 1, 3, 1]

A list can contain any (positive) number of elements. A special case is the list that has no elements. It is called the empty list and it is represented by [].

Declaring Lists
To declare the domain for a list of integers, you use the domains declaration, like this:

```
domains
   integer_list = integer*.
```

The asterisk "*" means "list of"; that is, integer* means "list of integers."
Note that the word "list" has no special meaning in Visual Prolog. You could equally well have called your list domain "zanzibar". It is the asterisk, not the name that signifies a list domain.
The elements in a list can be anything, including other lists. However, all elements in a list must belong to the same domain. When the elements in the list are not the standard types (integer, real, string) then in addition to the declaration of the list domain, there must be a **domains** declaration for the elements:

```
domains
   element_list = elements*.
   elements = string
or
domains
        integerlist = integer*.
```

Here *elements* must be equated to a single domain type (for example, integer, real, or string) or to a set of alternatives marked with different functors. Visual Prolog does not allow you to mix standard types in a list. For example, the following declarations would not properly indicate a list made up of integers, reals, and strings:

```
element_list = elements*.
elements =  integer;  real; string.       /* Incorrect */
```

The way to declare a list made up of integers, reals, and strings is to define a single domain comprising all three types, with functors to show which type a particular element belongs to. For example:

```
element_list = elements*.
elements =   i(integer);  r(real); s(string).       /* the functors are i, r, and s */
```

For more information about this, refer to "Compound Lists" later in this chapter.

Heads and Tails

A list is really a recursive compound object. It consists of two parts: the head of a list, which is the first element, and the tail, which is a list comprising all the subsequent elements.

The tail of a list is always a list; the head of a list is always an element from that list.

For example,

the head of [a, b, c] is the element a
the tail of [a, b, c] is the list [b, c]

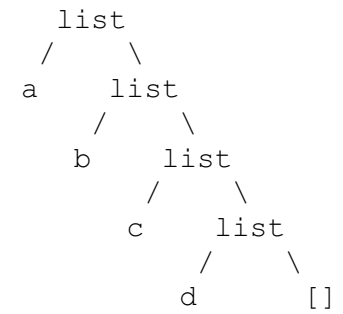
What happens when you get down to a one-element list? The answer is that:

the head of [c] is the element c
the tail of [c] is the list []. The empty list - still a list, but empty.

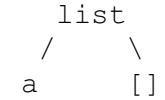
If you take the first element from the tail of a list enough times, you will eventually get down to an empty list ([ ]).

The empty list cannot be broken into head and tail.

This means that, conceptually, lists have a tree structure just like other compound objects. The tree structure of [a, b, c, d] is:



An explanation of this structure is that it says that the list [a,b,c,d] consists of the element a and the list [b,c,d]. The list [b,c,d] consists of the element b and the list [c,d]. Et cetera. Further, a one-element list such as [a] is not the same as the element that it contains, because [a] is really the compound data structure shown here:



List Processing

Prolog provides a way to make a head and a tail of a list explicit. Instead of separating elements with commas, you can separate the head and tail with a vertical bar "|". For instance,

[a, b, c] is equivalent to [a | [b, c]]

In this way you can split up the list in the head and the tail. This process can be continued,

[a|[b,c]] is equivalent to [a | [b | [c]]]
which is equivalent to [a | [b | [c | []]]]

You can even use both kinds of separators in the same list, provided the vertical bar is the last separator. So, if you really want to, you can write [a, b, c, d] as [a, b|[c, d]]. Here you make a head that consists of two elements, but we don't call that a head. Table 1 gives more examples.

List	Head	Tail
['a', 'b', 'c']	'a'	['b', 'c']
['a']	'a'	[] /* an empty list */
[ ]	undefined	undefined
[[1, 2, 3], [2, 3, 4], []]	[1, 2, 3]	[[2, 3, 4], []]

Table 1: Heads and Tails of Lists

Table 2 gives several examples of list unification.

List1	List2	Variable Binding for the clause: List1 = List2
[X, Y, Z]	[egbert, eats, icecream]	X=egbert, Y=eats, Z=icecream]
[7]	[X | Y]	X=7, Y=[]
[1, 2, 3, 4]	[X, Y | Z]	X=1, Y=2, Z=[3,4]
[1, 2]	[3 | X]	fail

Table 2: Unification of Lists

10.3 Lists and recursion[14]

Because a list is really a recursive compound data structure, you need recursive algorithms to process it. The most basic way to process a list is to work through it, doing something to each element until you reach the end.
An algorithm of this kind usually needs two clauses. One of them takes the head of a list and tells what to do with it. This statement is repeated till you reach an empty list. The other statements tells what to do with that empty list. I hope that you get the feeling that recursion can come in handy here. In this and the next section some things that were said in section 10.1 will be repeated. If you find that boring, skip it.

In this and the next section a number of small programs are presented. I take it for granted that by now you know how to create this program in the IDE. To remind you, these are the steps:
1. Create a new project <ProjectName> with strategy "console".
2. Add a class <ClassName> where the clauses (the code) can reside. Uncheck "creates objects".

[14]This section is from the same tutorial as section 10.2

3. Enter the clauses (the definition) for the predicate(s) in the file <ClassName>.pro
4. Enter the declaration of the predicate(s) in the file <ClassName>.cl
5. Change the “run”-predicate in the file “main.pro” so that the predicate in <ClassName>.pro is called.
6. If necessary, add the needed write-statements. Don’t forget the statement
   _ = stdIO::readChar()
   at the end, otherwise you won’t be able to read the results.

Writing Lists

If you just want to print the elements of a list, here is a program that does the job.

```
This is the declaration
predicates
   write_a_list : (integer*).
end class

These are the clauses

clauses
   write_a_list([]).              /* If the list is empty, do nothing more. */
   write_a_list([H|T]):-          /* Match the head to H and the tail to T, then... */
      stdio::write(H),stdio::nl,  /* Write the element that is the head of the list ... */
      write_a_list(T).            /* ... and continue with the tail */
end implement

run
   console::init(),
   writelist::write_a_list([1, 2, 3])          /* I gave the class the name “writelist” */
   _ = stdIO::readChar().
```

The program neatly writes the elements of the list to the console. Here are the two write_a_list clauses described in natural language:

The first says: To write an empty list, do nothing.
The second one says: Otherwise, to write a list, write its head (which is a single element), then write its tail (a list).

And here is how it works:
The first time through, the goal is:

```
writelist::write_a_list([1, 2, 3]).
```

This matches the second clause, with H=1 and T=[2, 3]; this writes “1" and then calls write_a_list recursively with the tail of the list:

```
writelist::write_a_list([2, 3]).        /* This is write_a_list(T). */
```

This recursive call matches the second clause, this time with H=2 and T=[3], so it writes “2" and again calls write_a_list recursively:

```
writelist::write_a_list([3]).
```

Now, which clause will this goal match? Recall that, even though the list [3] has only one element; it does have a head and tail; the head is 3 and the tail is the empty list []. So, again the goal matches the second clause, with H=3 and T=[]. Hence, “3" is written and write_a_list is called recursively like this:

```
writelist::write_a_list([]).
```

Now you see why this program needs the first clause. The second clause will not match this goal because [] cannot be divided into head and tail. So, if the first clause were not there, the goal would fail. As it is, the first clause matches and the goal succeeds without doing anything further.

To see if you understand what happens try to imagine what the result will be if you change the second clause into:

```
write_a_list([H|T]):-
   write_a_list(T),
   stdio::write(H),stdio::nl.
```

If you are not sure, go ahead and change the program!

Counting List Elements

Now consider how you might find out how many elements there are in a list. We call the number of elements the length of the list. How can you find the length of a list? The easiest way is to count the elements. That gives a simple algorithm:

1. The length of [] is 0.
2. When a list is not empty, we can represent it with [Head | Tail]. The length of this list is 1 plus the length of Tail.

Can you implement this? In Prolog it is very easy. It takes just two clauses:

```
predicates
   length_of : (integer* List, integer Length) procedure(i,o).
end class

implement listlength
clauses
   length_of([], 0).
   length_of( [ _ | Tail], Length):-
      length_of(Tail, TailLength),
      Length = TailLength + 1.
end implement

run
   console::init(),
   listLength::length_of([1, 2, 3], Length),          /* I called the class “listLength” */
   stdio::write(“the length of the list is: “,Length),
   _ = StdIO::readchar().
```

Take a look at the second clause first. Crucially, [_ | Tail] will match any nonempty list, binding Tail to the tail of the list. The value of the head is unimportant; as long as it exists, it can be counted as one element.
So the goal:

listlength::length_of([1, 2, 3], Length)

will match the second clause, with Tail=[2, 3]. The next step is to compute the length of Tail. When this is done (never mind how), TailLength will get the value 2, and the computer can then add 1 to it for the Head and bind Length to 3.
So how is the middle step executed? That step was to find the length of [2, 3] by satisfying the subgoal

listlength::length_of([2, 3], TailLength)

In other words, length_of calls itself recursively. This goal matches the second clause, binding [3] in the goal to Tail in the clause and TailLength in the goal to Length in the clause.

Recall that TailLength in the goal will not interfere with TailLength in the clause, because ***each recursive invocation of a clause has its own set of variables***. So now the problem is to find the length of [3], which will be 1, and then add 1 to that to get the length of [2, 3], which will be 2. So far, so good.
Likewise, length_of will call itself recursively again to get the length of [3]. The tail of [3] is [], so T is bound to [], and the problem is to get the length of [], then add 1 to it, giving the length of [3]. This time it's easy. The goal:

listLength::length_of([], TailLength)

matches the *first* clause, binding TailLength to 0. So now the computer can add 1 to that, giving the length of [3], and return to the calling clause. This, in turn, will add 1 again, giving the length of [2, 3], and return to the clause that called it; this original clause will add 1 again, giving the length of [1, 2, 3].
Confused yet? I hope not. In the following brief illustration we'll summarize the calls. We've used subscripts to indicate that similarly named variables in different clauses – or different invocations of the same clause – are distinct.

Listlength::length_of([1, 2, 3], $Length_1$).
Listlength::length_of([2, 3], $Length_2$).
Listlength::length_of([3], $Length_3$).
Listlength::length_of([], 0).
$Length_3$ = 0+1 = 1.
$Length_2$ = $Length_3$+1 = 2.
$Length_1$ = $Length_2$+1 = 3.

Tail Recursion

You probably noticed that length_of is not, and can't be, tail-recursive, because the recursive call is not the last step in its clause. Can you create a tail-recursive list-length predicate? Yes, but it will take some effort.
The problem with length_of is that you can't compute the length of a list until you've already computed the length of the tail. It turns out there's a way around this. You'll need a list-length predicate with three arguments.

Argument one is the list, which the computer will whittle away on each call until it eventually becomes empty, just as before. The second argument is a free argument that will ultimately contain the result (the length). The third argument is a counter that starts out as 0 and increments on each call. When the list is finally empty, you'll unify the counter with the (up to then) unbound result. What this predicate is doing is: it takes each element from the list and counts it when it is removed.

```
predicates
   length_of : (integer* List, integer Length, integer Counter) procedure(i,o,i).
end class

clauses
   length_of([], Result, Result).
   length_of([_|T], Result, Counter):-
      NewCounter = Counter + 1,
      length_of(T, Result, NewCounter).
end implement

run
   console::init(),
   listlength::length_of([1, 2, 3], L, 0),
      /* start with Counter = 0 */
   stdio::write(" L = ", L),
   _ = stdIO::readchar().
```

This version of the length_of predicate is more complicated, and in many ways less logical, than the previous one. We've presented it merely to show you that, by devious means, ***you can often find a tail-recursive algorithm for a problem that seems to demand a different type of recursion***.

Modifying the List

Sometimes you want to take a list and create another list from it. You do this by working through the list element by element, replacing each element with a computed value. For example, here is a program that takes a list of numbers and adds 1 to each of them:

```
predicates
   add1 : (integer*, integer*) procedure(i,o).
end class

clauses
   add1([], []).                              /* boundary condition */
   add1([Head|Tail],[NewHead|NewTail]):-  /* separate the head from the rest of the list */
   NewHead = Head+1,                          /* add 1 to the first element */
   add1(Tail, NewTail).                        /* call element with the rest of the list */
end implement

run
   console::init(),
   listmanager::add1([1,2,3,4], NewList),            /* I called the class “listmanager” */
```

```
   stdio::write("Newlist = ", NewList)),
   stdIO::readchar().
```

To paraphrase this in natural language: To add 1 to all the elements of the empty list, just produce another empty list.
To add 1 to all the elements of any other list, add 1 to the head and make it the head of the result, and then add 1 to each element of the tail and make that the tail of the result. Load the program, and run the goal with the specified goal

```
add1([1,2,3,4], NewList).
```

The goal will return

```
NewList = [2,3,4,5]
```

Tail Recursion Again

Is add1 tail-recursive? If you're accustomed to using Lisp or Pascal, you might think it isn't, because you think of it as performing the following operations:

- Split the list into *Head* and *Tail*.
- Add 1 to *Head*, giving *NewHead*.
- Recursively add 1 to all the elements of *Tail*, giving *NewTail*.
- Combine *NewHead* and *NewTail*, giving the resulting list.

This isn't tail-recursive, because the recursive call is not the last step.
But – and this is important – ***that is not how Prolog does it***. In Visual Prolog, add1 is tail-recursive, because its steps are really the following:

- Bind the head and tail of the original list to *Head* and *Tail*.
- Bind the head and tail of the result to *NewHead* and *NewTail*. (*NewHead* and *NewTail* do not have values yet.)
- Add 1 to *Head*, giving *NewHead*.
- Recursively add 1 to all the elements of *Tail*, giving *NewTail*.

When this is done, *NewHead* and *NewTail* are *already* the head and tail of the result; there is no separate operation of combining them. So the recursive call really is the last step.

More on Modifying Lists

Of course, you don't actually need to put in a replacement for every element. Here's a program that scans a list of numbers and copies it, leaving out the negative numbers. It uses a module called "listmanager".

```
predicates
   discard_negatives : (integer*, integer*) procedure(i,o).
end class

clauses
   discard_negatives([], []).
   discard_negatives([H|T], ProcessedTail):-
      H < 0, !,    /* If H is negative, just skip it, the processed tail will not be combined with H */
      discard_negatives(T, ProcessedTail).
```

```
   discard_negatives([H|T], [H|ProcessedTail]):-   /* here H will be added to the processed Tail */
      discard_negatives(T, ProcessedTail).
end implement listmanager

run
   console::init(),
   listmanager::discard_negatives ([2, -45, 3, 468], X),
   stdio::write(X)
   StdIO::readchar().
```

For example, the goal

```
listmanager::discard_negatives([2, -45, 3, 468], X)
```

gives

```
[2, 3, 468].
```

And here's a predicate that copies the elements of a list, making each element occur twice:

```
doubletalk([], []).
doubletalk([H|T], [H, H|DoubledTail]) :-
   doubletalk(T, DoubledTail).
```

List Membership

Suppose you have a list with the names *John*, *Leonard*, *Eric*, and *Frank* and would like to use Visual Prolog to investigate if a given name is in this list. In other words, you must express the relation "membership" between two arguments: a name and a list of names. This corresponds to the predicate

```
isMember : (name, name*).
        % "name" is a member of "name*"
        %  this means, that the list name* consists of members of the type name
```

In the program below, the first clause investigates the head of the list. If the head of the list is equal to the name you're searching for, then you can conclude that Name is a member of the list. Since the tail of the list is of no interest, it is indicated by the anonymous variable. Thanks to this first clause, the goal

```
listMan::isMember("john", ["john", "leonard", "eric", "frank"])
```

is satisfied. Here is the program

```
class listMan

predicates
  isMember : (string, string*) determ.
end class

implement listMan
clauses
  isMember(Name, [Name|_]) :-
    !.
  isMember(Name, [_|Tail]):-
```

```
    isMember(Name,Tail).
end implement

clauses
  run():-
    console::init(),
    listMan::isMember("john", ["john", "leonard", "eric", "frank"]),
    !,
    stdio::write("Success"),   _ = stdIO::readChar()
    ;
    stdio::write("No solution"),  _ = stdIO::readChar().
```

If the head of the list is not equal to Name, you need to investigate whether Name can be found in the tail of the list.
In English:

- Name is a member of the list if Name is the first element of the list, or
- Name is a member of the list if Name is a member of the tail.

The second clause of isMember relates to this relationship. In Visual Prolog:

```
isMember(Name, [ _ | Tail]) :-
   isMember(Name, Tail).
```

Appending One List to Another: Declarative and Procedural Programming

As given, the isMember predicate of the previous program works in two ways. Consider its clauses once again:

```
isMember(Name, [Name|_]).
IsMember(Name, [_|Tail]) :-
   isMember(Name, Tail).
```

You can look at these clauses from two different points of view: declarative and procedural.

From a declarative viewpoint, the clauses say:

- Name is a member of a list if the head is equal to Name;
- if not, Name is a member of the list if it is a member of the tail.

From a procedural viewpoint, the two clauses could be interpreted as saying:

- To find a member of a list, find its head;
- otherwise, find a member of its tail.

These two points of view correspond to the goals

```
isMember(2, [1, 2, 3, 4]).
```

and

```
isMember(X, [1, 2, 3, 4]).
```

In effect, the first goal asks Visual Prolog to check whether something is true; the second asks Visual Prolog to find all members of the list [1,2,3,4]. Don't be confused by this. The member predicate is the same in both cases, but its behavior may be viewed from different angles.

Recursion from a Procedural Viewpoint

The beauty of Prolog is that, often, when you construct the clauses for a predicate from one point of view, they'll also work from the other point of view. To see this duality, in this next example you'll construct a predicate to append one list to another. You'll define the predicate append with three arguments:

```
append(List1, List2, List3).
```

This combines List1 and List2 to form List3. Once again you are using recursion (this time from a procedural point of view).

If List1 is empty, the result of appending List1 and List2 will be the same as List2. In Prolog:

```
append([], List2, List2).
```

If List1 is not empty, you can combine List1 and List2 to form List3 by making the head of List1 the head of List3. (In the following code, the variable H is used as the Head of both List1 and List3.) The tail of List3 is Tail3, which is composed of the rest of List1 (namely, Tail1) and all of List2. In Prolog:

```
append([H|Tail1], List2, [H|Tail3]) :-
  append(Tail1, List2, Tail3).
```

The append predicate operates as follows: While List1 is not empty, the recursive rule transfers one element at a time to List3. When List1 is empty, the first clause ensures that List2 hooks onto the back of List3.

One Predicate Can Have Different Uses

Looking at append from a declarative point of view, you have defined a relation between three lists. This relation also holds if List1 and List3 are known but List2 isn't. However, it also holds true if only List3 is known. For example, to find which two lists could be appended to form a known list, you could use a goal of the form

```
append(L1, L2, [1, 2, 4]).
```

With this goal, Visual Prolog will find these solutions:

```
L1=[], L2=[1,2,4]
L1=[1], L2=[2,4]
L1=[1,2], L2=[4]
L1=[1,2,4], L2=[]
4 Solutions
```

You can also use append to find which list you could append to [3,4] to form the list [1,2,3,4]. Try giving the goal

```
append(L1, [3,4], [1,2,3,4]).
```

The inference engine finds the solution

```
L1=[1,2].
```

This append predicate has defined a relation between an *input set* and an *output set* in such a way that the relation applies both ways. Given that relation, you can ask

Which output corresponds to this given input?

or
Which input corresponds to this given output?

The status of the arguments to a given predicate when you call that predicate is referred to as a *flow pattern*. An argument that is bound or instantiated at the time of the call is an input argument, signified by (i); a free argument is an output argument, signified by (o).

The append predicate has the ability to handle any flow pattern you provide. However, not all predicates have the capability of being called with different flow patterns. When a Prolog clause is able to handle multiple flow patterns, it is known as an invertible clause.

10.4 Special list predicates

In the previous sections we created and specified every predicate that we needed. That is not necessary. As said before there is a class called “list” in which you find a lot of handy predicates to manipulate lists. Take a look in the help file, find “list” and look in “class list”.

In this section I should like to introduce to you three predicates that are often used with lists but that are standard predicates in Prolog. You don’t have to use the class “list” or any other class as you can call these standard predicates from anywhere in a program.

Backtracking and recursion are two ways to perform repetitive processes. Recursion unlike backtracking can pass information (through arguments) from one recursive call to the next. Because of this, a recursive procedure can keep track of partial results or counters as it goes along.
But there's one thing backtracking can do that recursion can't do – namely, find all the alternative solutions to a goal. But it finds them one by one, so you may find yourself in a quandary. When you need all the solutions to a goal, but you need them all at once, as part of a single compound data structure. What do you do?

Fortunately, Visual Prolog provides a way out of this impasse. You have already met the predicate findall/3. It is a standard predicate with the following call-template:

```
findall ( ArgumentName, predicateCall, ValuesList )
```

Findall/3 is a procedure with flow (i,i,o). It’s goal is to assemble a list of all solutions returned by a non-deterministic predicate. You offer as an input the first two arguments and findall/3 returns in the third argument the wanted list. The arguments are:

- ArgumentName
  Specifies which argument in the specified predicate *predicateCall* is to be collected into a list ValuesList.
- predicateCall
  Indicates the predicate from which values will be collected.
- ValuesList
  Is an output variable that holds the list of values collected through backtracking.

As an example think of a facts database with the functor

```
item(string ItemName, string ItemColor).
```

It could be the item list of an inventory. Each fact contains the name of an item and its color. Now when you want to have a list of all the items, then the call

findall(ItemName, item(ItemName, _), NameList).

returns in the variable NameList a list of all the items. If you only want to have a list of the, say, red items, then the call is:

findall(ItemName, item(ItemName, “red”), NameList).

Finally, if you want to see a list of the colors of the coats in your inventory, then the call should be:

findall(Color, item(“coat”, Color), ColorList).

Findall/3 returns every value that it finds. So most of the time there are multiples of the same color. But in the class “list” you will find a predicate to remove duplicates when that is necessary.

Some remarks that may be useful when you use findall/3. The findall predicate builds the list ValuesList by backtracking into the predicate predicateCall until it eventually fails. ValuesList consists of all instances of the variable ArgumentName, which must be an output argument of the specified predicate predicateCall. In order for findall to work, there must be a declaration of the list domain for ValuesList in the user domain declarations. That means that when e.g. you want a colorlist, this must have been declared beforehand.

Findall/3 generates a list alright, but there is a more general way to generate lists. It is called list comprehension. Let’s first take a formal look at list comprehension. It is a function that is called in the following way:

ResultList = [Exp || Gen]

Mind the double “||” in the expression on the right side! List comprehension is a list expression. That means that the list comprehension call is in fact an expression and as such it must be bound to a variable. Now take a closer look at [Exp || Gen]. In this expression Gen is (typically) a nondeterm term. That means when you call Gen it can fail and backtrack. You may think of Gen as a functor that is called and then generates values as solutions. Compare it to the “item” functor above. Exp is evaluated for each solution of Gen, and the resulting Exp's are collected in a list. The Exp corresponding to the first solution of Gen is the first element in the list, etc. This list is the result of the list comprehension term. On return the list is bound to ResultList. Exp must be of mode procedure (or erroneous). Both Exp and Gen are cut scopes. You can paraphrase the list comprehension as: Give me the list of Exp such that Gen.

Now let’s go to an example. Say there is a list L that contains numbers. If we want to retrieve the even numbers in this list, we could call:

EvenNumberList = [X || isMember(X, L), X div 2 = 0]

In plain English this says: “put in EvenNumberlist the list of X’s that are in list L and that are even”. It is comparable to findall/3, but in this case we have added an extra condition. Another example of the use if list comprehension is:

OddNumberList = [X + 1 || isMember(X, L), X div 2 = 0]

Here the collected expression is more complex. In English one could say: “put in OddNumberlist the list of X’s that are in list L and that are even, but before putting a number in the list, increment it with

1". So you get a list of odd numbers. (Not the odd numbers in the list!). If you want you can get the same list with

```
OddNumberList = [ Y || isMember(X, L), X div 2 = 0 , Y = X+1 ]
```

In English: "The list of Y's such that (there exists an) X which is member of L, and which is even, and Y equals X+1."

List comprehension is a generalization of findall/3. Consider the schematic findall term:

```
findall(V, Gen, L)
```

This is equivalent to writing

```
L = [ V || Gen ]
```

The main differences between findall and the list comprehension is that findall is a predicate that binds the result to a variable, whereas the list comprehension is an expression.

Now let's use list comprehension in a program. To repeat: when we say:

```
ListParam = [ VarName || mypredicate]
```

then it means:

- The first argument, VarName, specifies which argument in the specified predicate is to be collected into a list.
- The second argument, mypredicate, indicates the predicate from which the values will be collected.
- The output ListParam, is a variable that holds the list of values collected through backtracking.

The next program uses list comprehension to print the average age of a group of people.

```
class listcom
domains
   name = string.
   address = string.
   age = integer.

predicates
   person : (name, address, age) multi (o,o,o).
   sumlist : (age*, age, integer)  procedure(i,o,o).
end class listcom

implement listcom
clauses
   sumlist([],0,0).
   sumlist([H|T], Sum, N):-
      sumlist(T, S1, N1),
      Sum=H+S1, N=1+N1.
   person("Sherlock Holmes",  "22B Baker Street", 42).
   person("Pete Spiers",  "Apt. 22, 21st Street", 36).
   person("Mary Darrow",  "Suite 2, Omega Home", 51).
end implement listcom
```

```
in main.pro:
   run():-
      console::init(),
      L = [ Age || listcom::person(_, _, Age)],
      listcom::sumlist(L, Sum, N),
      Ave = Sum/N,
      stdio::write("Average=", Ave, ". "),
      _X = stdIO::readchar().
```

The list comprehension clause in this program creates a list L, which is a collection of all the ages obtained from the predicate person. If you wanted to collect a list of all the people who are 42 years old, you could give the following subgoal:

```
List = [ Who || my::person(Who, _, 42) ]
```

The following code will create the list of positive items:

```
List = [ X || X = list::getMember_nd([2,-8,-3,6]), X > 0]
```

where getMember_nd is a predicate from the class “list” to retrieve the members of a list.

Finally I want you to meet another construction. Remember that we used to print a list of numbers with a predicate that looked like:

```
write_a_list(List) :-
   member(Element, List),
   write(Element), nl()
  fail.
Write_a_list(_).
```

This writes every Element from the List, or, better said, it writes each Element that is generated as a solution by the nondeterministic predicate “member()”. To generate a complete list, we make the predicate fail. That is not really elegant, as in this case printing is what you want, so the writing succeeds by failing. And you need an extra clause to prevent the program from failing. In version 7 of VIP the foreach construction is introduced. It looks like this:

```
…,
foreach nondeterm_predicate(X) do
  write(X), nl()
end foreach,
…
```

The foreach construction is intended as a replacement for fail loops. In our example it could look like:

```
Write_a_list(List) :-
      foreach member(X,List)
```

```
do write(X), nl()
end foreach.
```

This predicate writes the list and succeeds. So no extra clause is needed. In this case you don't even need an special predicate, the for each construction can be inserted as a subgoal in another predicate:

```
…,
foreach nondeterm_predicate(X) do
  Process(X),
end foreach,
…
```

The "foreach" construction ***succeeds,*** when the iteration is completed. So computation will continue ***right after*** the "foreach" construction. In the clauses with fail you had to take precautions to see that the program continued rightly after the fail. Foreach takes care of that. Look at this piece of program, it continues after the printing is done.

```
…,
foreach p_nd(X)
do write(X), nl()
end foreach,
write("I will continue here"),
...
```

As a result foreach-terms can be followed by other terms and they can be properly nested.
Example

```
foreach member(L, LL) do
   write("<< "),
   foreach  member(X, L) do
      write(" ", X)
   end foreach,
   write(" >>\n")
end foreach.
```

LL is supposed to be a list of lists. The outer foreach-loop generates lists from the list_of_lists each L is a list. The inner foreach-loop iterates X through the elements of L and writes the elements..

There are a number of things to notice:

- There is no comma before the keywords "do" and "end foreach".
- The body of a foreach-term is a cut scope, so a cut met in the body will not influence the iteration.
- A foreach-term always succeeds (or raises an exception), and no extra variables are bound after the evaluation of a foreach-term. Therefore, a foreach-term can only be used for its side effects.

I realize that there may be some abracadabra in these last sentences. Sorry for that. Just keep in mind that foreach is a simple but powerful construction to proces the members of a list.

10.5 Sorting

In the previous section we said that the order of the elements in a list matters. That brings us to the subject of sorting. If you are not interested in sorting, then you can skip this section. There is a standard predicate available for sorting. But sorting a list is interesting because it gives you further insight into lists and recursion. That's why I included this section in this book.

There are many ways to sort a list. In this section I shall show you four ways. Three of them, insertSort, selectSort and bubbleSort, are just for fun, because they are not very efficient. The fourth way, quickSort is a very famous one that you have to know, I think. Another restriction is that I shall only use lists of integers, but that should not be a real restriction. I created a program with the name "ch10SortList" with strategy "console" for sorting. In the project I created a class without objects (a module) with the name "sort". This module contains all the predicates. The declarations are, as usual, in "sort.cl" and the clauses for the predicates are in "sort.pro".

The first method takes the elements of an unsorted list and one by one puts them on the right place in a new list. When all the elements have been put in the new list, that list will be sorted. In Prolog we can do that by taking the Head of the unsorted list, put it at the right place in a list that was empty at the start and then recursively take the next element, that is the first element (indeed, the Head) of the Tail of the unsorted list. This way of sorting is called "insert sorting". I shall use the predicate "insertSort". It clearly has two arguments: it needs an unsorted list as input and then returns as output the sorted list. So my declaration becomes:

```
insertsort : (integer*, integer*) procedure (i,o).
```

There are two clauses for this predicate. The first one is the sorting of an empty list.

```
insertSort([], []).
```

That's an easy one: when you sort an empty list you get a (sorted) empty list. An alternative could be the (trivial) sorting of a list with one element:

```
insertSort([X], [X]).
```

When there are more than one elements in the list, we can (in words) sort the list by taking the following steps:

1. remove the Head of the unsorted list
2. sort the Tail
3. insert the Head at the right place in the sorted Tail.

In Prolog it looks like this

```
insertSort([], []) :- !.
insertSort([Head | Tail], SortedList) :-      % take an unsorted list and produce the sorted list
    insertSort( Tail, SortedTail),            % sort the tail
    insert(Head, SortedTail, SortedList), !.  %insert the Head at the right place in the sorted Tail
```

By the way, I like to use long names for variables, so it will be clear what they represent when you read the program. That's why I use "SortedList" and "SortedTail" as names. But if you want, you can of course use names like L1 or L2.

Next we need a predicate to insert an element. I use the predicate "insert" with the declaration:

```
insert : (integer, integer*, integer*) procedure (i,i,o).
```

This predicate uses three arguments. The first is the element that must be inserted, the second is the list into which the element is inserted and the third is the resulting list. This predicate is also a recursive predicate. So I start with a simple stopping rule to insert an element into an empty list:

```
insert(Element, [], [Element]) :- !.
```

What to do when the list is not an empty list? Well, then it depends on the Element_to_insert and the first element, the Head of the list. Remember that the list into which we insert the Element is already sorted. So when Element is less than or equal to Head, then it should be placed in front of the Head. In Prolog:

```
insert(Element, [Head | Tail], [Element, Head | Tail]) :-
    Element <= Head, !.
```

Here I use the template [X,Y | Tail] to indicate the position of the first two elements in list. When Element is larger than Head, then Head should be skipped and the next element in the Tail should be considered. In Prolog:

```
insert(Element, [Head | Tail], [Head | TailWithElement] ) :-
    !,
    insert(Element, Tail, TailWithElement).
```

This completes the program. To run it you need to adjust the run/0 predicate in the main program. It should become:

```
run():-
    console::init(),
    UnsortedList = [4,2,5,3,1,2,6,4],
    stdIO::write("The unsorted list is: ", UnsortedList),
    stdIO::nl,
    sort::insertsort(UnsortedList, InSortedList),
    stdIO::write("The sorted list (insert sort) is: ",InsortedList),
    stdIO::nl,
    _X = stdIO::readchar(),
    succeed().
```

A second way to sort a list is called selection sorting. The algorithm is:

1. Take an unsorted list
2. Find the smallest element and take it out.
3. Sort the rest of the list
4. Put the element_taken_out in front of the sorted rest of the list.

Let's make a predicate for it. Just like we did with insert sort, we create a predicate that accepts an unsorted list and produces a sorted list. I declared the predicate "selectSort"

```
selectsort : (integer*, integer*) procedure (i,o).
```

As selection sort is clearly a recursive way to sort, I start with a clause to describe the sorting of an empty list:

```
    selectsort([],[]):- !.
```

When the list_to_be_sorted is not empty, we

1 take the smallest element out
2 sort the remaining elements and
3 put the smallest element in front of the sorted list.

In Prolog:

```
selectsort(UnsortedList, [SmallestElement | SortedRestOfList]) :-
    smallest(UnsortedList, SmallestElement),          % find the smallest element

    remove(SmallestElement, UnsortedList, UnsortedRestOfList), !,
                                                      % take the smallest element out
    selectSort(UnsortedRestOfList, SortedRestOfList).  % sort the rest of the list
```

To find the smallest element in a list, I use the recursive predicate “smallest”. The declaration is:

```
smallest : (integer*, integer) procedure (i,o).
```

It takes a list and returns the smallest element. I use three clauses. The first is when the list consists of one element:

```
smallest([Element], Element).
```

When there are more elements, I take the Head of the list, then find the smallest element in the Tail and compare the Head with it. When the Head is smaller, it clearly is the smallest element, if not, the other one is the smallest. In Prolog:

```
smallest([Head | Tail], Head) :-
    smallest(Tail, Element),
    Head <= Element, !.
smallest([_ | Tail], Element) :-
    smallest(Tail, Element), !.
smallest(_,0).
```

The last clause is a catchall. Probably I’m doing something wrong, because that clause will never be reached. But when I leave it out, the compiler complains that smallest/2 is not a procedure.

Finally we must remove the smallest element from the list. I use the predicate remove/3 with declaration:

```
remove : (integer, integer*, integer*) procedure (i,i,o).
```

And clauses

```
remove(Element, [Element | Tail], Tail) :- !.
remove(Elem1, [Elem2 | Tail1], [Elem2 | Tail2]) :-
    remove(Elem1, Tail1, Tail2), !.
remove (_,_,[0]).
```

If the element_to_be_removed is the first one, then the resulting list equals the Tail. If not, we must remove it from the Tail

Finally we must adjust the run/0 predicate. Add the clauses to produce the select sorting:

```
    sort::selectsort(UnsortedList, SelSortedList),
    stdIO::write("The sorted list (select sort) is: ",SelsortedList),
    stdIO::nl,
```

The third sorting technique I want to show is called “bubble sort”. It is performed by taking the following steps:

1. Take the first two elements of an unsorted list.
2. Compare them. If the first element is smaller than the second, do nothing. If the second element is smaller than the first one, change their places.
3. Take the right (right means here: not the left one) element and compare it to its neighbor on the right side.
4. If the element is smaller than its neighbor, do nothing. If neighbor is smaller than element, change their places.
5. Go to step 3 until you compared every element with its neighbor.
6. Goto step 1 and repeat the process until there are no more changes of places. Then the list is sorted.

In this way a big element will move from left to right like a bubble goes up in the water. I think that explains the name.

I will use the code proposed by Ivan Bratko. It is quite compact - it took me some time to understand what is happening and how the program works :-). As usual I start with the predicate that is called to perform the sort. I call it “bubbleSort” and the declaration is:

```
bubblesort : (integer*, integer*) procedure (i,o).
```

Bubblesort/2 takes the list and calls bubble/2. In bubble/2 we decide if there has to be a change. If so, bubble/2 changes the place of two elements and starts bubblesorting the new list. When there is no change, then clearly the whole list is sorted. Bubble/2 fails and the second clause for bubblesort/2 sees to it that the sortedlist is returned to the calling predicate.

```
bubbleSort(List, SortedList) :-
    bubble(List, List1), !,
    bubbleSort(List1, SortedList).
bubbleSort(SortedList, SortedList).
```

So the trick is in the predicate bubble/2. The declaration is:

```
bubble : (integer*, integer*) determ (i,o).
```

The clauses are:

```
bubble( [X,Y | Rest], [Y,X | Rest] ) :-
    X > Y ,!.
bubble( [Z|Rest], [Z | Rest1]) :-
    bubble(Rest, Rest1), !.
```

The first clause swaps the first two elements when the first one is larger than the second one. When that is not the case, then in the second clause we keep the first element in its place and bubble the rest of the list.

Finally add to the run/0 predicate the clause to produce the bubble sort.

```
    sort::bubblesort(UnsortedList, BubSortedList),
    stdIO::write("The sorted list (bubblesort) is: ",BubSortedList),
    stdIO::nl,
```

Run and enjoy the results.

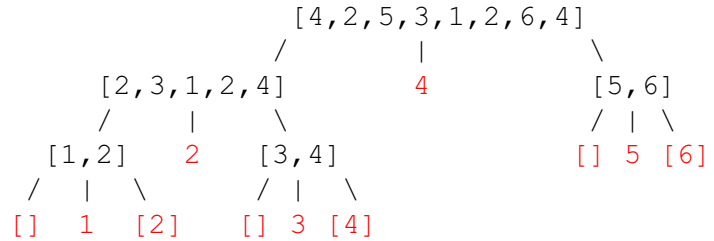
These three methods are not very efficient. During the sorting the lists must be searched more than once. A methods that does a better job is called “quicksort”. The method takes these steps:

1. take an element. Any element will do, so, as the list is unsorted, simply take the first one
2. put the remaining elements into one of two lists: one list for the smaller elements and one for the bigger ones.
3. then repeat the first two steps with each one of the two lists until finally you hold many lists with each one element in it.
4. as you did continuously divide the elements into smaller and bigger categories, these lists will be sorted.

You may find this hard to understand. Try to do it by hand with the list [4,2,5,3,1,2,6,4]

1 the first element is 4
2 splitting the list in two results in two lists: [2,3,1,2,4] and [5,6] with the element 4 between them. (I put the elements equal to four in the lower list)
3 take the lower list. The first element is 2
4 split the list in two. The result is: [1,2] and [3,4] with the element 2 between them.
5 take the lower list. The first element is 1.
6 Split the list in two, The result is [] and [2] with the element 1 between them. As the resulting lists are empty or contain only one element, you can stop this branch and continue with the next branch, that is the list [3,4].

Continuing like this you end up with a tree structure. The elements in red are at the end of a branch. They are the leaves and together they form the sorted list.



When you now walk from left to right along the leaves of the tree (the red numbers), neglecting the empty lists, you will see that the elements are sorted.

We will apply this principle in the predicate “quicksort”. As usual I shall presume that this predicate takes an unsorted list and produces a sorted list. So the declaration is familiar:

```
quickSort : (integer*, integer*) procedure (i,o).
```

From the description you can see that the predicate will be recursive. So the first clause is not too difficult, it is the usual sorting of an empty list:

```
quicksort( [], []) :- !.
```

Things get a bit more complex when the list is not empty. Then we have to take the first element and divide the rest of the list into two new lists, one with the smaller elements (and the equals) and one with the bigger elements. Here we go:

```
quicksort( [SplitElem | Tail], SortedList) :-
   split(SplitElem, Tail, LowerList, UpperList),
                /* first split the tail into two lists using SplitElem as a splitting element */
   quicksort(LowerList, SortedLowerList),
                /* then quicksort the lower list into a sorted list */
   quicksort(UpperList, SortedUpperList),
                /* then quicksort the upper list into a sorted list */
   append(SortedLowerList, [SplitElem | SortedUpperList], SortedList).
                /* then put the sorted lists and the splitelement together in one list */
```

As you can see it is the splitting that does the trick. We continue splitting lists until each list contains only one element or no element at all.

For splitting I use the predicate “split”. Here is the declaration:

```
split : (integer, integer*, integer*, integer*) procedure (i,i,o,o).
```

It takes an integer and a list for input and produces two lists that can be worked with in the next round. This predicate is also recursive. So we start with splitting an empty list. In that case it doesn’t mater what element we use as a splitting element. It gives the clause:

```
split( _, [], [], []):- !.
```

When the lists are not empty, we must compare the first element (the Head) in the list_to_be_splitted (Tail) with the splitting element. When Head is smaller or equal to splitting element, it goes into the lower list. If not, we use the second clause and put it into the upper list. In Prolog:

```
split(SplitElem, [Head | Tail], [Head | LowerList], UpperList) :-
   Head <= SplitElem, !,
   split(SplitElem, Tail, LowerList, UpperList).
split(SplitElem, [Head | Tail], LowerList, [Head | UpperList]) :-
   split(SplitElem, Tail, LowerList, UpperList).
```

In both cases we continue with splitting the Tail. Finally we need a predicate to put a list, an element and another list together into one list. I made the predicate “append” that takes two lists and puts the elements together into one list, thereby retaining the order of the elements. Because I call the predicate with

```
   append(SortedLowerList, [SplitElem | SortedUpperList], SortedList).
```

You should realize that the splitting element is already in place. So now I only have to take care of the elements of the lower list. Here is the declaration:

```
append : (integer*, integer*, integer*) procedure (i,i,o).
```

These are the clauses

```
append([], List, List) :- !.
append([Head | Tail], List2, [Head | List3]) :-
   append(Tail, List2, List3).
```

To run the program I adjusted the run/0 predicate

```
run():-
    console::init(),
    UnsortedList = [4,2,5,3,1,2,6,4],
    stdIO::write("The unsorted list is: ", UnsortedList),
    stdIO::nl,
    sort::insertsort(UnsortedList, InSortedList),
    stdIO::write("The sorted list (insert sort) is: ",InsortedList),
    stdIO::nl,
    sort::selectsort(UnsortedList, SelSortedList),
    stdIO::write("The sorted list (select sort) is: ",SelsortedList),
    stdIO::nl,
    sort::bubblesort(UnsortedList, BubSortedList),
    stdIO::write("The sorted list (bubblesort) is: ",BubSortedList),
    stdIO::nl,
    sort::quicksort(UnsortedList, QuickSortedList),
    stdIO::write("The sorted list (quicksort) is: ",QuickSortedList),
    _X = stdIO::readchar(),
    succeed().
```

This should do the job. If you think it is a pity that you don't see how the sorting is going on, feel free to insert write-statements at the appropriate places. As you can see in the code, the predicate stdIO::write() takes care of lists as well as other types of variables. That makes life easy.

10.6 Summary

These are the important points covered in this chapter

1 Lists can contain an arbitrary number of elements; you declare them by adding an asterisk at the end of a previously defined domain.

2 A list is a recursive compound object that consists of a head and a tail. The head is the first element and the tail is the rest of the list (without the first element). The tail of a list is always a list; the head of a list is an element. A list can contain zero or more elements; the empty list is written [].

3 The elements in a list can be anything, including other lists; all elements in a list must belong to the same domain. The domain declaration for the elements must be of this form:

```
domains
    element_list = elements*.
    elements = ....
```

where elements = ... one of the standard domains (integer, real, etc.) or a set of alternatives marked with different functors (int(integer); rl(real); smb(string); etc.), separated by semicolons. You can only mix types in a list in Visual Prolog by enclosing them in compound objects/functors.

4 You can use separators (commas, [, and |) to make the head and tail of a list explicit; for example, the list

```
[a, b, c, d]
```

can be written as:
[a|[b, c, d]] or[a, b|[c, d]] or[a, b, c|[d]] or[a|[b|[c, d]]] or[a|[b|[c|[d]]]] or even [a|[b|[c|[d|[]]]]].
Note the empty list in the last example.

5 List processing consists of recursively removing the head of the list (and usually doing something with it) until the list is an empty list.

6 The standard list predicates can be found in the **list** class.

7 Visual Prolog provides a built-in construction, list comprehension, which takes a goal as one of its arguments and collects all of the solutions to that goal into a single list. It has the syntax
Result = [Argument || myPredicate(Argument)]

8 Because Visual Prolog requires that all elements in a list belong to the same domain, you use functors to create a list that stores different types of elements.

9 Sorting a list can be done in various ways. Not every way is an efficient way.

Chapter 11 Reading, writing, streams and files[15].

To be useful, a computer program has to communicate with its user. So a computer needs to know where it can read input from the user and where it can send output to the user. During the development of computers and computing the communication with the user has developed as well. In this chapter we take a look at the way your program communicates with the user. To understand the way input and output is handled in VIP, you need to know a bit more about the background and history of computer input and output. So I decided to make this chapter more than only a summing up of the predicates that are there for input and output. But take care. As I am a newbee in object oriented programming myself, I had to gather everything from several sources and then had to give it the right interpretation. So maybe I am telling you things that are not true or only half true.

11.1 The console

The console is the entry and display device for system administration messages, for messages during booting (starting) the computer and from the system logger that lets you logon. It is a physical device (you can touch it :-)) for simple text entry and text display consisting of a keyboard and a screen.

On traditional computers, the console was a dumb terminal physically consisting indeed of a keyboard and a non-graphics screen. By way of a serial connection it was connected to the computer. This terminal was usually kept in a secured room since it could be used for certain privileged functions such as halting the system or selecting which media to boot from. Large midrange systems, e.g. those from Sun Microsystems, Hewlett-Packard and IBM, still use serial consoles.

On PCs the keyboard, the mouse and the screen are connected to the computer (or maybe you like to say that they are an integrated part of it) and they act as the console. But this “console” is more complicated. Not only a mouse is added, but the screen is capable of showing graphics (as you will notice when you logon to Windows). It is more than a simple place to show characters. When we talk of a console, you should keep in mind that we think of it as a simple device that receives input from the keyboard and shows output as a series of characters on a simple non-graphic screen. Think of it as the MsDos Command Prompt Window under Windows. This Windows console is a plain text window for console applications within the system of Windows Application Programming Interface (API). It is called the Win32 console. A Win32 console has a screen buffer and an input buffer. These buffers are used as an intermediary between the keyboard and the computer and between the computer and the screen. Think of it in this way. When you enter characters on the keyboard, they are put into the input buffer and when you hit <Enter> the computer reads the input buffer. When there is some output, the computer writes it to the buffer and when an End-Of-Line character is written to the buffer, then the contents of the buffer are written to the screen as a single line. Win32 consoles are typically used for applications that do not need to display images. In VIP you create console computer programs when you set “console” as a target for your VIP project.

---

[15] The general stuff in this chapter is based on Wikipedia and other theoretical sources, much of the specific stuff is from the VIP help files and there are a few things from private correspondence with Thomas Linder Puls. Thank you, Thomas!

In VIP there is a special class of procedures to read from and write to the console. Think of a VIP program as a process. Then you can imagine that when you work with a few programs at the same time, there are many processes going on at the same time. Each of these processes has a connection to a console and at the same time each process has no more then one console. For input and output the console supports three streams: an input stream where it reads characters, an output stream where it writes characters and an error stream where it writes special messages in case of errors. More on the concept of streams in a later section, for now think of it as a channel through which the computer gets or writes (a stream of) characters.

The input stream is by default connected with the keyboard, the output and error streams are by default connected to the screen. But they can be redirected.

When you want to write directly to the console, use the predicates in the class "console". But I must strongly advise you not to do so, unless you have very good reasons and know what you are doing. Writing to the console means among others that you must be sure that there is a console. In VIP and the IDE this is clear from setting the target to "console"[16]. But what happens when you export your program to another computer? In that case the program may not work as intended.

Instead of writing to the console, you've better use the class for Standard Input and Output called "stdIO" that we used before. The procedures in this class will always write to the right output device. In a console environment it uses the Win32 console, in a GUI environment it uses the Message Window in the Task Window.

You will find more details of the class "console" and its predicates in the help file.

11.2 The Message Window and the Error Window in VIP

In VIP you have encountered the Messages Window. It plays the role of a console in this way that it displays all kinds of messages from the system, especially various progress messages describing important IDE events like: a file or the project is saved, a new project is opened, a module is compiled, a project is built, etc. In the Messages window you can see a list of all important IDE events, it contains a history of the IDE execution. You see the Messages Window in figure 11.1

But as the Messages Window is a Windows window, you can do much more with it. To name a few:

- The Messages window can be scrolled or re-sized at any time to view more events than are visible in the window.
- It is possible to copy any selected part of text from the Messages window to the clipboard.
- Using the right mouse button anywhere in the Messages window will bring up a local pop-up menu.
- The Font for the Messages window can be changed with the *Tools/Options* command in the Fonts tab or with the Font item in pop-up menu.
- In the Messages Window tab of the Options dialog (you can activate it with the *Tools/Options* command) it is possible to set:
    - the number of lines to remember in the Messages window,

[16] A console target program starts with the clause "console::init()".

- where the Messages window should be positioned,
- whether messages in the Messages window should be wrapped.

- If the Messages window is closed, it can be opened again by clicking the option *View/Messages Window* in the menu.

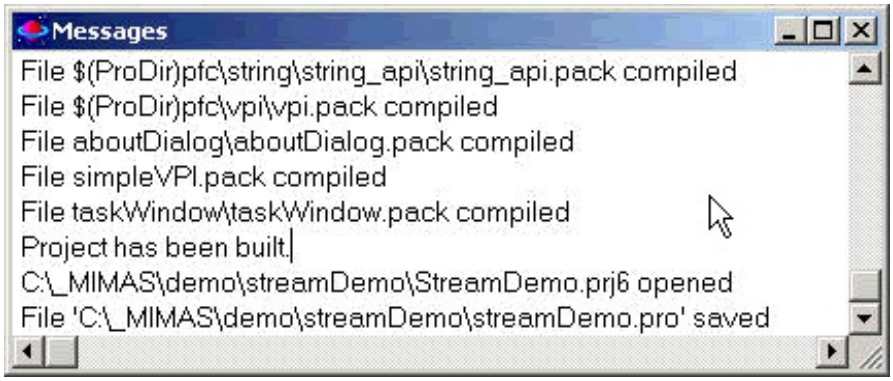


Figure 11.1 The Messages Window in the IDE

In the IDE there is a separate window for displaying errors, for displaying the error stream. Most of the time it is hidden and of course you (as a user) hope it will never show :-). The IDE displays this window when error or warning messages are generated by the compiler while compiling Prolog files or by the linker while combining object files, libraries, and resource files into the single target file. If the Errors Window appears, it looks like the one see in figure 11.2.

:

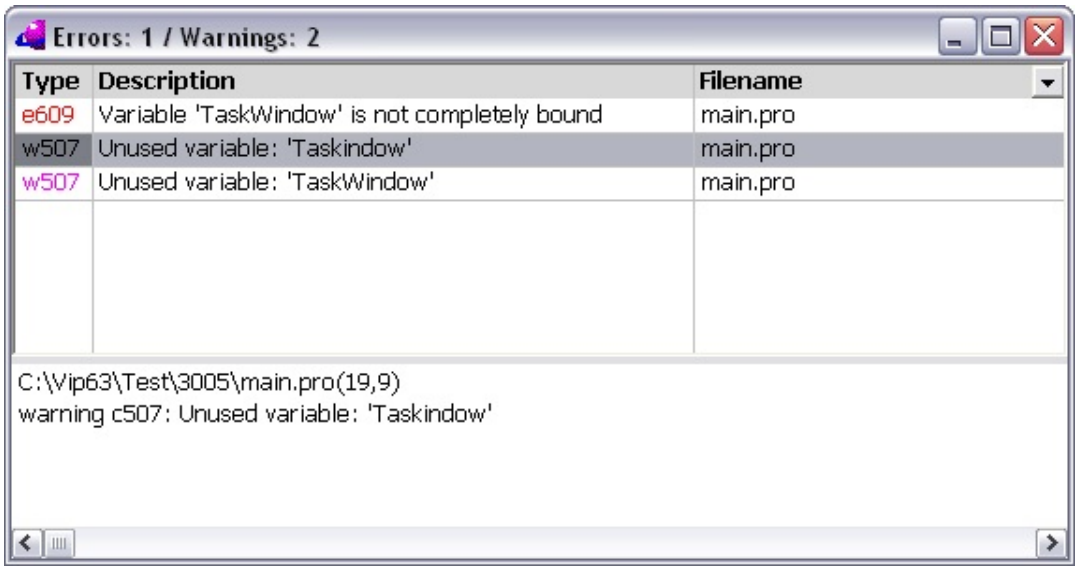


Figure 11.2 The Errors and Warnings Window

Each line in the window contains three parts: Type, Description and Filename. The Type column displays type of the error and a certain number of the message. The type can be Error (“e”) or Warning (“w”). When it is an Error, the compilation continues as far as possible, but no linking or execution will occur. When it is a Warning, compilation and linking continue and, if wanted, execution takes place. The color of Errors is red and the color of Warnings is magenta. For example

"e609" means an Error with number 609. The message "w507" means a warning with number 507. You can find detailed descriptions of Errors and Warnings in the Visual Prolog Help File. When the cursor is positioned on some line inside the Errors (Warnings) Window and you press the F1 key, then the Visual Prolog Help is activated and opens the topic containing the detailed description of the error reported in the selected line.

The Description column contains description of the message. The Filename column contains the name of the file with relative path where an error or warning has occurred.

When you double-click the line in the Errors window, which contains a compiler error or a warning message, then the IDE opens the referenced source file and puts the cursor on the reported error/warning position.

The lower edge of the Error Window contains a full description of the error: full path of the file with position in text, full name of message including number and description. If for some reason F1 cannot be used, then you can find the topic with detailed error description if you open the Index tab in the Visual Prolog Help and specify the reported error number, like this "e507".

There is a class with predicates to write directly to the Messages window. The classs is called “vpiMessages” and you will find details in the VIP help file. It is tempting to write to the Messages Window (and maybe the Error Window) directly. But again, I must strongly advice you not to do so unless you have very good reasons. It is better to use the Standard Input Output class “stdIO”. Just like it was the case with the console, you are not sure if the Messages Window is there to accept your writings. The user may have closed the messages window. In that case your writings are lost. When you use “stdIO” in stead, these predicates will always write to the “current” output channel.

11.3 Streams

In the old times there were not many devices to read from or to write to. That has changed considerably. You read from and write to USB sticks, hard disks, CD’s, DVD’s, ftp-sockets, graphical interfaces, your computer receives radio, TV, telephone, FAX and video signals. It’s a mess. Imagine that you as a programmer had to take care of all these different signals in different buffers and that you had to code and decode every signal in its own way and maybe in its own language. A hopeless job.

That’s why nowadays we think of input and output as coming and going through channels that are called streams. Every input and output channel is a different stream. Standard there are three streams: the input stream, the output stream and the error stream. Streams have two advantages above the older IO-systems where the programmer connects his program directly with input and output devices. Streams are automatically connected, you don’t have to do anything. And secondly streams are covered from your program. You simply use something like “stdIO::write(...)” and the system takes care of writing your output to the device. That means that you can concentrate on the problem that you want to solve with your computer program en can forget about technical details of input or output. Besides in this way it is easier to transport programs to other computer with probably other implementations.

A stream is thought of as a sequence of bytes. A byte is a sequence of eight binary digits, so a sequence of bytes looks like:

11001010 01010101 01010100 11010101 01010010 10110101 11010101 10101010 11111100 ...

In a text you can think of a byte as a code that stands for one character, but that is not always true. The spaces that I typed are only for human eyes. In a real stream they are not there. A stream like this one looks impressive, but to read it you need some interpretation. What does 11001010 mean in plain English? To solve questions like these and to create a universal standard for interpretation, several solutions have been proposed. The first one (as far as I know) was the ASCII code table. In the ASCII code table there were 256 codes that represented capitals, lower case letters, control characters (newline, backspace) and punctuation characters like colon, semicolon and comma. The capital letter “A” for instance has ASCII code 65, or in bytes: 0100 0001.

Nowadays other codes are used. In VIP two codes are supported, ANSI code and Unicode. ANSI code is an extension of the older ASCII code. It gives the possibility to add control characters that control a.o. the color of the screen. These controls are called ANSI escape sequence. An ANSI escape sequence is a sequence of ASCII characters, the first two of which are the ASCII "Escape" character 27 (binary: 0001 1011 and hexadecimal: 1B) and the left-bracket character " [" (binary: 0101 1011 and hexadecimal: 5B). The character or characters following the escape sequence characters specify an alphanumeric code that controls a keyboard or display function. You already met this principle when we wrote “\n” in a write statement. The backslash works in VIP as an escape sequence, the “n” denotes a new line.

Unicode is a code that was created in an attempt to deliver a code that can be used for every language in the world. As this is rather ambitious, the founders of unicode started to base the code on 16 bits in stead of 8 bits. You don’t have to bother too much about all this, as long as you know what code is used for the files that you have. Even a simple editor program like Notepad nowadays can read and write both codes.

Now, what has all this to do with streams in VIP? Well, a stream is not just a sequence of binary data, it has also two attributes. One attribute is the mode, the other is the position. In VIP there are two or three modes, depending on how you want to think about it. These modes indicate how to interpret the bytes in the stream. Here they are:

- Binary mode
  In this mode your program reads the bytes from the input stream just as they are: a sequence of bits. What you do with it, is purely your business. For output goes the same, what you write is put in the output stream as binary digits. There is no interpretation.
- Text mode
  In text mode there is an interpretation involved. The bytes in the stream are read as letters, numbers et cetera. This can lead to a conversion of bytes into characters. When you read a text in text mode, there will be all kinds of special characters, like “newline”. When you write to a printer, this “newline”-character is converted into two control characters for the printer: a “linefeed” and a “carriage return”. There are other transformations. E.g. in Unicode the number twelve is one code: 0x0C. When this code is sent to the screen there is an automatic conversion so on your screen you see “12", that is you see the characters “1" and “2". There are two text

modes, ANSI mode and Unicode mode. The difference is that ANSI mode uses ANSI coding and Unicode mode uses Unicode coding.

The second attribute, the position, takes care of where you are in an input or output stream. It offers e.g. the possibility to go back in an input stream to a previous read character. More on this in the next sections.

11.4 Standard Input and Output: the class StdIO

In VIP there is the class Standard Input and Output. It is called class “stdIO” (without the quotes) and we have used it already. When a program starts, the stdIO consists of an input stream, an output stream and an error stream. The input stream is standard connected to the keyboard. When the strategy of the project is “console”, then the output stream and the error stream are connected to the console window (the Win32 console). When the project strategy is “GUI” then the output and error stream are connected to the Message Window.

To begin with, let us take a look at an example program. We create (strategy: “console”) a project with one class called “database”. The program will read some data from the keyboard, assert them to a data base and when we end the program, it will write the contents of the database to a file. I put the predicates in a module “database”. Here is the code, including some clarifying comments.

```
class database
   open core

predicates
   classInfo : core::classInfo.
   storeNames : (unsigned Counter) -> unsigned Number procedure (i).
   fill : () procedure ().

end class database
```

And in the implementation “database.pro” I add this code

```
class facts - persons                  % the facts are stored in a database with the name “persons”
   person : (string Name).

clauses
   storeNames(N) = N :-                % this predicates reads a line and adds a fact to the local %
                                       database
      stdIO::write("\nInput Name (<Enter> '0' to stop): "),
      '0' = stdIO::readchar(),         % the program looks if the first character is ‘0'
      !.                               % if it is, the program stops
   storeNames(I) = Number :-
      stdIO::ungetchar(),              % if not, the program puts back the first character
      Term = stdIO::readline(),        % and reads the complete inputline
      assert(person(Term)),            % and adds the new person to the database
      N=I+1,                           % raises the counter by one
```

```
    Number = storeNames(N).          % Number counts the number of records

  fill() :-                          % starts filling the database by calling “storeNames/1”
% Fill local database
    N = storeNames(0),
    stdIO::writef("\n %d terms were stored \n Press any key\n", N),
    stdIO::save(persons),            % saves the db to the current output stream, the screen
  % Wait
    _Char = stdIO::readchar(),
    DBFile = outputStream_file::create8("persons.txt"),
                        /* this creates an outputfile in ANSI text mode. We refer to this file with the
                        variable “DBFile”. This variable is now of type Stream */
    stdIO::setoutputstream(DBFile),      % DBFile becomes the current output stream
    stdIO::save(persons),                % write the database to the outputbuffer
    stdIO::flush(),                      % empty the buffer to the file
    _Char2 = stdIO::readchar().

end implement database
```

Some comments here. I give the data base a name, so I can refer to it in the “save”-command. To be able to write to a file instead of the screen, I first create this file. When the file is created, an existing file with the same name is overwritten. Then I redirect the “standard” output stream to that file and with the save-command the database contents are written to the output file. In writing, the program writes to a buffer. We do not write directly to the hard disk, because then the program will be too slow. In stead we use a buffer. A buffer is a piece of memory. It has fast access and when the buffer if full, then a separate program writes the buffer to the file on the disk while the processor can do another (fast) part of the job. So before we close the output stream (this is done automatically when the program ends), we first must “flush()” the buffer. This sends what is left in the buffer to the file. If you don’t do this, the contents of the buffer are lost.

With the command

```
stdIO::setoutputstream(DBFile)
```

we change the current output stream. That means that the write and save commands write to that file in stead of the standard output stream. (You may still see some <returns> on the win32 console. That is because the things that you type are echoed to that screen.) If you want to write to the screen again, you have to redirect the output stream with another call to setOutputStream(). It could be done with:

```
    NewOutput = console::getconsoleoutputstream(),
    stdIO::setoutputstream(NewOutput),
    stdIO::write("Thats all"),
```

When you put these lines at the end of the clauses for fill/0, then the last write statement will appear on the screen again.

What’s left to do is the code for “main.pro”. Here it is.

```
clauses
  run():-
    console::init(),
```

```
    database::fill(),
    _=stdIO::readchar(),
    succeed(). % place your own code here
```

In the remainder of this section I give an overview of the most important predicates from stdIO. Most important means that these are for a beginner the most useful ones. At least I think they are. I group them in four categories: predicates for input, for output, for error and for databases. For every predicate I give the name, the declaration and a short description. What follows is more or less an edited copy of parts of the helpfile.

First let's take a look at the predicates to read from the **input stream**.

---

stdio::getInputStream/0->
getInputStream : () -> inputStream Stream procedure ().

Description
The predicate returns current input stream. That means that it binds a variable to the stream and from this moment on you can refer to the stream by way of the variable. The variable is of type Stream, you cannot print it. The help file says that it should be set previously by the predicate setInputStream/1, but when I tried, it worked also when I didn't set it myself. You may use this predicate to temporary change the input stream and then return to the previous one.

---

stdio::read/0->
read : () -> _ Term procedure ().

Description
The predicates reads a specified term from the input stream. The type of Term can be any legal Prolog term. That means that with this predicate you can read integers, reals, chars, strings, facts, whatever. The program decides what is to be expected and reads the appropriate term type. E.g. when you have declared a variable to be of type integer, and you read it somewhere in your program, then the program knows what it must expect. When you are not sure or when a variable has not been declared, then you can use the predicate hasdomain/2 to exactly define the term's domain. Because this may sound tricky, I devote the next section to this subject.

---

stdio::readBytes/1->
readBytes : (byteCount Size) -> binary Binary procedure (i).

Description
When the input stream has mode "binary", then your program should read bytes. This predicate reads a number of bytes from the input stream and stores them into the variable Binary. The number of bytes to read is given in the variable Size when you call the predicate. If this predicate meets the end of the input stream before the number Size of bytes is read, then it returns in Binary the number of bytes actually read.

---
stdio::readChar/0->
readChar : () -> char Char procedure ().

Description
The predicate reads and returns a single character from the current input stream. To my surprise, the predicate expects also an end-of-line, but maybe I am mistaken.

---
stdio::readLine/0->
readLine : () -> string String procedure ().

Description
The predicate reads a line from the input stream. Indicator of the end of the line is a symbol '\n' (new line) or zero-char (string termination symbol) or end of stream.

---
stdio::readString/1->
readString : (charCount NumberOfChars) -> string String procedure (i).

Description
The predicate tries to read NumberOfChars characters from the input stream and stores them into the String. If the predicate meets end of stream when reading it returns the string with the number of characters actually read.

---
stdio::setInputStream/1
setInputStream : (inputStream Stream) procedure (i).

Description
This predicate sets Stream as current input stream. As an input you need a variable of type Stream. You can get such a variable e.g. when you create a new file

---
stdio::ungetChar/0
ungetChar : () procedure ().

Description
The predicates returns last read character back to the current input stream.

Now it is time to look at the predicates for writing to the **output stream**. Here they are.

---
stdio::flush/0
flush : () procedure ().

Description
This predicate forces the contents of the internal buffer of the current output stream to be written to the resource (e.g. the file) that it is connected to. The flush is useful when output is directed to a serial port and it becomes necessary to send the data to the port before the buffer is full. It is also used to ensure that another process will get latest data from the stream.

---

stdio::getOutputStream/0->
getOutputStream : () -> outputStream Stream procedure ().

Description
This predicate returns the name of the current output stream. It should be set previously by setOutputStream/1 predicate.

---

stdio::nl/0
nl : () procedure ().

Description
The predicate writes a “new line symbol” to the current output stream.

---

stdio::setOutputStream/1
setOutputStream : (outputStream Stream) procedure (i).

Description
This predicate sets Stream as current output stream. All output will be directed to this new current output stream. The variable Stream is of type “stream”. That type is invoked when you open new file for output. That is done with (e.g.) A clause like:

```
DBFile = outputStream_file::create8("persons.txt"),
```

as we did in the last program. By definition the variable DBFile gets type “stream”and can be used everywhere where you use a variable of this type.

---

stdio::write/...
write : (...) procedure (...).

Description
The predicate writes an arbitrary number of variables to the current output stream. The variables are separated by comma’s and you can include strings to clarify the output. You can use any type of variable.

---

stdio::writeQuoted/...
writeQuoted : (...) procedure (...).

Description

The predicates writes specified variables to the current output stream. It can be called with an arbitrary nonzero number of arguments. For all string (char) variables the prefixed and post fixed double (single) quotes will be written into the stream too.

---

stdio::writef/1...
writef : (string Format <formatString>, ...) procedure (i,...).

Description
The predicate write() produces output that is formatted “automatically”. With this predicate you produce output that is strictly formatted. E.g. when you want to produce neatly formatted columns. The trick is that you precede the list of variables to be output by a string of format specifications. As this is an overview of predicates, I postpone the details of the formatting till section 11.6

And here are the predicates to write to the **Error Stream**. They look a lot like the predicates for the Output Stream, So I give them without much further comment.

---

stdio::flush_error/0
flush_error : () procedure ().

Description
This predicate forces the contents of the internal buffer of the current error stream to be written to resource. The flush is useful when output is directed to a serial port, and it becomes necessary to send the data to the port before the buffer is full. It is also used to ensure that other process will get the latest data from the stream.

---

stdio::getErrorStream/0->
getErrorStream : () -> outputStream Stream procedure ().

Description
This predicate returns the current error stream. That means that it binds a variable to the stream and from this moment on you can refer to the stream by way of the variable. The Error stream should be set previously by the predicate setErrorStream/1.

---

stdio::nl_error/0
nl_error : () procedure ().

Description
The predicate writes new line symbols to the current error stream.

---

stdio::setErrorStream/1
setErrorStream : (outputStream Stream) procedure (i).

Description
This predicate sets Stream as current error stream.

---
stdio::writeQuoted_error/...
writeQuoted_error : (...) procedure (...).

Description
The predicates writes specified variables to the current output stream. It can be called with an arbitrary nonzero number of input arguments. For all string(char) variables the prefixed and post fixed double(single) quotes will be written into the stream too.

---
stdio::write_error/...
write_error : (...) procedure (...).

Description
The predicate writes specified variables to the current error stream.

---
stdio::writef_error/...
writef_error : (string Format [formatString], ...) procedure (i,...).

Description
The predicate produces formatted output to the current error stream. See description of format string in outputStream interface

Finally there are some predicates for **special purposes** that you should know. They are:

---
stdio::consult/1
consult : (factDB FactsSectionName) procedure (i).

Description
The predicate consult/1 reads facts from the current input stream and asserts them in the internal facts section with the name FactsSectionName. An input file must first be created with the save/1 predicate or created by hand in a text editor (or another program). The file should have each fact on a separate line, it must not contain extra spaces and the functors must be written in lower case letters. When you use a file that was previously saved with save/1, there should be no problem. Comments are allowed, both single and multi- line comments.

---
stdio::save/1
save : (factDB FactsSectionName) procedure (i).

Description

The save predicate saves the contents of the facts section with the name specified by FactsSectionName into the current output stream.

---

stdio::save_error/1
save_error : (factDB FactsSectionName) procedure (i).

Description
The save predicate saves the contents of the facts section with the name specified by FactsSectionName into the current error stream.

---

stdio::endOfStream/0
endOfStream : () determ ().

Description
Checks if current input stream position is the end of the stream. Predicate fails, when stream pointer does not point to the stream's end. This predicate always fails for undetermined streams.

11.5 The predicate stdIO::read

The predicate read/0 is a general predicate. It has as a characteristic that the predicate determines itself what type of input it should read. I think it is best to show you a little program. I made a new project (target console) and added these clauses to main.pro

```
clauses
   run():-
      console::init(),
      readExamples::writeSomeTerms(),
      Intfile = inputstream_file::openfile("intSource.txt"),
      Stringfile = inputstream_file::openfile("stringSource.txt"),
      FunctorFile = inputstream_file::openfile("functorSource.txt"),
      readExamples::readSomeTerms(Intfile, Stringfile,FunctorFile),
      readExamples::writeSomeTerms(),
      readExamples::readSomeTerms(Intfile, Stringfile,FunctorFile),
      readExamples::writeSomeTerms(),
      _ = stdIO::readchar().
end implement main
```

I made three files (in Unicode). File “intsource.txt” contained two integers
123 456
File “stringsource.txt” contained three lines with a string:
"here is another string"
"And again another one"
File “functorsource.txt” contained two values for the functorfact:
person("AnotherThomas")

person("Anna")
Take care to put each person on a new line and NOT to use a period at the end of the line!

In main three objects are created that are connected to these three files. Then we call the class "readexamples" and make it write the values of three fact variables. The class readexamples looks like this.

```
class readexamples
   open core

predicates
   readSomeTerms : (inputStream Ints,  inputStream Strings, inputstream Functors) procedure (i,i,i).
   writeSomeTerms : () procedure ().

end class readexamples
```

And the implementation is:

```
domains
   person = person(string Name).

class facts
   intfact : integer := 1.
   stringFact : string :="This is a string".
   functorFact : person := person("Thomas").

clauses
   classInfo(className, classVersion).

clauses
   writeSomeTerms() :-
      stdIO::write("intfact = ",intfact, "\nstringfact = ", stringfact, "\nfunctorFact = ", functorfact),
      stdIO::nl.

   readSomeTerms(Intfile, StringFile,FunctorFile) :-
      intfact := Intfile:read(),
      stringfact := StringFile:read(),
      functorFact := FunctorFile:read().

end implement readexamples
```

I initialize the facts with a value. When "readsometerms()" is called, new values are read from the three files. Although the three facts are of different type, the program determines what to read in each case from the type declarations.

Sometimes it is not possible to give such neat type declarations like the ones in this example. In that case you can force the typing of a variable with the help of a special predicate "hasDomain/2". It takes two arguments, a type argument and a variable name. The use is like this:

```
predicates
        readint : (inputStream S) -> integer Value.
clauses
        readint(S) - Value :-
                ...,
                hasDomain(integer, Var),
                Value = S:read().
```

By the use of hasDomain/2 you explicitly state that the variable Value has type integer, so the predicate read/0 will try to read a integer from inputstream S. In stead of “integer” you can of course use other types, predefined or declared.

11.6 The predicate writef() en the format string

The predicate writef() produces formatted output to the output stream. To format the output you provide a so-called format string. The general syntax of calling the predicate is

stdIO::writef(<format string>, <list_of_variables>).

The <format string> is a string that contains one or more <format field>s. For every variable in the <list_of_variables> you must give a format field in the string. The format string can become complicated because you can also add text that will be interwoven with the variables. The number of actual variables in <list_of_variables> must match the format fields specified in <format string>, otherwise a run-time error "Wrong number of arguments in the format string" is generated.

A <format field> begins with the percent '%' sign and contains up to five <format specifier>s. A <format specifier> exactly indicates how the matching variable should be written in the output stream. If the percent sign is followed by some unknown character (not a <format specifier>) - then this character will be printed without modification and thus will appear in the output. The <format field> syntax is:

%<-><0><width><.precision><type>

That means that it starts with a %-sign that is followed by five possible <format specifier>. All specifiers are optional and written without < and >. Here is a description

<-> Hyphen indicates that the field is to be left justified; right justified is the default. The hyphen has no effect as the <width> value is not set because then the width will automatically be set to the number of places needed. Nor has a hyphen effect when the number of characters in the actual value is greater than width value.

<0> The zero before width means for values that zeros will be added until the minimum width is reached. If <0> (zero) and <-> -(hyphen) appear, the 0 is ignored.

<width> The <width> is a natural number, that is it is a positive decimal integral number specifying a minimum field width. If the number of characters in the actual value is less than width value - then the required number of space ' ' characters will be added before the value (or after it, if the <-> field was set). No changes occurs if number of characters in the actual value is greater than the width value.

<.precision> The point '.' with the following unsigned decimal integral number can specify either the precision of a floating-point value or the maximum number of characters to be printed from a string. It depends on the matching variable.

<type> Specifies other formats then the default for the given variable. For example, in the type field, you can give a specifier that says an integer must be formatted as an unsigned. The possible values are:

f	Format reals in fixed-decimal notation (such as 123.4 or 0.004321). This is the default for reals
e	Format reals in exponential notation (such as 1.234e+002 or 4.321e-003).
g	Format reals in the shortest of f and e format, but always in e format if exponent of the value is less than -4 or greater than or equal to the precision. Trailing zeros are truncated.
d or D	Format as a signed decimal number.
u or U	Format as an unsigned integer.
x or X	Format as a hexadecimal number.
o or O	Format as an octal number.
c	Format as a char.
B	Format as the Visual Prolog binary type.
R	Format as a database reference number.
P	Format as a procedure parameter.
s	Format as a string.

Note: when converting real values into a text representation they are truncated and rounded to 17 digits, unless another quantity is specified.

Here is an example:

```
clauses
   run():-
      console::init(),
      stdIO::writef("a <formatstring> without variables\n"),
      IntegerVar = 12345,
      stdIO::writef("Numbers %5\n wider %10 \n left justified %-10 <-end of field is here",
IntegerVar, IntegerVar, IntegerVar),
      RealVar = 12345.6789,
      stdIO::writef("\nRealNumbers %5.4\n wider %10.10\n left justified %-10.4 <-end of field is
here\n rounded %5.2", RealVar, RealVar, RealVar, RealVar),
      String = "This is a string",
      stdIO::writef("\nString \n %20|end is here\n %-20|end is here\n %5|end is here\n %20.7|end is
here \n", String, String, String, String),
      stdIO::write("\n hit <Enter>"),
      _ = stdIO::readchar(),
      succeed(). % place your own code here
end implement main
```

Take care that when you add text in the format string next to a format field, then the first letter is interpreted as part of the format field. E.g.

```
      RealVar = 12345.6789,
```

```
    stdIO::writef("%10.4 end of field is here", RealVar)
```
produces
```
     12345.6789 end of field is here
```
But
```
    RealVar = 12345.6789,
    stdIO::writef("%10.4end of field is here", RealVar)
```
produces
```
     12346.e+004nd of field is here
```
because the 'e' of "end" is interpreted to modify the format to exponential notation.

Note that you can also add escape sequences in the format string like "\n" to go to a new line. The escape sequences that you can add here are:

\\	representing \ .
\t	representing Tab character.
\n	representing New Line character.
\r	representing carriage return.
\"	representing double quote
\uxxxx,	here xxxx should be exactly four hexadecimalDigit's representing the Unicode character corresponding to the digits.

So if you want to insert a backslash in the output, put "\\" in the <format string>. When you put "\t" in the <format string>, the printer will jump to the next tab-position. Et cetera.

I suggest that you fool around some time with different formats to get a good feeling for them.

11.7 General input and output: the class Stream[17]

The standard IO class allows you to read from and write to a stream, but only one at the time: the current input or output stream. The class Stream is a more general class. I'm not sure if what I am going to say now is correct. But you must think of the class Stream a a class that creates an object for every stream that you open for reading or writing. In standard I/O a stream is for reading OR for writing. In the stream class a stream can be for reading AND writing.

The class Stream is the root of a hierarchy of classes. On top is the class Stream. It contains some predicates that are most general, that means they can be used with any stream. Think of predicates to close the stream, to set or get its mode or to set or get the position in the stream. The next layer consists of the two classes Inputstream and Outputstream. These classes contain predicates to work in general with an inputstream or an outputstream respectively. In the class "Inputstream" there are predicates like read/0, readchar/0, readstring/0 and consult/1. In the class "Outputstream" there are predicates like flush/0, nl/0, save/1, write(), writeQuoted() and writef(). These predicates should be familiar by now and you probably are not surprised that they are there. But that isn't all. There is a third layer of classes below "Inputstream" and "Outputstream". They contain predicates for special

[17]Here I start the name of a class with a capital. This to avoid the extensive use of quote. In your code you need to use lower case letters.

purpose input or output streams. Each class is concerned with a special kind of input or output medium. Examples are

- class Outputstream_console for writing to the console.
- class Outputstream_file for writing to files.
- class Inputstream_file for reading from files.
- class Inputstream_console for reading from the console.

As this is an introductory text, I shall restrict myself to the classes “Inputstream_file” and “Outputstream_file”. The class Inputstream_file contains predicates to open a file in one of the three modes (binary, ANSI, Unicode), the class Outputstream_file contains predicates to create a new file in a certain mode, to open an existing file in a certain mode and to append to a file.

So generally speaking the predicates to handle input and output are in a hierarchy that is depicted in figure 11.3. When you are looking for a predicate, look in the class or its interface that is as high as possible in the hierarchy given the type of predicate you are looking for.

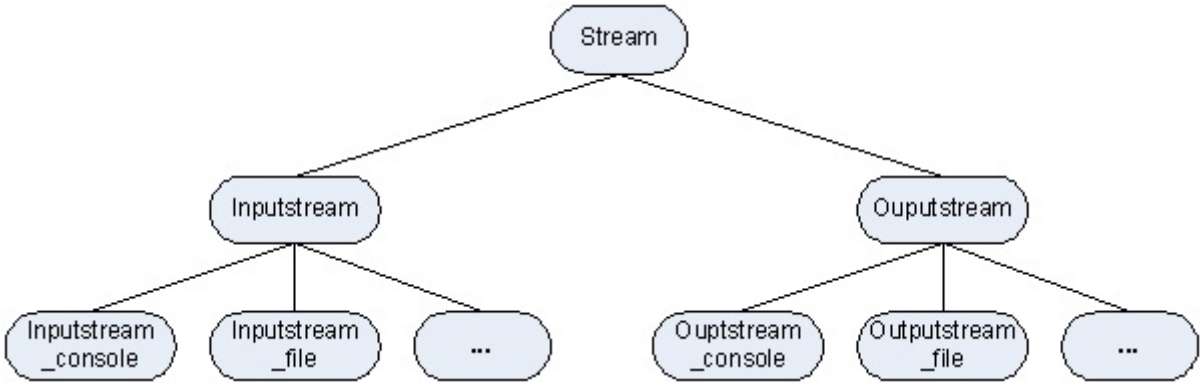


Figure 11.3 The class hierarchy of Stream

Before I go into the details of the predicates, here’s an example program. It will give me the opportunity to indicate some characteristics. The program is from the VIP helpfile but is slightly adjusted.

The program will copy the contents of a file to another file. The project has target “console” and consists of a class “copystream” that contains predicates to copy the contents of one file to another. In the file “main.pro” we call the predicates from “copystream” as you can see here.

```
clauses
   run():-
      console::init(),
      copystream::copy("source.txt","destination.txt"),
      _ = stdIO::readchar(),
      succeed(). % place your own code here
end implement main
```

The only thing we do here is to call the predicate “copy(“source.txt”, “destination.txt)” in the class “copystream”. We want to copy the contents of a file with the name “source.txt” to “destination.txt”. Be sure that the file “source.txt” is present in the .exe-folder of your project.

The code in class “copystream” looks as follows. I comment on the code below it. The declaration is:

```
class copystream
   open core

predicates
   copy : (string InputFileName, string OutputFileName) procedure (i,i).

end class copystream.
```

And here is the implementation

```
implement copystream
   open core

constants
   className = "copystream".
   classVersion = "".

clauses
   classInfo(className, classVersion).

  clauses
      copy(InFile, OutFile):-
         InputFile = inputStream_file::openFile8(InFile),
         OutputFile = outputStream_file::create(OutFile),
         copyStream(InputFile,OutputFile),
         InputFile:close(),
         OutputFile:close().

   class predicates
      copyStream : (inputStream InputFile, outputStream OutputFile) procedure (i,i).

   clauses
      copyStream(InputFile, OutputFile):-
         repeat(),
           String = InputFile:readLine(),
           stdIO::write(String),
           OutputFile:write(String, "\n"),
         InputFile:endOfStream(),
         !.
      copyStream(_InputFile, _OutputFile).

   class predicates
      repeat: () multi ().
   clauses
      repeat().
```

```
    repeat() :-
      repeat().

end implement copystream
```

I shall comment on most of the clauses, the rest is clear, I hope.

The program starts with calling in “main” the predicate copy/2

```
    copy(InFile, OutFile):-
```

When we call it, Infile is bound to “source.txt” and Outfile is bound to “destination.txt”. These are files on disk. Then we proceed with the clause of copy/2. First we open both files

```
        InputFile = inputStream_file::openFile8(InFile),
        OutputFile = outputStream_file::create(OutFile),
```

Source.txt is opened as an inputstream_file and the mode is ANSI. Source.txt is bound to the variable Inputfile. From now on when we want to refer to “source.txt” we can simply use the variable Inputfile. In this way you can copy many files by simply changing the names in the call. You could also ask the user for names. For the output stream a new file is created in Unicode mode. It gets the name “destination.txt” and it is bound to the variable Outputfile.
Now we can start copying. Now we call

```
      copyStream(InputFile,OutputFile),
```

This predicae takes care of the copying. And when the copying is done, we close both the input file and the output file

```
        InputFile:close(),
        OutputFile:close().
```

Take good notice of the difference between

```
        InputFile = inputStream_file::openFile8(InFile),
```

and

```
        InputFile:close(),
```

In the first clause you call a predicate from the class “Inputstream_file”. In the second clause you call a predicate from the object “Inputfile”. Clearly when you opened the file, an object was created that can take care of everything concerning that particular stream. Now let’s take a look at the copying. The main code is:

```
    copyStream(InputFile, OutputFile):-
        repeat(),
          String = InputFile:readLine(),
          stdIO::write(String),
          stdIO::nl,
          OutputFile:write(String, "\n"),
        InputFile:endOfStream(),
```

First you start a predicate repeat. It is used for repetition in Prolog and I shall leave it out of the explanation. Then we bind String to a line that is read from “Inputfile”. Again you see that like in real good object oriented tradition we ask the object “Inputfile” to read a line from the stream he is controlling and give it to String. Then we write to the standard IO so you can see in the Messages

Window on the screen what's going on and finally the string is written to the Outputfile. In fact we ask the object "Outputfile" to write the string to the stream he is controlling.

In the example above I used a separate class for the copy predicates. That is not necessary. If you want you can put everything in main.pro. Here is the code. I made a new project, target "console", and changed/added the code in main.pro into:

```
class predicates
   copystream : (inputStream InputFile, outputStream OutputFile) procedure (i,i).
clauses
   copystream(InputFile, OutputFile) :-
      repeat(),
      String = InputFile:readLine(),
      stdIO::write(String), stdIO::nl,
      OutputFile:write(String, "\n"),
      InputFile:endOfStream(), !.
   copystream(_,_).

class predicates
   repeat : () multi ().
clauses
   repeat().
   repeat() :-
      repeat().

clauses
   run():-
      console::init(),
      InputFile = inputStream_file::openFile8("Source.txt"),
      OutputFile = outputStream_File::create("Destination.txt"),
      copystream(InputFile, OutputFile),
      InputFile:close(),
      OutputFile:close(),
      _ = stdIO::readChar(),
      succeed(). % place your own code here
end implement main

goal
   mainExe::run(main::run).
```

This is the way to use streams in VIP. You create an object with every stream that you need and then communicate with that object to get or to give what you want. To help you there are a lot of predicates. I shall show you the ones most used. But I think you should really take a look in the help

file to get an idea of the richness of the class Stream and it's sub classes. Here I only give an overview as I think by now you will understand what the predicates are used for.

Class Stream
In class Stream There are a.o. these predicates.

close : () procedure ().
Closes the stream.

getCRLFconversion : () -> boolean Enabled procedure ().
Return value specifies whether the stream data will be handled with CRLF<->LF conversion (if true) or passed without modification (if false).

getMode : () -> mode Mode procedure ().
Returns current mode of the stream: binary, ANSI or Unicode.

getPosition : () -> unsigned64 Position procedure ().
Returns a stream position, at which the next read or write operation will occur.

setCRLFconversion : (boolean Enabled) procedure (i).
Defines whether the stream data will be handled with CRLF<->LF conversion before writing to stream (LF->CRLF) or after reading from stream (CRLF->LF).

setMode : (mode Mode) procedure (i).
Sets Mode for read/write operations: binary, ANSI or Unicode.

setPosition : (unsigned64 Position) procedure (i).
Sets new stream position, at which the next read or write operation will occur.

Class Inputstream
Here are some predicates from the class inputstream

consult : (factDB FactDB) procedure (i).
Reads facts from a file and asserts them to FactDB

endOfStream : () determ ().
Checks if the stream pointer points at the end of stream.

read : () -> _ Value procedure ().
Reads a specified term from the input stream.

readBytes : (byteCount Size) -> binary Binary procedure (i).
Reads Size bytes from the input stream.

readChar : () -> char Character procedure ().
Reads character from the input stream.

readLine : () -> string Line procedure ().
Reads line from the input stream.

readString : (charCount NumberOfChars) -> string String procedure (i).
Reads NumberOfChars characters from the input stream.

reconsult : (factDB FactDB) procedure (i).
Reads facts from a stream to a specified facts section, reconsult first retracts the fact database.

ungetChar : () procedure ().
Returns last read character back to input stream.

Class Outputstream
Here are some predicates form the class Outputstream

flush : () procedure ().
Forces the contents of the internal stream buffer to be written to resource.

nl : () procedure ().
Writes new line symbols to the output stream.

save : (factDB FactsSectionName) procedure (i).
Save the contents of a named internal facts section in the text stream.

write : (...) procedure (...).
Writes specified variables to output stream.

writeBytes : (pointer Value, byteCount Size) procedure (i,i).
Writes Size bytes to the output stream.

writeQuoted : (...) procedure (...).
Writes specified variables to the output stream. If variable has string(char) domain the prefixed and post fixed double(single) quotes will be written in the stream too.

writef : (string FormatString [formatString], ...) procedure (i,...).
Produces formatted output to the output stream.

Class Inputstream_file
Here are some of the predicates from class InputStream_File. Take note that in fact these predicates are a kind of constructors: they create objects that can be referred to.

openFile : (string FileName).
Opens Unicode file for input.

openFile : (string FileName, stream::mode Mode).
Opens file for input in the specified Mode

openFile8 : (string FileName).
Opens ANSI file for input.

Class Outputstream_File
Here are some of the predicates from class OutputStream_File. Take note that in fact these predicates are a kind of constructors: they create objects that can be referred to.

append : (string FileName).
Opens Unicode file for output, setting the stream position to the end of file.

append : (string FileName, stream::mode Mode).
Opens file for output, setting the stream position to the end of file.

append8 : (string FileName).
Opens ANSI file output, setting the stream position to the end of file.

create : (string FileName).
Creates a new file for output in Unicode mode.

create : (string FileName, stream::mode Mode).
Creates Unicode file for output, and set data output mode to Mode.

create8 : (string FileName).
Creates ANSI file for output.

openFile : (string FileName).
Opens Unicode file for output.

openFile : (string FileName, stream::mode Mode).
Opens file for output, specifying initial stream mode.

openFile8 : (string FileName).
Opens ANSI file for output.

11.8 Files and Directories.

Next to the predicates now for input and output, there is a class with numerous other useful predicates. It is the class “file” with among others the classes “directory” and “file”. These classes (and their interfaces) contain predicates for manipulating directories and files. You may want to take a look in the help file, but for now I leave these classes out of this book.

Chapter 12 More data structures: Stacks, Queues and Trees[18]

In programming we use data structures to store data so they can be used efficiently. That means that for different uses of data, we need different structures. In the previous chapter you have seen several structures to store (or represent) data: facts, functors, databases and lists. In this chapter I want to show you how to represent three other structures that are widely used in programming. They are the stack, the queue and the tree. I shall present ways to implement these structures in VIP and then tell you that partly you can find them in standard classes in VIP. But by then you will understand what they are and how to use the standard classes, when available.

12.1 Data Structures

A data structure is a way of storing data in a computer (program) so that it can be used efficiently. In the design of many types of programs, the choice of data structures is a primary design consideration, as experience in building large systems has shown that the difficulty of implementation and the quality and performance of the final program depends heavily on choosing the best data structure. After the data structures are chosen, the algorithms to be used often become relatively obvious. Sometimes things work in the opposite direction - data structures are chosen because certain key tasks have algorithms that work best with particular data structures. In either case, the choice of appropriate data structures is crucial.

A programming language should offer you several options to implement the data structure that you need. A well-designed data structure allows you to have a variety of operations be performed, using as few resources, both execution time and memory space, as possible. Data structures are implemented using the data types, references and operations on them provided by a programming language. There are many formalized design methods and programming languages in which data structures, rather than algorithms, are the key organizing factor. Most languages feature some sort of module system, allowing data structures to be safely reused in different applications by hiding their verified implementation details behind controlled interfaces. Object-oriented programming languages such as C++, Java and VIP in particular use classes for this purpose.

Later on you will use classes like these as built-in procedures in your program via their interfaces. But for now we will program the data structures as if there are no standard classes. This will give you more insight into the different structures. In this chapter we will take a closer look on lists, stacks, queues and trees. Stacks, queues and trees are new, you have seen lists before. The reason to bring it up again is that I want you to understand that the list is a recursive structure.

12.2 Again: the list.

We've seen that a list is a series of elements between square brackets and separated by comma's. That is a good working definition, but it is only partial true. To give another definition of a list I shall use a way of defining that is called Backus Naur Form, or BNF for short. In BNF a concept is defined by

[18] Large parts of this chapter are inspired by the Wikipedia lemma's on data structure, stack, queue and tree. The sections on trees contains parts of the user manual that came by VIP version 5.0

stating the name, followed by the sign ::=, followed by summing up the possible items that fall under the definition. The possible items are separated by a vertical bar "|", that is read as the logical **or**. The concepts are given a name that makes sense and the name is put between angle brackets < and >. As an example let's look at the definition of an unsigned integer. An unsigned integer is a sequence of digits. A definition could be:

<unsigned integer> ::= <digit> | <digit> <digit> | <digit> <digit> <digit> | ...

The definition says that the concept <unsigned integer> can be a <digit> **or** it can be a <digit> followed by a <digit> **or** it can be three <digits> in a row **or** ...

Additionally we can define a <digit> by:

<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

This says that a <digit> can be one of the natural digits including zero. This definition of <digit> works fine. It is clear that in any place where <digit> appears you can use any of the ten mentioned digits. Maybe you would like to place a restriction on leading zeros, but that is not necessary.

But the first definition, of <unsigned integer>, raises a problem. Immediately you will see that the first definition of <unsigned integer> doesn't work very well. An unsigned integer number can have any number of digits - so this definition will never show every possibility. That would take an infinite number of expressions. The solution for this problem is recursion. In BNF you can create recursive definitions. For an integer the definition becomes:

<unsigned integer> ::= <digit> | <digit> <unsigned integer>

The definition now says that an unsigned integer can be one <digit> **or** it can be a <digit> followed by an <unsigned integer>. In the first option, <unsigned integer> is defined as one <digit>, so this second option can be read as saying that an <unsigned integer> is a <digit> followed by a <digit>. Thus an unsigned integer can be two <digits> at a row. But then the second option can also be read as saying that an <unsigned integer> is a <digit> followed by two <digits>. You can repeat this infinitely. This is called a recursive definition and in fact it says that an <unsigned integer> can be a sequence of any number of <digits>.

Now let's go back to the list and try to give a recursive definition.

<list> ::= [ ] | [<element>] | [<element>, <list>]

This says that a list can be the empty list [ ] **or** it can be a single <element> between square brackets **or** it can be an <element> followed by a comma and a <list>. As a list can be the empty list, the latter could mean:

[<element>, [ ]]

and as a list can also be [<element>] it could also mean

[<element> , [<element>]]

and finally it could even mean

[<element> , [<element>, <list>]]

In the latter case we can extend the <list> recursively into:

[<element> , [<element>, [<element>, [<element>]]]]

and make the list as long as we want. This is what we mean when we say that the list is a recursive structure. We define it with a recursive definition - that creates lists that can have any length that we want or need.

It is only a matter of convenience that we do not write:

    [<element> , [<element>, [<element>, [<element>]]]]

but write

    [<element> , <element>, <element>, <element>]

in stead.

12.3 The Stack

In computer science, a stack is a temporary data type. That means that in a computer program the stack is used at run time, but when the program ends it is not kept like you do with the data in a database. The stack is based on the principle of Last In First Out (LIFO). You can think of a stack as a pile of papers on your desk. Whenever an element is put on the stack, it is put on top. Whenever you need an element from the stack, you take the top element. A frequently used metaphor is the idea of a stack of plates in a spring loaded cafeteria stack. In such a stack, only the top plate is visible and accessible to the user, all other plates remain hidden. As new plates are added, each new plate becomes the top of the stack, hiding each plate below, pushing the stack of plates down. As the top plate is removed from the stack, it can be used and the other plates pop back up, and the former second plate becomes the top of the stack. Two important principles are illustrated by this metaphor: the Last In First Out principle is one; the second is that the contents of the stack are hidden. Only the top plate is visible or accessible, so to see what is on the third plate, we have to remove the first and second plate.

Stacks are used extensively at every level of a modern computer system. For example, a modern PC uses stacks at the architecture level, which are used in the basic design of an operating system for interrupt handling and operating system function calls.

As an abstract data type, the stack is a container of elements and has two basic operations: push and pop. Push adds a given element to the top of the stack leaving previous elements below. Pop removes and returns the current top element of the stack.

Other operations
In modern computer languages, the stack is usually implemented with more operations than just "push" and "pop". The length of a stack can often be returned as a parameter. Another helper operation top (also known as peek or peak) can return the current top element of the stack without removing it from the stack.

In VIP the list is a convenient way to create a stack. The procedures for pop, peek and push are relatively simple. To show how to use a stack in VIP I created a project called “ch12Stack” with target GUI. In this project I created a class (that does not create objects) with the name “stackprocs” to contain the procedures. For the stack I created a facts variable with the name “stacklist. It’s declaration in stackprocs.pro is:

    class facts

```
stacklist : stringlist :=["ddd", "ccc", "bbb", "aaa"].
```

The stacklist was initialized with ["ddd", "ccc", "bbb", "aaa"] so I didn't have to push some elements first. But normally your program will start with an empty stack.

In stackprocs.cl I declared the domain stringlist:

```
domains
   stringlist = string*.
```

In stackprocs.cl I declared the predicates for "pop", "push" et cetera as procedures, but in this case I guess it is equally well to declare the predicates peek and pop as functions

```
predicates
  push : (string Element) procedure (i).
  peek : (string Element) determ (o).
  pop : (string Element) determ (o).
```

The procedure "push" puts an element at the first position of stacklist and returns the new stacklist.

```
  push(Element) :-
    stacklist := [Element | stacklist].    %Put the new element in front of the stack
```

Peek takes a look at the first element and returns it without changing the stack list.

```
  peek(Element) :-
    [Element | _ ] = stacklist.     % take a look at the first Element but don't change stacklist
```

The procedure "pop" takes the first element away from the stack list:

```
  pop(Element) :-
    [Element | Tail] = stacklist,         % Get the first Element and ...
    stacklist := Tail.                    % ... take it off the stacklist
```

There are other procedures possible to work with the stack. I added the following declaration to stackprocs.cl.

```
  dup : () determ.
  swap : () determ.
  showstack : () .
```

The procedure Dup(for duplicate) peeks the top item and pushes it again onto the stack so that an additional copy of the former top item is now on top, with the original below it. In Vip it could be done with:

```
  dup() :-
    [Element | _] = stacklist,                % Get the first Element and ...
    stacklist := [Element | stacklist].       % ... put it in front of the stacklist
```

Swap or exchange: the two topmost items on the stack exchange places. In Prolog:

```
  swap() :-
    [Element1, Element2 | Tail] = stacklist,    % Take the first two elements and ...
    stacklist := [Element2, Element1 | Tail].    % ... change their places
```

For your convenience I added also a procedure to show the stack contents. Normally this is not done as you are only interested in the top element (If not, you wouldn't use a stack :-)).

```
showstack() :-
    stdIO::write("The stack contains: "),
    stdIO::nl,
    stdIO::write(stacklist).
```

As you can see I use the fact that StdIO/... can write lists. Please note also that the predicates shown here are not complete. In any case you should check if the stack is empty before you peek or pop and take appropriate action.

To use these predicates I made a new menu called "stack" in the taskmenu in the taskwindow and added several options like "Push", "Peek", "Pop". The code for these options (I used the code expert to generate the standard code) is:

```
predicates
    onStackPush : window::menuItemListener.
clauses
    onStackPush(_Source, _MenuTag) :-
        Element = vpiCommonDialogs::getString("Give a string", "input a string", "DefaultString"), !,
        stackprocs::push(Element),
        stdIO::write("Element ", Element, " has been pushed"), stdIO::nl.
    onStackPush(_Source, _MenuTag).

predicates
    onStackPeek : window::menuItemListener.
clauses
    onStackPeek(_Source, _MenuTag) :-
        stackprocs::peek(Element), !,
        stdIO::write("the peeked element is ", Element), stdIO::nl.
    onStackPeek(_Source, _MenuTag).

predicates
    onStackPop : window::menuItemListener.
clauses
    onStackPop(_Source, _MenuTag) :-
        stackprocs::pop(Element), !,
        stdIO::write("the popped element is ", Element), stdIO::nl.
    onStackPop(_Source, _MenuTag).
```

I guess that by now you can create the code for the other options yourself. It looks very much alike.

There are more possibilities. Rotate: then the topmost items are moved on the stack in a rotating fashion. For example, if n=3, items 1, 2, and 3 on the stack are moved to positions 2, 3, and 1 on the stack, respectively. Many variants of this operation are possible, with the most common being called left rotate and right rotate. A right rotate will move the first element to the third position, the second

to the first and the third to the second. A left rotate goes the other way around. Here are two equivalent visualizations of these processes:

apple		banana
banana	==right rotate==>	cucumber
cucumber		apple

apple		cucumber
banana	==left rotate==>	apple
cucumber		banana

In Prolog rotation can be done analog to swapping elements. Take a look:

```
rotateRight() :-
        [ E1, E2, E3 | Tail ] = stackList,
        stacklist := [ E2, E3, E1 | Tail ].
```

I assume in this case that the stacklist is a facts variable. If not the stacklist can be given as an argument. You call the pradicate with a Stacklist that must be right rotated:

```
rotateRight(OldStackList, RotatedStacklist)
```

Then the predicate rotateRight/2 may look like:

```
rotateRight( [ E1, E2, E3 | Tail ], [ E2, E3, E1 | Tail ] ) .
```

Calculators employing reverse Polish notation use a stack structure to hold values and give a nice last example. Expressions can be represented in prefix, postfix or infix notations. Conversion from one form of the expression to another form needs a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

For example, The calculation: ((1 + 2) * 4) + 3 can be written down in postfix notation with the advantage of no precedence rules and parentheses needed. In postfix notation ((1 + 2) * 4) + 3 is written as:

```
1 2 + 4 * 3 +
```

The expression is evaluated from the left to right using a stack. We call a number an operand and the signs for addition and multiplication are called operations. The action rules for evaluation are:

- when you encounter an operand, push it onto the stack
- when you encounter an operation, pop two operands, apply the operation to them and push the result onto the stack
- finally, when the input line is empty, pop the (top) element and show it.

The processing of 1 2 + 4 * 3 + goes like this. In this example the Stack is displayed after Operation has taken place.

Input	Operation	Stack
1	Push operand	1
2	Push operand	2, 1
+	Pop two elements, apply operand and push result	3
4	Push operand	4, 3

*	Pop two elements and multiply	12
		/* Note: this is the number twelve */
3	Push operand	3, 12
+	Pop two elements and add	15

At the end of the calculation the final result, 15, lies on the top of the stack.

12.4 The Queue

A queue (pronounced “kju”) is a sequence of elements in which the elements are kept in order and the principal (or only) operations on the sequence are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that whenever an element is added, all elements that were added before have to be removed before the new element can be accessed.

Queues provide services in computer science, transport and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer. The defining attribute of a queue data structure is the fact that it allows access to only the front and back of the structure. Furthermore, elements can only be removed from the front and can only be added to the back. In this way, an appropriate metaphor often used to represent queues is the idea of a checkout line. Other examples of queues are people traveling up an escalator, machine parts on an assembly line, or cars in line at a gas station. The recurring theme will be clear: queues are essentially waiting lines.

In each of the cases, the customer or object at the front of the line was the first one to enter, while at the end of the queue is the last element that entered. Every time a customer finishes paying for his bought items (or a person steps off the escalator, or the machine part is removed from the assembly line, etc.) that object leaves the queue from the front. This represents the queue “pop” function. Every time another object or customer enters the line to wait, they join the line at the end and this represents the “push” function. The queue “size” function would return the length of the line, and the “empty” function will return “true” only if there is nothing in the waiting line.

In Prolog the list is not really a convenient way to implement a queue. The trouble is in the “push”. The program needs to travel all along the waiting elements to the end of the line before the element can be put in place. That’s why I’ll use a fact database to implement a queue. There is one restriction: in VIP the fact database cannot be empty as a queue can. So if you implement a queue in a fact database, be sure to enter a dummy element at the start of the program. When there is a fact database, you can use the standard procedures that work on a database for simulating (or implementing) a queue.

Let’s assume that we need a queue with customers that are going to pay some amount of money. I created a project wit name ch12Queue. In it I created a class (no objects) with the name classQ. In classQ.pro I declared a database with the fact functor “q”.

```
class facts
    q : (string Name, real Amount) nondeterm.
```

Nondeterm means that the fact database can contain. zero, one or more terms. To manipulate the queue I declared four predicates in classQ.cl.

```
predicates
   push : (string Name, real Amount) procedure (i,i).
   peek : (string Name, real Amount) procedure (o,o).
   pop : (string Name, real Amount) procedure (o,o).
   listQ : ().
```

Then I added code for these predicates in classQ.pro. The predicate push is for adding an item at the end of the queue. To push an element, simply assertz it. (Take care to use “assertz”and not “assert”).

```
   push(Name, Amount) :-
      assertz(q(Name, Amount)).
```

To peek the first element (without removing it), simply get it. Due to the fact that the inference engine takes by definition the first item in the database, it goes like this:

```
   peek(Name, Amount) :-
      q(Name, Amount), !.
   peek("0",0).
```

The second clause is to catch the case that the queue is empty. In this case the values that are returned have a special meaning. I take care of this in the menu.

To pop the first element, simply retract it.

```
   pop(Name, Amount) :-
      retract(q(Name, Amount)), !.
   pop("0",0).
```

Variables Name and Amount will contain the values of the first fact in the database. If the queue is empty, the returned value for Name will be “0" (zero).

Finally there is a predicate to show every item in the queue. When the queue is empty, this predicate shows nothing.

```
class predicates
   listAllInQueue : () procedure .
clauses
   listQ() :-
      q(_Name, _Amount), !,                    % Look if there is something in the queue
      listAllInQueue().
   listQ() :-                                  % When the queue is empty, say so
      stdIO::write("nothing"), stdIO::nl.

   listAllInQueue() :-
      foreach q(Name, Amount)
      do stdIO::write(Name, " ", Amount),
         stdIO::nl
      end foreach.
```

The predicate listQ/0 checks if there is ane fact in the queue. If that is the case, then it calls the predicate listAllInQueue/0 to do the job of showing all the items. If the queue is empty, the second clause of listQ/0 gives a warning.

To work with these predicates I created a new menu option in the taskmenu of the taskwindow with the name “Q”. I added four options, “Push”, “Peek”, “Pop” and “ListQ”. Here is the code that I entered/changed after the code expert had generated the standard code.

```
predicates
   onQPush : window::menuItemListener.
clauses
   onQPush(Source, _MenuTag) :-
      NewForm = enterdata::new(Source), NewForm:show().

predicates
   onQPeek : window::menuItemListener.
clauses
   onQPeek(_Source, _MenuTag) :-
      classQ::peek(Name, Amount),
      Name <>"0", !,
      stdIO::write("q(", Name, ", ", Amount, ") is first in the queue"),
      stdIO::nl.
   onQPeek(_Source, _MenuTag) :-
      stdIO::write("Queue is empty"),
      stdIO::nl.

predicates
   onQPop : window::menuItemListener.
clauses
   onQPop(_Source, _MenuTag) :-
      classQ::pop(Name, Amount),
      Name <>"0", !,
      stdIO::write("q(", Name, ", ", Amount, ") is first in the queue and is retracted"),
      stdIO::nl.
   onQPop(_Source, _MenuTag) :-
      stdIO::write("Queue is empty"),
      stdIO::nl.

predicates
   onQListq : window::menuItemListener.
clauses
   onQListq(_Source, _MenuTag) :-
      stdIO::write("The queue contains:"), stdIO::nl,
      classQ::listQ().
```

To enter a new item to the queue I use a form with the name “enterdata”. It contains two edit controls, named “name_ctl” and “amount_ctl”. When the user clicks the <OK> button, the content of these two controls are added to the queue. Here is the code that you should add to the code of the form.

```
predicates
   onOkClick : button::clickResponder.
clauses
   onOkClick(_Source) = button::defaultAction :-
      Name = name_ctl:gettext(),
      Amount = toterm(amount_ctl:gettext()),
      classQ::push(Name, Amount),
      stdIO::write("q(", Name, ", ", Amount, ") is pushed at the end of the queue"), stdIO::nl.
```

Now run the program and try to manipulate the queue. Just for fun I added two more predicates. One to duplicate the first element in the queue and one to swap the first two elements in the queue. Here are the clauses for the predicates that I placed in classQ.pro

```
   duplicate(Name, Amount) :-
      q(Name, Amount), !,
      asserta(q(Name, Amount) ).
   duplicate("0","0").

   swap(Name1, Amount1, Name2, Amount2) :-
      retract(q(Name1, Amount1)),          % remove the first one in the queue
      assert(qout(Name1, Amount1)),        % put the first one in a temporary place
      retract(q(Name2, Amount2)), !,       % try get the second one in the queue
      asserta(q(Name1, Amount1)),          % put them back in the queue in reversed order
      asserta(q(Name2, Amount2)),
      retract(qout(Name1, Amount1)).       % remove the temporary fact
   swap("0",0,"0",0) :-
      retract(qout(Name1, Amount1)),       % remove the temporary item
      asserta(q(Name1, Amount1)).          % restore it's position in the queue
```

The declarations in classQ.cl are:

```
   duplicate : (string Name, real Amount) procedure (o,o).
   swap : (string Name1, real Amount1, string Name2, real Amount2) determ (o,o,o,o).
```

In the predicate duplicate/2 I use a temporary fact to keep the first element in the queue. You have to declare that one as a class fact in the file “classQ.pro”. The declaration becomes:

```
class facts
        q : (string Name, real Amount) nondeterm.
        qout : (string Name, real Amount) determ.
```

The type of qout is determ as we are going to store exactly one fact in it.

To use the predicates, add two more options to the taskmenu. I called them Duplicate and Swap. Here is how to change the code that the Code Expert generates in taskmenu.pro.

```
predicates
   onQDuplicate : window::menuItemListener.
```

```
clauses
   onQDuplicate(_Source, _MenuTag) :-
      classQ::duplicate(Name, Address),
      Name <> "0", !,
      stdIO::write(Name, " ", Address, " has been duplicated"), stdIO::nl.
   onQDuplicate(_Source, _MenuTag) :-
      stdIO::write("Queue is empty, nothing to duplicate"), stdIO::nl.

predicates
   onQSwap : window::menuItemListener.
clauses
   onQSwap(_Source, _MenuTag) :-
      classQ::swap(Name1, Address1, Name2, Address2),
      Name1 <> "0", !,
      stdIO::write("(",Name1, " ", Address1, ") and (", Name2, "", Address2, ") have been swapped"),
stdIO::nl.
   onQSwap(_Source, _MenuTag) :-
      stdIO::write("Queue contains less than two items. Nothing to swap"), stdIO::nl.
```

Maybe it is a good idea to play a while with stacks and queues to get to grips with their opportunities.

12.5 Trees

Not only can clauses be recursive; so can data structures, as we have seen with the list. As a matter of fact, Prolog is the only widely used programming language that allows you to define recursive data types. A data type is recursive if it allows structures to contain other structures like themselves. Compare a recursive clause that calls itself within itself.

The most basic recursive data type is the list, although it doesn't immediately look recursively constructed. A lot of list-processing power is built into Prolog, lists were discussed in chapter 10 and in section 12.1. In this section (and the following ones) I will discuss another recursive data type, implement it, and use it to show you a very fast sorting program. The structure of this recursive data type is a tree (Figure 12.1). Crucially, each branch of the tree is itself a tree; that's why the structure is recursive. Figure 12.1 shows an example of a tree relating members of a family. You may read this tree as Cathy is a child of Michael and Melody, Michael is a child of Charles and Hazel, et cetera. Maybe you are used to draw a diagram like this with the parents above the children, but upside like in figure 12.1 is equally possible.

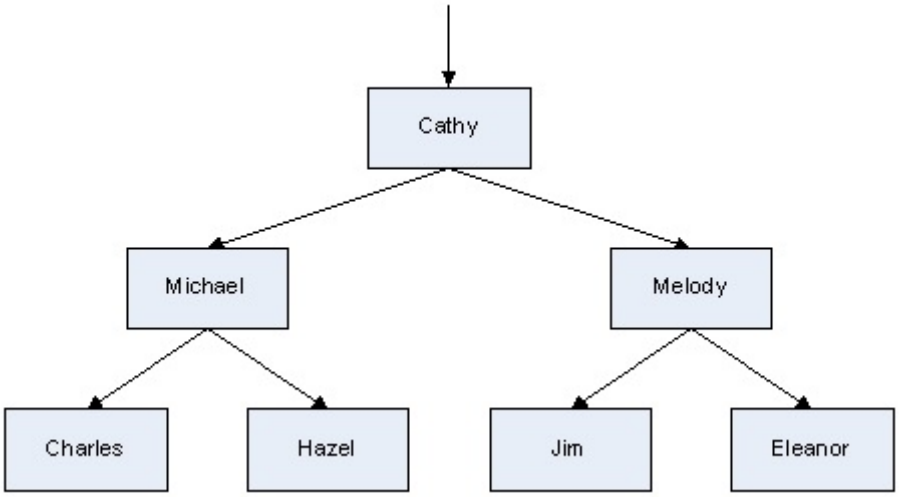


Figure 12.1 Part of a family tree

Trees are important for sorting, but are also widely used in problem solving (e.g. in Artificial Intelligence) where you use a tree-like structure to find a solution in a specific problem domain. A tree consists of so-called nodes and links. In figure 12.1 the rectangles represent nodes and the arrows represent links. Nodes have names to identify them. So you can say that node Cathy is linked to node Michael and node Melody. When we talk about trees (in treetalk) we say that Michael is the left link of Cathy and Melody is the right link of Cathy. Cathy is also called the parent node of Michael and Melody and in reverse they are the child nodes of node Cathy. On its turn node Michael is the parent node of nodes Charles and Hazel and they are the child nodes of node Michael. In A tree every node has at least a parent node and a child node. Exceptions are the topmost node of the tree, it has no parent, and the bottom nodes, they have no children. The top node is called the root of the tree, the bottom nodes are called the leaves of the tree. In figure 12.1 every node has exactly two children, but that is not a condition for a tree. On the other hand, in a tree a node has one and only one parent, except again the root node.

When you take a look at the tree in figure 12.1, the recursive character of a tree will be clear. Cathy is the root node of the complete tree, but Michael is the root node of a subtree, consisting of nodes Michael, Charles and Hazel. This is true for any tree: a node with all the nodes below it together form a tree.

In this section I will restrict myself to a special kind of tree that is widely used: the binary tree. A binary tree is a tree in which every node has at most two children. In a binary tree the children are called the left and the right child. Let's define some more concepts[19].

- The depth of a node n is the length of the path from the root to the node. So in figure 12.1 node Charles has depth=2.
- The set of all nodes at a given depth is sometimes called a level of the tree. Nodes Michael and Melody are at the same level
- The root node is at depth zero.
- The height of a tree is the depth of its furthest leaf. The height of the tree in figure 12.1 is 2.

[19] from Wikipedia.

- A tree with only a root node has a height of zero.
- Siblings are nodes that share the same parent node. Nodes Michael and Melody are siblings.
- If a path exists from node p to node q, where node p is closer to the root node than q, then p is an ancestor of q and q is a descendant of p. So Cathy is an ancestor of every other node in the tree and Charles is a descendant of Michael.
- The size of a node is the number of descendants it has including itself.
- In-degree of a node is the number of links arriving at that node from ancestors. The root is the only node in the tree with in-degree = 0.
- Out-degree of a node is the number of links leaving that node to descendants. In a binary tree the out-degree can be 0, 1 or 2.

12.6 Trees as a Data Type

Recursive types were popularized by Niklaus Wirth in his book “Algorithms + Data Structures = Programs”. Wirth derived Pascal from ALGOL60 and published his work in the early 70's of the last century. He didn't implement recursive types in Pascal, but he did discuss what it would be like to have them. Visual Prolog allows truly recursive type definitions. For example, you can define a tree as follows:

```
domains
    treetype = tree(string, treetype, treetype)
```

This declaration says that a variable of type “treetype” will be written as the functor “tree” whose first argument is a string (that could represent the name of the node) and whose second and third argument are of treetype. We can interpret the second argument as the left subtree and the third argument as the right subtree. In this way the treetype consists of “things” (we call them nodes) that have a name and are connected (left and right) to other “things” (nodes) of treetype. Compare figure 12.1. But this is not enough. This declaration provides no way to end the recursion, and, in real life, the tree does not go on forever. Some nodes, the leaves, don't have links to further subtrees. We have to take care of that in the declaration. That’s why we define two kinds of nodes in a tree: ordinary nodes like the ones defined in treetype above that have a left and right subtree and nodes that do not have a left and right subtree. This is done by allowing a treetype to have one of two functors: “tree” with three arguments, or “empty” with no arguments. So the declaration of a treetype becomes:

```
domains
    treetype = tree(string, treetype, treetype) or empty
```

Remember that “or” is used instead of a semicolon. Notice that the names “tree” (a functor taking three arguments) and “empty” (a functor taking no arguments) are created by the programmer; neither of them has any pre-defined meaning in Prolog. You could equally well have used xxx and yyy, but of course that would not really improve the understandability of the declaration.

This is a good moment to realize that in fact we just defined a tree with two kinds of nodes: nodes that have a subtree and nodes that have not. We’ve better paraphrase that in another way: the nodes that don’t have a subtree are of treetype, but with two subtrees “empty”. So node Charles in figure 12.1 could be written as:

```
tree(“Charles”, empty, empty).
```

This indicates that the node has name “Charles” and that its left and right subtree are empty. This gives a way to recognize a leave: a node with two empty subtrees is a leave.

The subtree starting with Michael can be written as:

```
tree(“Michael”,                    % the name of the (root)node
     tree(“Charles”, empty, empty), % the left subtree
     tree(“Hazel”, empty, empty)    % the right subtree
     ).                             % the finishing bracket
```

Of course we can see that the left and right subtrees are only leaves, but the structure is the same as for the other nodes. The integral tree from figure 12.1 can be written as:

```
tree("Cathy",
     tree("Michael",                    % the node of the left subtree
          tree("Charles", empty, empty), % the left subtree of the left subtree
          tree("Hazel", empty, empty))   % the right subtree of the left subtree
     tree("Melody",                     % the node of the right subtree
          tree("Jim", empty, empty),    % the left subtree of the right subtree
          tree("Eleanor", empty, empty))) % the right subtree of the right
                                          subtree
```

This is indented here for readability, but Prolog does not require indentation, nor are trees indented when you print them out normally. Another way of writing this same data structure is:

```
tree("Cathy"
     tree("Michael", tree("Charles", empty, empty), tree("Hazel", empty, empty))
     tree("Melody", tree("Jim", empty, empty), tree("Eleanor", empty, empty)))
```

You can see that the structure is defined by the functors, the commas and the parentheses. The layout is only for the human eye. Make sure you understand that this is not a Prolog clause; it is just a complex data structure.

By the way, this looks not really very efficient. It is more or less clumsy to represent the tree in the way we do it . When using a tree, you will probably act differently. You will let the computer program create a tree, use it and then discard it without you ever really seeing the tree. For the computer program the tree is a pointer structure that it can easily maintain and traverse.

12.7 Traversing a Tree

Before going on to the discussion of how to create trees, first consider what you'll do with a tree once you have it. One of the most frequent tree operations is to examine all the nodes and process them in some way. You could be looking for a particular value (say a solution to a problem) or you could be collecting all the values in the tree. This is known as traversing the tree. One basic algorithm for doing so is the following three step algorithm:

1. If the tree is empty, do nothing. (In fact here you encountered the empty subtree of a leaf and with an empty tree there is nothing to do).
2. Otherwise, process the current node, then traverse the left subtree.
3. When the left subtree is done, then traverse the right subtree.

Like the tree itself, the algorithm is recursive: it treats the left and right subtrees exactly like the original tree. Prolog expresses it with two clauses, one for empty and one for nonempty trees:

```
traverse(empty).      /* do nothing */
traverse(tree(Node, Left, Right)) :-
   do_something_with_Node,          % process the node
   traverse(Left),                  % process the left subtree
   traverse(Right).                 % process the right subtree
```

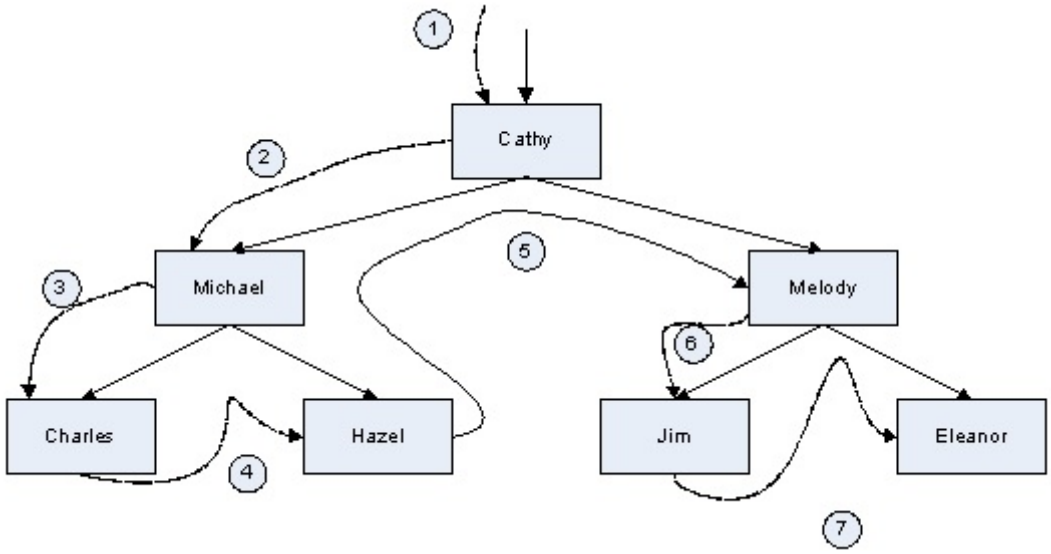


Figure 12.2 depth-first tree traversal

This tree traversal algorithm is known as depth-first search because it goes as far as possible down each branch before backing up and trying another branch (Figure 12.2). To see it in action, look at following program that traverses a tree and prints all the elements as it encounters them.

For this program I created a new project, target console, and entered the declarations in “main.cl”:

```
domains
   treetype = tree(string, treetype, treetype) ; empty().

predicates
  traverse : (treetype).
```

Then in “main.pro” I added these clauses

```
implement main
  open core, stdIO

clauses
  traverse(empty).
  traverse(tree(Name,Left,Right)):-
```

```
        write(Name,'\n'),
        traverse(Left),
        traverse(Right).

clauses
    run():-
        console::init(),
        FamilyTree = tree("Cathy",
                          tree("Michael", tree("Charles", empty, empty), tree("Hazel", empty, empty)),
                          tree("Melody", tree("Jim", empty, empty), tree("Eleanor", empty, empty))),
        traverse(FamilyTree),
        _ = stdIO::readchar(),
        succeed(). % place your own code here
end implement main
```

Given the tree in Figures 12.1 and 12.2, program ch12e01 prints

```
Cathy
Michael
Charles
Hazel
Melody
Jim
Eleanor
```

Of course, you could easily adapt the program to perform some other operation on the elements, rather than printing them.

Depth-first search is strikingly similar to the way Prolog searches a knowledge base, arranging the clauses into a tree and pursuing each branch until a query fails.

If you wanted to, you could describe the tree by means of a set of Prolog clauses such as:

```
        father_of("Cathy", "Michael").
        mother_of("Cathy", "Melody").
        father_of("Michael", "Charles").
        mother_of("Michael", "Hazel").
...
```

So you have two different ways to describe the family: with a tree structure or with separate clauses. The last one is preferable if the only purpose of the tree is to express relationships between individuals. But this kind of description makes it impossible to treat the whole tree as a single complex data structure; and as you'll see, complex data structures are very useful because they simplify difficult computational tasks.

12.8 Creating a Tree

One way to create a tree is to write down a nested structure of functors and arguments, as we did in the preceding example. But this is not a good way. It is too easy to make mistakes and you cannot change the tree dynamically. So ordinarily we create trees by computation. We start by creating a tree

that consists only of the root node and then successively add the next nodes. When you have to add a node in a tree, you first have to find the place where to add the node. So you go down the branches to the leave where the new node is to be inserted and there the insertion takes place. The program to insert a new node acts in the same way. In each step, an empty subtree is replaced by a nonempty one through Prolog's process of unification.

To create a tree we need a predicate. Let's call it "createTree". To create a one-cell tree from an ordinary data item is trivial:

```
create_tree(N, tree(N, empty, empty)).
```

This says: "If we want to insert a node N in an empty tree, then the result is a node with two empty subtrees and we write tree(N, empty, empty) that is a one-cell tree containing node N." Building a tree structure is almost as easy. The following procedure takes three arguments. Argument one and two are for input (the new node and the existing tree) en argument three is for output (the new tree). Each argument is a tree. The procedure inserts the first tree as the left subtree of the second tree, giving the third tree as the result:

```
insert_left(X, tree(A, _, B), tree(A, X, B)).          /* (i,i,o) */
```

Notice that this rule has no body--there are no explicit steps in executing it. All the computer has to do is match the arguments with each other in the proper positions, and the work is done.
Suppose, for example, you want to insert tree("Michael", empty, empty) as the left subtree of tree("Cathy", empty, empty). To do this, just execute the clause

```
insert_left(tree("Michael", empty, empty),
        tree("Cathy", empty, empty),
        NewTree)
```

and NewTree is immediately bound to

```
tree("Cathy", tree("Michael", empty, empty), empty).
```

This gives a way to build up trees step-by-step. The next program below demonstrates this technique. In the program the items to be inserted are given in the clause "run()". In real life, the items to be inserted into the tree will come from some external input, like a list of items or from a database.

I created a new project (target console) en declared the domain and the predicates in "main.cl"

```
domains
   treetype = tree(string,treetype,treetype) ; empty().

predicates
   create_tree: (string,treetype) procedure (i,o).
   insert_left : (treetype,treetype,treetype) determ (i,i,o).
   insert_right : (treetype, treetype, treetype) determ (i,i,o).
```

Then in "main.pro" I added the following clauses.

```
clauses
   create_tree(A,tree(A,empty,empty)).
   insert_left(X,tree(A,_,B),tree(A,X,B)).
```

```
        insert_right(X,tree(A,B,_),tree(A,B,X)).

    clauses
        run():-
            console::init(),
/* First create some one-cell trees... */
        create_tree("Charles",CharlesTree),
        create_tree("Hazel",HazelTree),
        create_tree("Michael",MichaelTree),
        create_tree("Jim",JimTree),
        create_tree("Eleanor",EleanorTree),
        create_tree("Melody",MelodyTree),
        create_tree("Cathy",CathyTree),
/* ...then link them up... */
        insert_left(CharlesTree, MichaelTree, Mi2),
        insert_right(HazelTree, Mi2Tree, Mi3),
        insert_left(JimTree, MelodyTree, Me2),
        insert_right(EleanorTree, Me2, Me3),
        insert_left(Mi3, CathyTree, Ca2),
        insert_right(Me3, Ca2, Ca3),
/* ...and print the result. */
        stdIO::write(Ca3,'\n'),
        _ = stdIO::readChar(), !,
        succeed(). % place your own code here
    run().
end implement main
```

The program prints the tree as it is built. You may want to extend the program with a procedure that writes the tree in a better readable way.

A problem with this program is that we use a lot of variables when we build the tree. That is because there is no way to change the value of a Prolog variable once it is bound. That's why this program uses so many variable names; every time you create a new tree, you need a new variable. The large number of variable names here is unusual; more commonly, repetitive procedures obtain new variables by invoking themselves recursively, since each invocation has a distinct set of variables.

12.9 Binary Search Trees

So far, we have been using the tree to represent relationships between its elements. This is not the best use for trees, since Prolog clauses can do the same job. But trees have other uses.
Trees provide a good way to store data items so that they can be found quickly. A tree built for this purpose is called a search tree; from the user's point of view, the tree structure carries no information--the tree is merely a faster alternative to a list or array. Recall that, to traverse an ordinary tree, you look at the current node (cell) and then at both of its subtrees. When the tree is adequately structured, you only have to search one of the subtrees. But that is not always possible, sometimes you

have to search the whole tree to find a particular item. In that case you have to look at every cell in the whole tree.

The time taken to search an ordinary tree with N elements is, on the average, proportional to N.
A binary search tree is constructed so that you can predict, upon looking at any cell, which of its subtrees a given item will be in. This is done by defining an ordering relation on the data items, such as alphabetical or numerical order. Items in the left subtree precede the item in the current cell and, in the right subtree, they follow it. Figure 12.3 shows an example. Note that the same names, added in a different order, would produce a somewhat different tree. Notice also that, although there are ten names in the tree, you can find any of them in--at most--five steps, starting from the top and deciding in every node to take the left or right subtree.

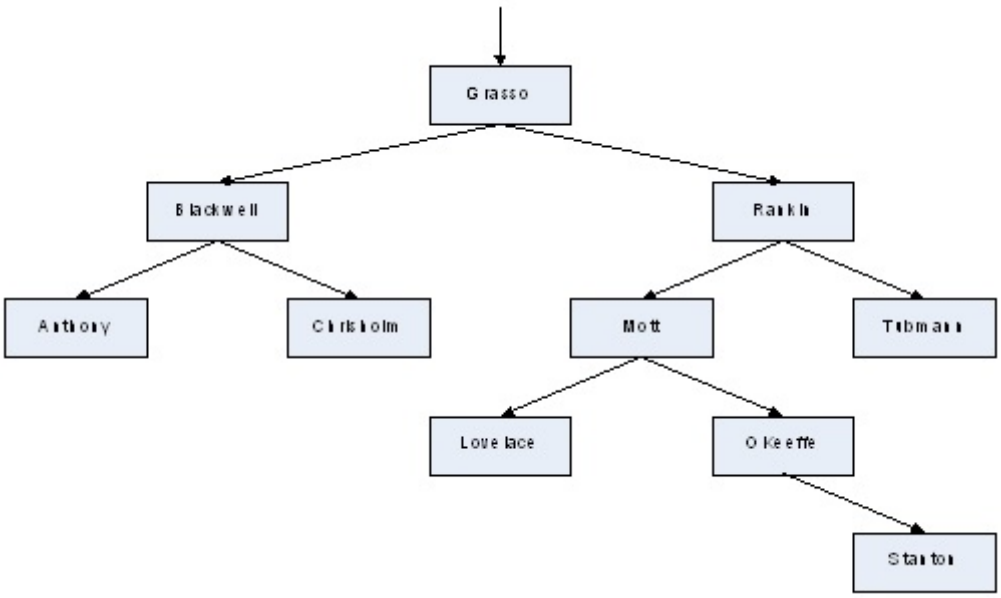


Figure 12.3 Binary Search Tree

Searching a binary tree for a specific value is a process that can be performed recursively because of the order in which values are stored. E.g. suppose a tree contains the names and addresses of our friends. To search for an address, we begin by examining the name in the root node. If the value we are searching for equals the root, we have found what we were looking for. If it is less than the root (with names that is the name we are looking for precedes the name in the root), then it must be in the left subtree, so we recursively search the left subtree in the same manner. Similarly, if it is greater than the root, then it must be in the right subtree, so we recursively search the right subtree. If we reach a leaf and have not found what we were looking for, then the address is not where it would be if it were present, so it is not in the tree at all.

Every time you look at a cell in a binary search tree during a search, you eliminate half the remaining cells from consideration, and the search proceeds very quickly. If the size of the tree (that is the number of nodes) were doubled, then, typically, only one extra step would be needed to search it.
The time taken to find an item in a binary search tree is, on the average, proportional to $\log_2 N$ (or, in fact, proportional to log N with logarithms to any base).

To build the tree, you start with an empty tree and add items one by one. The procedure for adding an item is the same as for finding one: you simply search for the place where it ought to be, and insert it there. The algorithm is as follows:

1 If the current node is an empty tree, insert the item there.

2 Otherwise, compare the item to be inserted and the item stored in the current node. If the item is less than the item in the current node, insert it into the left subtree. Otherwise insert it in the right subtree.

In Prolog, this requires three clauses, one for each situation. We suppose that there is already a tree that can be described with the functor:

tree(Node, LeftTree, RightTree) or empty

and that we are going to insert new items in this tree. We give each clause for insertion the procedure name “insert” and use three arguments: the item_to_be_inserted, the node that we look at and the resulting new tree with the item inserted in it. The first clause is

```
insert(NewItem, empty, tree(NewItem, empty, empty) :- !.          /* (i,i,o) */
```

Translated into natural language, this code says "The result of inserting NewItem into a node that is (an) empty (tree) is tree(NewItem, empty, empty)." The cut ensures that, if this clause can be used successfully, no other clauses will be tried for insertion of the element NewItem.

The second and third clauses take care of insertion into nonempty trees. Both have the flow pattern (i,i,o):

```
insert(NewItem, tree(Element, Left, Right), tree(Element, NewLeft, Right) :-
   NewItem <= Element,!,
   insert(NewItem, Left, NewLeft).
insert(NewItem, tree(Element, Left, Right), tree(Element, Left, NewRight) :-
        insert(NewItem, Right, NewRight).
```

If NewItem <= Element, you insert it into the left subtree; otherwise, you insert it into the right subtree. Notice that, because of the cuts, you get to the third clause only if neither of the preceding clauses has succeeded. Also notice how much of the work is done by matching arguments in the head of the rule.

12.10 Other tree traversals

In computer science, tree-traversal refers to the process of visiting each node in a tree data structure, exactly once, in a systematic way. Such traversals are classified by the order in which the nodes are visited. Linear data structures like lists and one dimensional arrays have only one logical means of traversal: element by element in the order they are stored. But tree structures can be traversed in many different ways. Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps are:

- Performing an action on the current node (referred to as "visiting" the node)
- Repeating the traversal process with the subtrees rooted at the left child
- Repeating the traversal process with the subtrees rooted at the right child.

The three ways are called pre-order traversal, in-order traversal and post-order traversal. The process is most easily described through recursion.

To traverse a non-empty binary tree in pre-order, perform the following operations:

Visit the root.
Traverse the left subtree.
Traverse the right subtree.

This way is called depth-first traversal or depth-first search and it was described in the previous section. When we pre-order travers the example tree in figure 12.4 we will visit the nodes (and probably process them) in the order: F, B, A, D, C, E, G, I, H

To traverse a non-empty binary tree in in-order, perform the following operations:

Traverse the left subtree.
Visit the root.
Traverse the right subtree.

In-order traversing the tree in figure 12.4 gives the sequence: A, B, C, D, E, F, G, H, I
Note that the in-order traversal of a binary search tree yields an ordered list. This is because of the way the tree is built.

To traverse a non-empty binary tree in post-order, perform the following operations:

Traverse the left subtree.
Traverse the right subtree.
Visit the root.

Post-order traversal sequence is: A, C, E, D, B, H, I, G, F

Finally, trees can also be traversed in level-order, where we visit every node on a certain level before going to a lower level. This is also called breadth-first traversal. Level-order traversal sequence is: F, B, G, A, D, I, C, E, H..

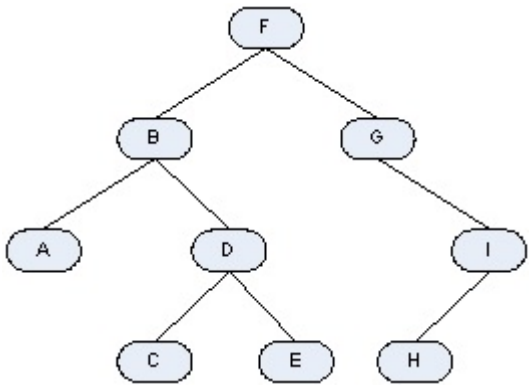


Figure 12.4 an example tree

12.11 A program for tree traversal

Now let's make the program that creates the tree from figure 12.4 and that traverses this tree in several ways. I started a new project with the name ch12Trees. I added an extra option to the taskmenu in the task window called "Trees" and I added four options: "create", "traverse PRE-order", "traverse IN-order"and "traverse POST-order". I made the IDE generate the standard code and then changed it into this code:

```
predicates
   onTreesCreate : window::menuItemListener.
clauses
   onTreesCreate(_Source, _MenuTag) :-
      classtree::add_from_list(["F", "B", "G", "A", "D", "I", "C", "E", "H"]),
      stdIO::write("Tree has been created"), stdIO::nl.
```

```
predicates
   onTreesTraversePreorder : window::menuItemListener.
clauses
   onTreesTraversePreorder(_Source, _MenuTag) :-
      classtree::traverse("preorder").

predicates
   onTreesTraverseInorder : window::menuItemListener.
clauses
   onTreesTraverseInorder(_Source, _MenuTag) :-
      classtree::traverse("inorder").

predicates
   onTreesTraversePostorder : window::menuItemListener.
clauses
   onTreesTraversePostorder(_Source, _MenuTag):-
      classtree::traverse("postorder").
```

I put the tree and all the predicates for manipulating the tree in the class “classTree”. So every menu option calls a predicate from that class. To create the tree, I give a listof characters in random order. The predicate “add_from_list()” then creates a tree. To traverse the tree the predicate “traverse()” is called. It has an argument that indicates (as a string) how to traverse the tree.

The domains and the predicates for manipulating the tree are declared in classtree.cl. It looks like this.:

```
        domains
           stringlist = string*.
           treetype = tree(string, treetype, treetype) ; empty.

        predicates
           add_from_list : (stringlist) procedure (i).
           treeinsert : (string NewItem, treetype OldTree, treetype Newtree) procedure (i,i,o).
           traverse : (string TraverseOrder) procedure (i).
           preOrderTraverse : (treetype Tree) procedure (i).
           inOrderTraverse : ( treetype Tree) procedure (i).
           postOrdertraverse : (treetype Tree) procedure (i).
```

The tree is kept in a class fact variable, that is declared in classtree.pro in this way:

```
class facts
   treefact : treetype := empty.
```

It is a fact variable of the type treetype and it is initialized as empty. In classtree.pro I also entered the code for the predicates.

```
clauses
   add_from_list([]) :- !.
   add_from_list([NewItem | Tail]) :-
```

```
    treeinsert(NewItem, treefact, NewTree),
    treefact := Newtree,
    add_from_list(Tail).

treeinsert(NewItem, empty, tree(NewItem, empty, empty)) :- !.
treeinsert(NewItem, tree(Element, LEft, Right), tree(Element, Newleft, Right) ) :-
    NewItem < Element, !,
    treeinsert(NewItem, Left, NewLeft).
treeinsert(NewItem, tree(Element, LEft, Right), tree(Element, Left, Newright) ) :-
    treeinsert(NewItem, Right, NewRight).

traverse("preorder") :-
    stdIO::nl, stdIO::write("the tree in PRE-order:"),
    preOrderTraverse(treefact), !, stdIO::nl.
traverse("inorder") :-
     stdIO::nl, stdIO::write("the tree in IN-order:"),
    inOrderTraverse(treefact), !, stdIO::nl.
traverse("postorder") :-
    stdIO::nl, stdIO::write("the tree in POST-order:"),
    postOrderTraverse(treefact), !, stdIO::nl.
traverse(_).

preOrderTraverse(empty) :- !.
preOrderTraverse(tree(Element, Left, Right)) :-
    stdIO::write(Element, " "),
    preOrderTraverse(Left),
    preOrderTraverse(Right).

inOrderTraverse(empty) :- !.
inOrderTraverse(tree(Element, Left, Right)) :-
    inOrderTraverse(Left),
    stdIO::write(Element, " "),
    inOrderTraverse(Right).

postOrderTraverse(empty) :- !.
postOrderTraverse(tree(Element, Left, Right)) :-
    postOrderTraverse(Left),
    postOrderTraverse(Right),
    stdIO::write(Element, " ").
```

Enjoy the program. You may want to experiment with the program. E.g. in the program as it is, a new element is placed in the left tree when it is smaller then or equal to the node element. See what happens when you change that. Change

```
    treeinsert(NewItem, tree(Element, Left, Right), tree(Element, Newleft, Right) ) :-
        NewItem <= Element, !,
        treeinsert(NewItem, Left, NewLeft).
```

Into:

```
treeinsert(NewItem, tree(Element, LEft, Right), tree(Element, Newleft, Right) ) :-
    NewItem < Element, !,                    % the change is in this line
    treeinsert(NewItem, Left, NewLeft).
```

Especially note what happens when you create the tree twice. Then the insertions and traversals change.

Appendix A1. Everything about Dialogs and Forms

This appendix gives a complete overview for editing Dialogs and Forms. The first section is on creating a Dialog and editing its attributes, the next section is about designing the Dialog and the controls, the third section focuses on the Control Properties. The text is mainly a copy of parts of the VIP Help File. In the text the words Dialog, Form, Window refer to the same thing: a window where the user can click and edit controls. Sometimes this window is called a container as it contains the controls.

In the Project Tree you can recognize files by their extension. A Dialog has extension ".dlg", a Form has extension ".frm". There are little differences between Forms and Dialogs. One difference is that a Dialog is not resizable at run time. With a Form you can specify the border and this specifies if the Form can be resized. In this appendix I shall use the words Dialog and Form as if there are no differences.

A1.1 Create a Dialog or a Form

New Form Creating

Forms can be created in any project with target GUI Graphical User Interface. They can be created both in projects using the Conventional GUI (pfc/vpi) and the Object GUI (pfc/gui) types of UI Strategy . Notice that windows can be created only in Graphical User Interface projects, which use the Conventional GUI (pfc/vpi) type of UI Strategy . (See description of forms in the GUI package overview in the PFC part of Help.)

To create and register a new form in a project you need to click the File/New menu command. The IDE opens the Create Project Item Dialog (see figure A.1) that has on the left side a list of items that can be created with this dialog. You should select the type Form to create a Form or the type Dialog to create a Dialog. When you select type Form the Create Project Item dialog takes the shape as shown in figure A1.1.

After selecting the type Form the following fields are shown that you can edit.

Name
In the Name field you should type the name, which will be associated with this form in the automatically generated code. This name should be a correct Visual Prolog name. It should be any sequence of letters, digits, and underscores, beginning with a lowercase letter. When you build the project after you create a file, the IDE generates several files to handle the form. Each of these files will have the Name that you enter here; the files will only differ in their extensions. These files appear in the Project Tree in the Project window. Later you will find the Name in the Title Bar of the Form.

Package
In the Package list button you should select one of the packages known to the project (the main.pack in figure A1.1) or a new package that will be created. The files that are generated for handling the created form will be included into this package. In figure A1.1 we choose as existing package the project itself, so the files will be added to "main.pack".

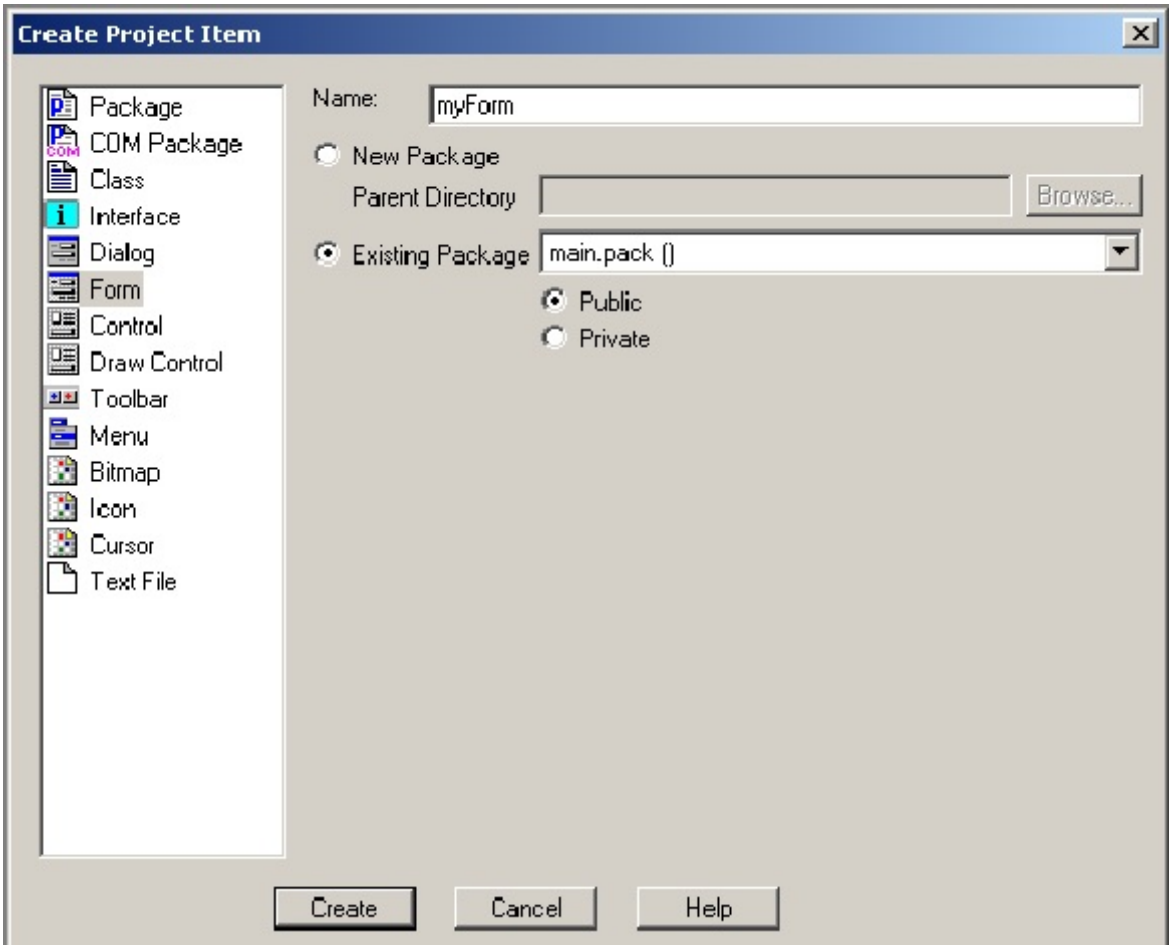


Figure A1.1 Creating a Form

Public or Private

A Form is treated as a class. That means that a.o. two files are generated, <Name>.i and <Name>.cl where <Name> is the name you gave to the Form. The file <Name>.i is called the interface, the file <Name>.cl is called the class declaration. Both files contain the predicate declarations that are seen by other parts of the project. You should check one of the Public or Private radio buttons to specify whether the generated interface and class, which handle the form, will be *public* or *private* to the package in which the form is included.

If you check the Public radio button, then the include directives for the generated interface and class will be added to the header file (in figure A1.1 main.ph) of the specified package. In this file these lines will be inserted:

```
% exported interfaces
#include @"myForm.i"
% exported classes
#include @"myForm.cl"
```

Since the package header file will be inserted by the #include @"main.ph" directive in all places, where the package is used, the result is that in all these places also the #include @"myForm.i" and #include @"myForm.cl" directives will be seen. This means that the predicates that are declared in the

interface and class that handle the “myForm” form are callable from any place in the project and hence they are public.

If you check the Private radio button, then the include directives will be added to the package implementation file (in figure A.1: main.pack) of the specified package. In that file these lines will be added:

```
% private interfaces
#include @"myForm.i"
% private classes
#include @"myForm.cl"
```

These include statements are only seen within the package itself. Therefore, the interface and the class declarations will not be seen outside of the package implementation. Hence the interface and class that handle the “myForm” form are private to the package implementation and the predicates in them are private.

You should realize that in the above the name “main” is more or less accidentally chosen. The essential difference is shown in the extensions “.PH” for Package Header and “.PACK” for the package.

After you fill in all required settings, you should press the Create button, then the Form editor appears with a quite standard prototype of a form. See figure A1.2

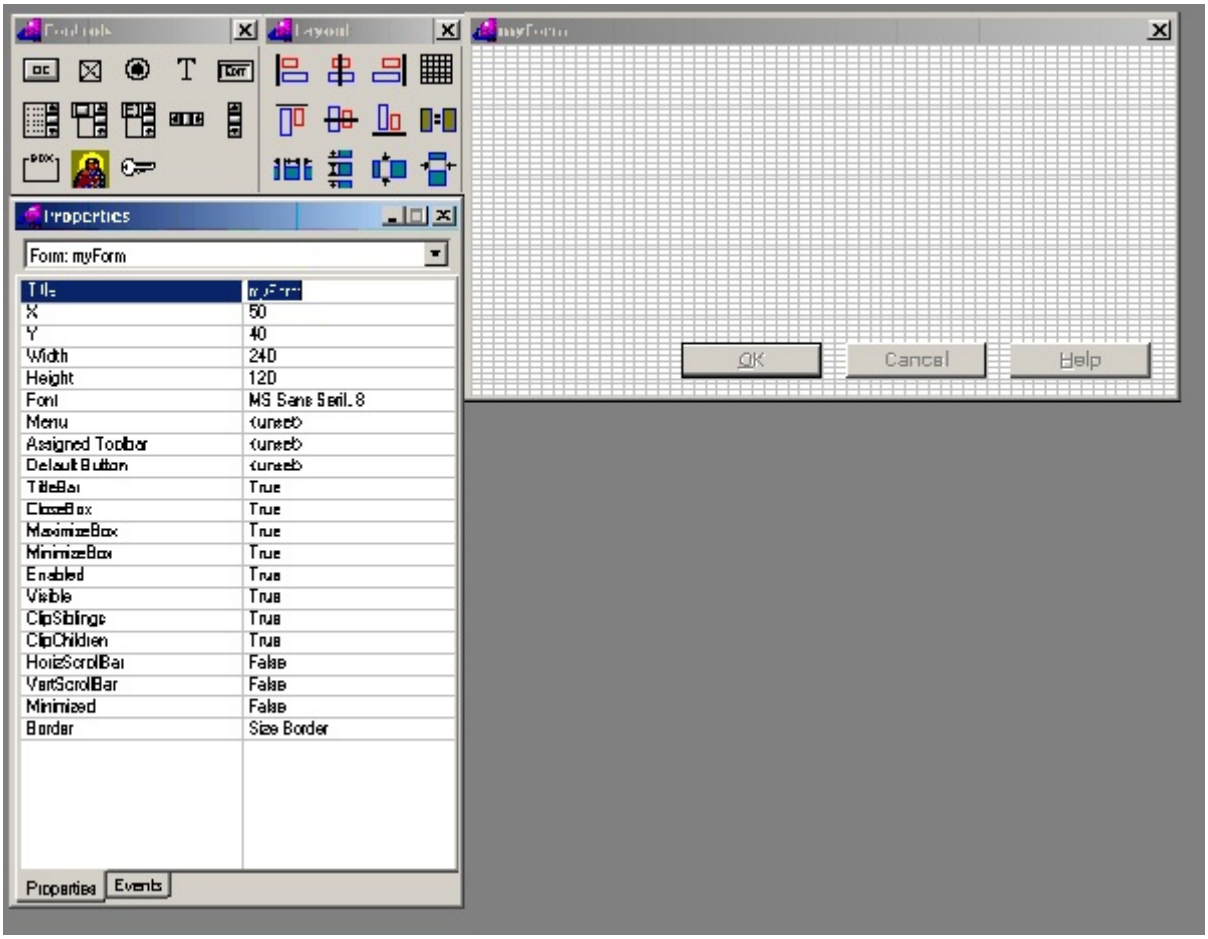


Figure A1.2 the Forms Editor

A1.2 Edit a Dialog

To edit a Dialog, in the Project Tree (in the Project window) double click the name of a file with a dialog filename extension ".dlg". To edit a form you have to choose the file with the extension ".frm". An alternative is to highlight the name of the file and then hit Enter. The IDE Form Editor (also called the Designer) appears and you can edit whatever form of a dialog you wish.

When a new dialog is created, it will by default have three push button controls: OK, Cancel and Help. These can freely be rearranged or deleted. But take care. These buttons are supposed to be used in the usual MsWindows tradition. That means that when you keep these buttons, a click on <OK> and <Cancel> will close the Dialog, no matter what code you add to these buttons yourself.

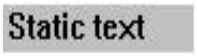
Controls

A control is a part of a Dialog that the user can use to perform some activity. The best known controls are for choosing (e.g. a file name), clicking (e.g. the <OK> button) and for editing (e.g. entering or changing a name). Each control in a Dialog must have an identifying name that ends with "_ctl", and that is unique within that dialog. While two controls within one dialog must not have the same name it is actually good practice when controls in different dialogs have the same name when they perform the same action. The table on the next page shows the types of control that are available in VIP.

A Push Button control serves to initiate a specialized action in an application.

A Check Box control lets you indicate a choice among two alternatives. For example some facility may be switched ON or switched OFF.

A group of Radio Button controls serves to indicate one choice from among a list of alternatives.



A Static Text control reserves an area for the text in the dialog. Although called static, in fact the application can change the text during execution. This is often used for prompts or field names relating to other controls.

An Edit Control reserves an area for text editing (e.g. names, numbers, text constants etc.). Editing text strings with multiple lines is possible.

A List Box allows you to view a list of elements and to select one - or several - elements from this list.

A List Button control serves for choosing one from the pop-out set of the alternatives revealed by pressing the button. Pressing the button again pops the list back in again.

A List Edit control allows single line text editing like an Edit control, or selection from a list of elements.

The Vertical Scroll Bar and the Horizontal Scroll Bar controls serve to select a value within a scale of values.

A Group Box control serves for assembling together a number of controls in a functional group with a Group Name. Its only function is a visual one.

An Icon control reserves an area for an icon image.

A Custom Control can be either IDE Controls created in the project, it can be VPI user-defined control with a window class defined by vpi::classCreate, or it can be controls imported as COM packages from DLLs, VBXs or OCXs.

Inserting Controls

To insert a new control in a Dialog, youfirst select it and then click on the place in the Dialog where you want the control to be. There are three ways to select a control

- click on the icon in the Controls Toolbar
- select the Control in the Controls menu of the IDE
- select the Control in the popup menu that appears when you right click the Dialog in the IDE Designer. You see the popup menu in figure A1.3?

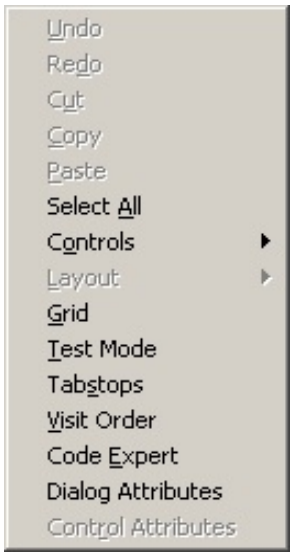


Figure A1.3 the Popup Menu

Controls Toolbar

When the IDE Designer is opened, the Controls toolbar is displayed next to the IDE Designer window. You see the Controls Toolbar in figure 7.?

Figure A1.4 The Controls Toolbar

By clicking icons in the Controls toolbar you select a type of a control to be placed into the edited Dialog. When you click a control type icon, the current cursor is replaced with the cursor correspondent to the control type to be inserted. Move the cursor to some position in the Design Area (the container client area) and click the left mouse button. The control with the default size will be inserted at the specified position.

Instead of dropping a control into a container with a click, it is possible to drag out the rectangle. When you have selected a Control, drop it in the Design Area but keep the mouse button pressed. As long as you keep the button pressed, you can change the size of the Control. When you release the

button, the Control is placed with the indicated size. In this way you can specify not only the control position but also the control size in one action.
After a GUI control is placed with the mouse into a GUI style container, it immediately appears at the specified location in the container. To change properties of the GUI control, you one should select it. Then the list of GUI control properties appears in the GUI control Properties table.

There are the following icons in the Controls Toolbar:

Icon	Insert a new:	Description
	Push Button	After selection of this control the cursor becomes 
	Check Box	After selection of this control the cursor becomes 
	Radio Button	After selection of this control the cursor becomes  .
	Static Text Control	After selection of this control the cursor becomes  .
	Edit Control	After selection of this control the cursor becomes  .
	List Box	After selection of this control the cursor becomes  .
	List Button	After selection of this control the cursor becomes  .
	List Edit	After selection of this control the cursor becomes  .
	Horizontal Scroll Bar	After selection of this control the cursor becomes  .
	Vertical Scroll Bar	After selection of this control the cursor becomes . 

Group Box After selection of this control the cursor becomes 

.

Icon Control After selection of this control the cursor becomes 
.

Custom Control After selection of this control the cursor becomes 
.

Selecting and Deselecting Controls in the Dialog Designer
Click inside a control to select it. A frame appears around the selected control. It is called the selection frame. To create an extended selection that contains more than one Control, there are two ways that will be familiar to you. The first way is to move the mouse pointer to the point in the dialog where you want one corner of the selected area to be and press and hold the left mouse button. While holding the left mouse button move the mouse until all the desired controls in the dialog are inside the selected area. Then release the mouse button. All controls inside the specified rectangle will be selected. The second way is when one or more controls are already selected. You can include one extra control in the selection by holding down the Ctrl key and click the control. In this way you can also deselect individual Controls.
To deselect the control or group of controls click outside the selection frame. Deselected controls lose their frame.

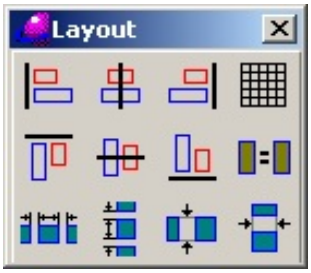
Resizing Controls
If you simply drop a control in the Dialog, it has a default size. To re-size a control, first click inside the control area to select it. Then move the cursor to the sizing handles on the selection frame. A new shape of cursor indicates the direction in which you can re-size the control. Press and hold the mouse button, drag until the selected control has the size you want, then release the mouse button. You can also change the size of a control by changing the values in the Control Properties List. Select the Control and then change the values for X and/or Y to change the position. Change the values for Width and Height to change the size. There is one exception, you cannot change the size of icons. You can only change their location.

Moving Controls
To move a control or a group of controls first select them. Then move the cursor inside the selection frame, press and hold the mouse button, drag the mouse pointer to a new location and release the mouse. Also, when one or more controls are selected, it is possible to use the cursor keys to move the selection in small steps (corresponding to the Grid settings).

Arranging the layout of the Controls
There are three ways to arrange the layout of Controls. You can use

- the Layout Toolbar
- the Layout menu-item in the IDE menu
- the popup menu that appears when you right click in the Dialog Designer.

To arrange the layout of a group of controls first select the group. Then click an appropriate button in the Layout toolbar or a menu item in the Layout menu of the IDE menu (or in the pop-up menu).



The Layout Toolbar

There are the following Layout commands for justifying and resizing controls:

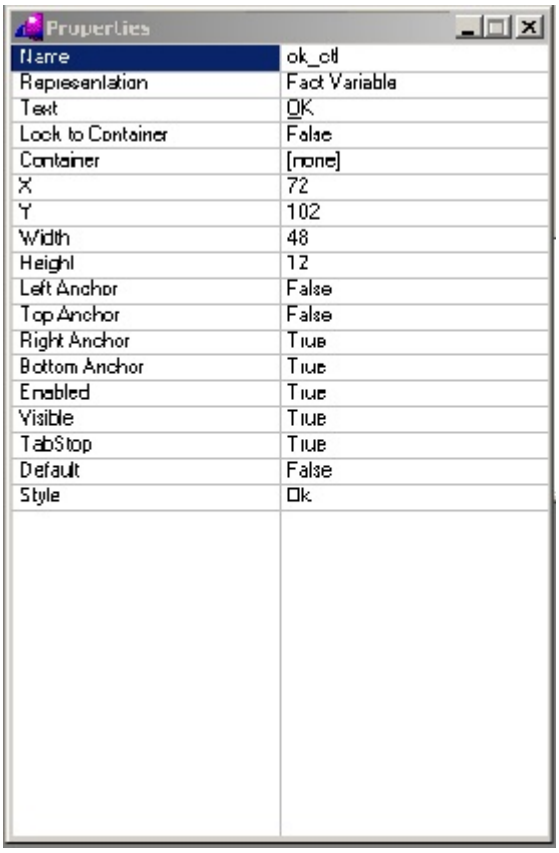
	Align Left	Align the selected controls along their left sides.
	Align Center	Center the selected controls vertically.
	Align Right	Align the selected controls along their right sides.
	Align Top	Align the selected controls along their tops.
	Align Middle	Center the selected controls horizontally.
	Align Bottom	Align the selected controls along their bottoms.
	Even Horizontal Spacing	Spacing the selected controls evenly between the leftmost and the rightmost controls.
	Grid	Toggle the Grid (see below).
	Even Vertical Spacing	Spacing the selected controls evenly between the topmost and the bottom-most controls.
	Make Same Size	Make the selected controls the same size as a model control.

	Make Same Horizontal Size	Make the selected controls the same width as a model control.
	Make Same Vertical Size	Make the selected controls the same height as a model control.
	Size To Contents	Resize the selected controls to optimally display its titles .

Control Properties Table

When you open a GUI dialog or a Form (GUI window) in the IDE Designer, then next to the opened dialog (form) the Control Properties table appears. If the Control Properties table is closed, then double-click a control - the Control Properties table appears. Also the Control Attributes command, from the IDE Designer speed menu (that pops up when you right click the Dialog), can be used to open the Control Properties table. You will find a comprehensive description of the Control properties in section A1.3.

To change properties of a GUI package control first select this control (by clicking in it) in the dialog (form). The current set of the control properties appears in the Control Properties table. Now you can edit the displayed GUI control properties.



Please noticew that in fact the properties table also contains the events that are processed by the control. You can switch between the properties and the events by clicking the tabs at the bottom.

Initially, when no Control is selected, the Control Properties table contains the properties (and events) of the form. As soon as one of the Controls in the edited Dialog or Form is selected, the Properties Table displays the properties of the selected control. The displayed set of properties depends on the control type. The Control Properties table contains two types of properties. The General group contains attributes, which are common to all types of controls: the Text determines the control title, the Constant (name) is used as the control identifier, and the Control Size group determines position and size of the control. Notice that the Text property/attribute is not used for some controls, for instance, it is not used for scroll bars.

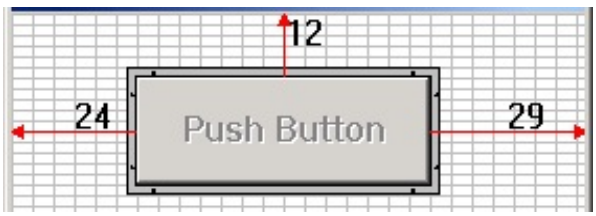
Notice also that in the Text fields the ampersand symbol & is reserved for indicating that the next to it character is to be underlined. For example, if you use the text string E&xit as a control name, then this control will be displayed with the title Exit at runtime. I.e. the character x will be displayed underlined (thus visually indicating that the x is an accelerator key). If you need to display the ampersand & symbol in the Text of a control, then use two of them &&.

Anchoring a Control

GUI controls have additional Anchor properties, which help to reposition (and may be resize) GUI controls when the containing GUI dialog (or form) is resized. Each GUI control has four anchor properties: Left Anchor, Top Anchor, Right Anchor, and Bottom Anchor. You find them in the properties Table. Each anchor determines that the specified (Left, Top, Right or Bottom) control boundary should be always positioned on the stated distance from the nearest (correspondent) border of the dialog (frame).

Each of anchors can have the value True or False. When the anchor value is True, then this anchor is active that is, when the dialog is resized, the anchor is used to position the control. For example, when the Left Anchor is active (has the value True), then the left control boundary should be positioned on the stated distance from the nearest (left) border of the dialog (frame).

The simplest way to see which anchors are active for a control, is to select the control. Then some red arrows between control boundaries and correspondent boundaries of the Dialog can bee seen. Each arrow specifies, that the correspondent anchor is active. For example, the arrow from the control top to the dialog top identifies, that the Top Anchor is active. In the picture below, you see three arrows, which specify that the Left, Top, and Right anchors are active to the Push Button control.



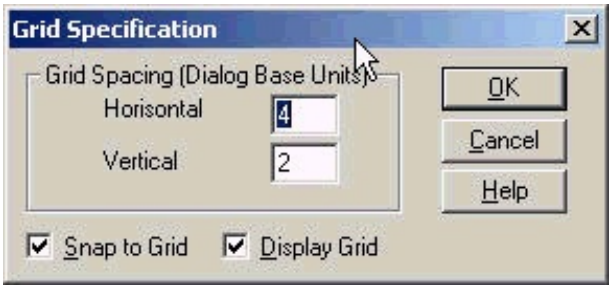
Numbers near such arrows show distances (in Dialog Base Units) between the correspondent boundaries of the control and the dialog. Normally each control has one active horizontal and one active vertical anchor. When the dialog is resized, then such control is always positioned on the specified distances from the specified boundaries of the dialog. The size of the control is not changed! When a control has both horizontal (or/and both vertical) anchors active, then both horizontal (or/and both vertical) boundaries of the control should be placed on some specified distances from the

correspondent horizontal (vertical) boundaries of the dialog. Therefore, when the dialog is resized, then such control is also resized accordingly to keep the specified distances.
When no one of the horizontal or vertical anchors of a control are active, then no one of the horizontal or vertical boundaries of the control are bounded to the correspondent horizontal or vertical boundaries of the dialog. Coordinates of such controls are handled on the "proportional basis". For example, when the dialog width is increased onto Delta dialog base units, then the X coordinate of the control is also increased, but it is increased two times smaller. That is the X coordinate of the control is increased only onto Delta/2 dialog base units. So the Size of the control changesw proportional to the change in size of the Form.

The other Control properties are specific for different kinds of controls. They are described in the next section.

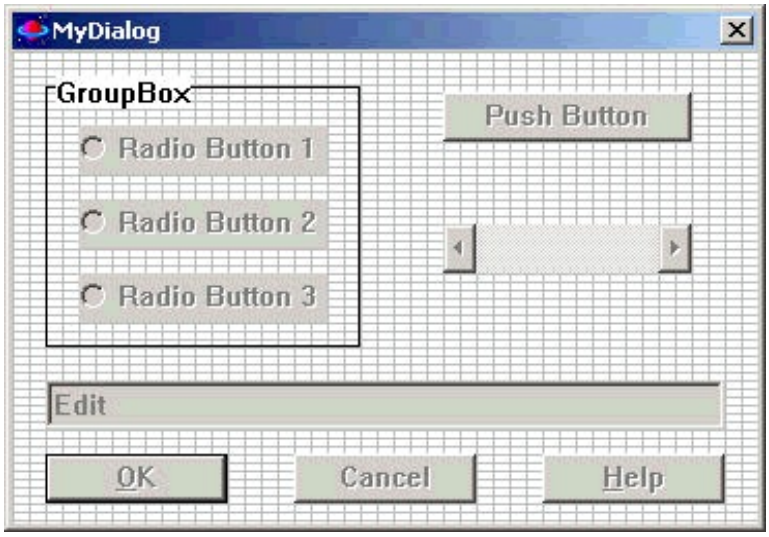
Cut, Copy and Paste, Undo and Redo
You can Cut or Copy a group of selected controls onto the Windows clipboard, and Paste controls from the clipboard back into a dialog. The Undo and Redo commands serve to delete or restore the last editing operations of your dialog.

Grid
When the menu (or the pop-up menu) entry Resource | Grid is activated, or when the  button from the Layout toolbar is pressed, the following dialog appears:



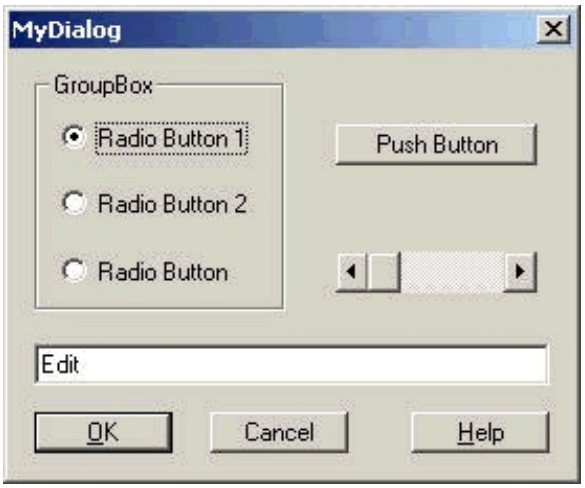
The Dialog to Specify the Grid Properties

With a grid in place, arranging the controls inside a dialog is easier. Also it is possible to tell the IDE Designer by the Snap to Grid that it should place controls at the grid intersections to give a satisfactory result.



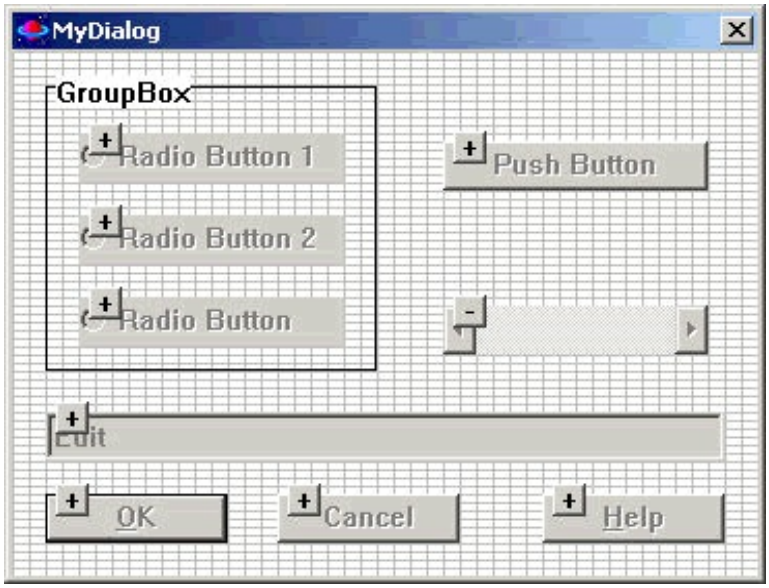
Using a grid in the Dialog/Window/Form Editor

Test Mode
The IDE Designer has the Test Mode, so it is possible to see how the dialog with the created controls looks like and behaves. To illustrate the Test Mode we will assign some default values to the controls. The test mode is activated and deactivated from the Resource menu (or from the pop-up menu).



The Dialog in Test Mode

Tab Stops
When activating the menu (or the pop-up menu) entry Resource | Tabstops, it is possible to specify to which controls you can Tab to in the dialog. When this command is activated a small button with the + or - appears on controls depending upon whether the control has a tab stop or not. By clicking the small buttons, it is possible to toggle the setting.

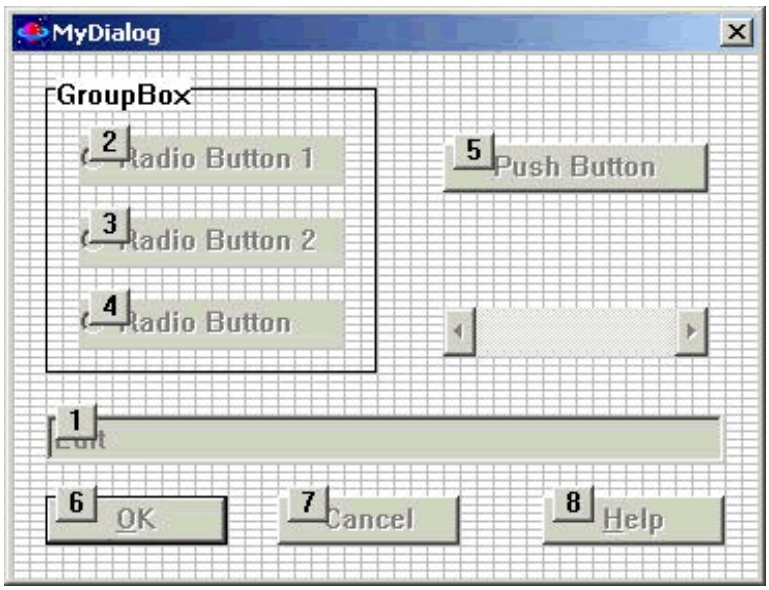


Specifying Tab Stops for a Dialog

To exit the tab stop mode, just click in the dialog but outside of any controls.

Visit Order

When the tab stops have been specified for a dialog, then it is possible to specify the order in which controls will receive focus when tabbing.



Displaying the Visit Order.

To change the visit order, click a small rectangle displaying the visit order number for a control that has an incorrect sequence number and this will bring up another dialog to change this sequence:

Changing the Visit Order.

To stop the visit order mode, just click in the dialog outside any controls.

A1.3 The Control Properties Table

When you open a GUI dialog or a form (GUI window) in the IDE Designer, then the Control Properties Table also opens. The set of properties displayed in the Control Properties Table depends on the control type and contains two groups of properties, General properties and specific properties. General properties are common to all types of controls. These are: Left Anchor, Top Anchor, Right Anchor, Bottom Anchor, Container Name, Enabled, Lock to Container, Name, Representation, TabStop, Title, X, Y, Width, Height, and Visible. Specific properties are individual to different types of GUI controls. These are: 3State, AlignBaseline, Alignment, AllowPartialRows, Auto, AutoNotify, AutoHScroll, AutoVScroll, Border, Case, Class, Default, ExtendedSel, HideSel, HScroll, Icon Name, IgnorePrefix, LeftText, MultiColumn, MultiLine, MultiSelect, Password, ReadOnly, Rows, Sort, StaticScrollbar, Style, UseTabStops, VScroll, WantReturn, Wrap.

In the Control Properties table you can edit the displayed properties of the selected GUI control. Initially this table is empty. However, as soon as any of GUI controls in the edited GUI dialog (form) is selected, the Control Properties table displays the current set of properties of the selected control. If the Control Properties table is closed, double-click a control to show the table. The **Control Attributes** command, from the speed menu (right click to make it popup) of the **IDE Designer**, can also be used to open the Control Properties table.

A1.3.1 Common Properties of Almost All GUI Controls

There are some basic control properties, which are used by almost all GUI controls. These are:

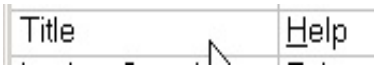
Name

The **Name** input box is where you can specify some correct Visual Prolog name, which will be used as the control name (identifier). The default Name is generated from the control type (for example, pushButton, checkBox, listButton, etc.) and the _ctl suffix. If several controls of the same type are created, then the control type is suffixed with the number, for example, pushButton1_ctl. You can edit this default name or simply type in your own version.

Representation

This property can have one of the following two values: Fact Variable, Variable.
Fact Variable
When Fact Variable is specified, then the IDE generates the fact variable, which will store the object of the control. This fact variable is used to reference to this control in the code. This fact variable name is determined by the Name property. For example, ok_ctl. Notice that in this case the fact variable name is automatically converted to be started with the lowercase character.
Variable
When Variable is specified, then the IDE unifies the object of the control with an ordinary variable. This variable will be used to reference to this control in the code. This variable name is determined by the Name property. For example, Cancel_ctl. Notice that in this case the variable name is automatically converted to be started with the uppercase character.
The default value is Fact Variable.

Title
The Title property is where you specify the title (label), which you want to be displayed as the control caption. One character in the Title text may be underlined to indicate an access key. To underline a character precede it with the ampersand & character. At runtime this character in the control Title will be underlined. Notice that this character is displayed underlined also in the value cell of the Title property:

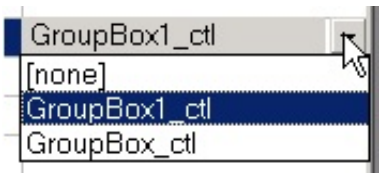


The used ampersand & character is seen only when the value cell of the Title property has the focus:

If you need to display the ampersand & character in the Title text, then use two of them &&.

Lock to Container
Possible values are True and False. True defines that the control is locked inside the container selected in the Container Name property.

Container Name
In this list box you can select the Name of one of containers (currently only the group box groupBox type of containers is supported) defined in the dialog or in the form. All controls belonging to the same container are in the same group of related controls. The user may use the arrow keys to move from one control in the *group* to the next one. Only the Names of group boxes, inside which the control is placed, appear in the list of Container Name property values. For example, if you see:



this means that the control is placed inside two group boxes with names GroupBox_ctl and GroupBox1_ctl. If you do not need to relate the control to any container, then you should select [none].

X, Y, Width, Height
These properties determine the control coordinates and size:
X
The **X** coordinate of the upper-left corner of the control (in Dialog Base Units).
Y
The **Y** coordinate of the upper-left corner of the control (in Dialog Base Units).
Width
The **Width** of the control (in Dialog Base Units).
Height
The **Height** of the control (in Dialog Base Units).

Left, Top, Right, and Bottom Anchors
These properties determine how the control coordinates (and may be size) will be handled when the dialog or form (containing this control) is resized. The general rules are described in the " Using Anchors for Positioning GUI Controls while Dialog Resizing " in section A2.
Left Anchor
Possible values are True and False. True defines that the Left boundary of the control is always positioned at the specified distance (equal to the X property) from the Left boundary of the dialog (form) containing this control. The distance is specified in the Dialog Base Units.
Top Anchor
Possible values are True and False. True defines that the Top of the control is always positioned at the specified distance (equal to the Y property) from the Top boundary of the dialog (form) containing this control. The distance is specified in the Dialog Base Units.
Right Anchor
Possible values are True and False. True defines that the Right boundary of the control is always positioned at the specified distance from the Right boundary of the dialog (form) containing this control. The distance is specified in the Dialog Base Units. The distance value can be seen near the arrow, which appears when the control is selected
Bottom Anchor
Possible values are True and False. True defines that the Bottom of the control is always positioned at the specified distance from the Bottom boundary of the dialog (form) containing this control.
The distance is specified in the Dialog Base Units. The distance value can be seen near the arrow, which appears when the control is selected.

Enabled
Possible values are True and False. They define whether the control will be created initially enabled (True) or disabled (False). The default value is True (enabled).
You can use the window::getEnabled/0-> predicate to get whether the control is enabled

```
IsEnabled  = checkButtonName:getEnabled(),
IsEnabled  = true,
```

and the window::setEnabled/1 predicate

```
IsEnabled  = true,
checkButtonName:setEnabled( IsEnabled ),
```

to set this flag programmatically.

Visible

Possible values are True and False. They define whether the control will be created initially visible (True) or invisible (False). The default value is True (visible).
You can use the window::getVisible/0-> predicate to get whether the control is visible

```
IsVisible  = checkButtonName:getVisible(),
IsVisible  = true,
```

and the window::setVisible/1 predicate

```
IsVisible  = true,
checkButtonName:setVisible( IsVisible ),
```

to set this flag programmatically.

TabStop
Possible values are True and False. They define whether this control belongs to a subset of controls through which the user can move using the TAB key. The default value is True (the control belongs to a group of controls between which the user can navigate with TAB).
You can use the control::getTabStop/0-> predicate to get whether the control has this flag

```
IsTabStop  = checkButtonName:getTabStop(),
IsTabStop  = true,
```

and the control::setTabStop/1 predicate

```
IsTabStop  = true,
checkButtonName:setTabStop( IsTabStop ),
```

to set this flag programmatically.
If IsTabStop is true, then the focus will be able to jump to this control while using the Tab for navigation. Pressing the TAB key moves the focus to the next control, which has the TabStop = True.

A1.3.2 Specific Properties of Different GUI Control Types
The following properties are specific to different types of controls:

3State
Possible values are True and False. They define whether this check box can have 3 states (unchecked, checked, undetermined). You can use the checkButton::getStyle/2 predicate to get whether this state is set to a control

```
checkButtonName:getStyle (_ IsAuto ,  Is3State ),
Is3State  = true,
```

and the checkButton::setStyle/2 predicate

```
Is3State  = true,
checkButtonName:setStyle ( _IsAuto ,  Is3State ),
```

to set this state programmatically. A 3-states check box is the same as an ordinary check box, except that the box can be grayed (dimmed) as well as checked. The grayed state is used to show that the state of the check box is not determined. The default value is False (3State).

AlignBaseline
Possible values are True and False. They define whether the control caption text should be positioned on the same horizontal line with caption texts of all other controls, which have the same vertical position with this control. This property is defined to all controls, which can have captions (static texts, check boxes, edit and list edit controls, etc.).

You can use the getAlignBaseline predicates (like editControl::getAlignBaseline/0->) to get whether the control has this flag

```
IsAllignBaseLine  = checkButtonName:getAllignBaseLine(),
IsAllignBaseLine  = true,
```

and the setAlignBaseline (like textControl::setAlignBaseline/1) predicates

```
IsAllignBaseLine  = true,
checkButtonName:setAllignBaseLine( IsAllignBaseLine ),
```

to set this flag programmatically. The default value is True (align captions horizontally).

Alignment

Possible values are Left, Center, and Right. Radio buttons in this group box determine how the text in the control is aligned. The default value is Left (align left).

Left

Defines that the text in the control will be aligned left.

You can use the textControl::getAlignment/0-> predicate to get which of these flags the control has

```
AlignmentType  = textControlName : getAlignment (),
AlignmentType  = alignLeft(),
```

and the textControl::setAlignment/1 predicate

```
AlignmentType  = alignLeft() ,
textControlName : setAlignment ( AlignmentType ),
```

to set this flag programmatically.

Auto

Possible values are True and False. They define whether this control is an automatic (an automatic check box or an automatic radio button). When you click an automatic control, then it change its state in the proper way.

You can use the checkButton::getStyle/2 and radioButton::getAuto/0-> predicates to get whether this flag is set to a control:

```
checkButtonName:getStyle ( IsAuto ,  _Is3State ),
IsAuto  = true,
```

and the checkButton::setStyle/2 or radioButton::setAuto/1 predicates

```
IsAuto  = true,
checkButtonName:setStyle ( IsAuto ,  _Is3State ),
```

to set this flag programmatically. The default value is True (auto).

Center

Defines that the text in the control will be centered. The AlignmentType = alignCenter().

Right

Defines that the text in the control will be aligned right. The AlignmentType = alignRight().

AllowPartialRows

Possible values are True and False. If it is True, then the size of the list box is exactly the size specified by the application when it creates the list box. Normally, Windows re-sizes list boxes so that a list box does not display partial items. The default value is True for list box controls and False for list button and list edit controls.

You can use the listControl::getAllowPartialRows/0-> predicate to get whether the control has this flag

```
IsAllowPartialRows  = controlName : getAllowPartialRows (),
```

```
        IsAllowPartialRows  = true,
```
and the listControl::setAllowPartialRows/1 predicate
```
        IsAllowPartialRows  = true,
        controlName : setAllowPartialRows ( IsAllowPartialRows ),
```
to set this flag programmatically.

AutoHScroll
Possible values are True and False. They define whether the control automatically scrolls text in the control in order to display a typed character, when the user types the character at the end of the line. If it is False, then only text that fits within the rectangular boundary is allowed. When the user presses the Enter key, the control scrolls all text back to the zero position. The default value is True (scrolls text automatically).
You can use the editControl::getAutoHScroll/0-> predicate to get whether the control has this flag
```
        IsAutoHScrollSet  = controlName : getAutoHScroll (),
        IsAutoHScrollSet  = true,
```
and the editControl::setAutoHScroll/1 predicate
```
        IsAutoHScrollSet  = true,
        controlName : setAutoHScroll ( IsAutoHScrollSet ),
```
to set this flag programmatically.

AutoNotify
Possible values are True and False. If it is True, then the list box control notifies the parent window with an input message whenever the user clicks or double-clicks in the list box. The default value is True (notify).
You can use the listBox::getAutoNotify/0-> predicate to get whether the control has this flag
```
        IsAutoNotify  = controlName : getAutoNotify (),
        IsAutoNotify  = true,
```
and the listBox::setAutoNotify/1 predicate
```
        IsAutoNotify  = true,
        controlName : setAutoNotify ( IsAutoNotify ),
```
to set this flag programmatically.

AutoVScroll
Possible values are True and False. They define whether the multi-line edit control automatically scrolls text up when the user presses the Enter on the last line. The default value is False (does not scroll). This property has no effect on single-line edit controls.
You can use the editControl::getAutoVScroll/0-> predicate to get whether the control has this flag
```
        IsAutoVScrollSet  = controlName : getAutoVScroll (),
        IsAutoVScrollSet  = true,
```
and the editControl::setAutoVScroll/1 predicate
```
        IsAutoVScrollSet  = true,
        controlName : setAutoVScroll ( IsAutoVScrollSet ),
```

to set this flag programmatically.

Border

Possible values are True and False. They define whether the control is created initially with a border around the control. The default value is False.
You can use the editControl::getBorder/0-> or listBox::getBorder/0-> predicates to get whether the control has this flag

```
BorderIsSet  = controlName : getBorder (),
BorderIsSet  = true,
```

and the editControl::setBorder/1 or listBox::setBorder/1 predicates

```
BorderIsSet  = true,
controlName : setBorder ( BorderIsSet ),
```

to set this flag programmatically.

Case
Possible values are Insensitive, Upper, Lower. The default value is Insensitive.
Insensitive
Defines that the edit control leaves the typed-in text "as-is". This is the default value.
Upper
Defines that the edit control converts all typed-in characters into the upper case.
You can use the editControl::getUpperCase/0-> predicate to get whether the control has this flag

```
IsUpperCase  = controlName : getUpperCase (),
IsUpperCase  = true,
```

and the editControl::setUpperCase/1 predicate

```
IsUpperCase  = true ,
controlName : setUpperCase ( IsUpperCase ),
```

to set this flag programmatically.
Lower
Defines that the edit control converts all typed-in characters into the lower case.
You can use the editControl::getLowerCase/0-> predicate to get whether the control has this flag

```
IsLowerCase  = controlName : getLowerCase (),
IsLowerCase  = true,
```

and the editControl::setLowerCase/1 predicate

```
IsLowerCase  = true ,
controlName : setLowerCase ( IsLowerCase ),
```

to set this flag programmatically.

Class
Specify a name of a Windows class for the custom control (your application code must register the custom control by this name).

Default
Possible values are True and False. They define whether the push button control is the default push button. When <Enter> is pressed by the user, the action of the default button is activated.
You can use the dialog::tryGetDefaultButton/0-> predicate to get the object of the default button in the dialog.
Similarly you can use the dialog::setDefaultButton/1 predicate to set the default button in the dialog.

Only one button in each dialog can be the default button, but any number of push buttons can have the True value of this property. IDE makes the default button that of dialog buttons, with the True value of this property, which value was set the latest one.
By default this property is True (is the default button) only for the OK button, for all other buttons the default value is False.

ExtendedSel
Possible values are True and False. If it is True, then the list box control allows selection of multiple items by using the Shift key and the mouse or using the Shift key and the Up or Down Arrow keys. The default value is False (cannot select multiple items).
You can use the listBox::getExtendedSel/0-> predicate to get whether the control has this flag

```
IsExtendedSel  = controlName : getExtendedSel (),
IsExtendedSel  = true,
```

and the listBox::setExtendedSel/1 predicate

```
IsExtendedSel  = true,
controlName : setExtendedSel ( IsExtendedSel ),
```

to set this flag programmatically.

HideSel
Possible values are True and False. If it is True, then the edit control hides selection, when it loses the input focus, and shows selection, when it receives the input focus. The default value is True (hides selection).
You can use the editControl::getHideSel/0-> predicate to get whether the control has this flag

```
IsHideSel  = controlName : getHideSel (),
IsHideSel  = true,
```

and the editControl::setHideSel/1 predicate

```
IsHideSel  = true,
controlName : setHideSel ( IsHideSel ),
```

to set this flag programmatically.

HScroll
Possible values are True and False. They define whether the multi-line edit or list box control gets a horizontal scroll bar when it needs one. The default value is False (does not scroll). This property has no effect on single-line edit controls.
You can use the editControl::getHScroll/0-> or listBox::getHScroll/0-> predicates to get whether the control has this flag

```
isHScrollSet  = controlName : getHScroll (),
IsHScrollSet  = true,
```

and the editControl::setHScroll/1 or listBox::setHScroll/1 predicates

```
IsHScrollSet  = true,
controlName : setHScroll ( IsHScrollSet ),
```

to set this flag programmatically.

Icon Name
The icon name must be selected from the list of icon names registered in the Project (by clicking the  button). In order to an icon name appears in the list of registered icon names, you should add the

icon file (IconName.ICO) into the project files. To do this you should use the File | Add command (the Resource Files should be selected in the Files of Type).

IgnorePrefix
Possible values are True and False. Setting True value prevents interpretation of any & characters in the caption of the static text control as access keys (accelerator prefix characters). Accelerator characters are displayed underlined and the prefix & character is deleted. The default value is False. Notice that static text controls cannot be activated. Therefore, when an access key of a static text control is pressed, then the control next to this static text control, in the tab-stop order of controls in the dialog, is activated. Most often this flag is used when filenames, which can contain the & character, need to be displayed in static text controls.
You can use the textControl::getIgnorePrefix/0-> predicate to get whether the control has this flag

```
IsIgnorePrefix  = controlName : getIgnorePrefix (),
IsIgnorePrefix  = true,
```

and the textControl::setIgnorePrefix/1 predicate

```
IsIgnorePrefix  = true,
controlName : setIgnorePrefix ( IsIgnorePrefix ),
```

to set this flag programmatically.

LeftText
Possible values are True and False. They define whether the control title text should be positioned on the left or on the right of the control (check box or radio button).
You can use the checkButton::getLeftText/0 -> or radioButton::getLeftText/0-> predicates to get whether the control has this flag

```
IsLeftText  = checkButtonName:getLeftText(),
IsLeftText  = true,
```

and the checkButton:setLeftText/1 or radioButton:setLeftText/1 predicates

```
IsLeftText  = true,
checkButtonName:setLeftText( IsLeftText ),
```

to set this flag programmatically. The default value is False (caption is on the right side).

MultiColumn
Possible values are True and False. If it is True, then the list box control can have multiple columns. The default value is False.
You can use the listBox::getMultiColumn/0-> predicate to get whether the control has this flag

```
IsMultiColumn  = controlName : getMultiColumn (),
IsMultiColumn  = true,
```

and the listBox::setMultiColumn/1 predicate

```
IsMultiColumn  = true,
controlName : setMultiColumn ( IsMultiColumn ),
```

to set this flag programmatically.

MultiLine
Possible values are True and False. When this property is True, then the edit control allows multiple lines. The default value is True (allows multiple lines). Notice that the VScroll, HScroll, AutoVScroll, and WantReturn properties do not work for single-line (False) edit controls.
You can use the editControl::getMultiLine/0-> predicate to get whether the control has this flag

```
        IsMultiLine  = controlName : getMultiLine (),
        IsMultiLine  = true,
```
and the editControl::setMultiLine/1 predicate
```
        IsMultiLine  = true,
        controlName : setMultiLine ( IsMultiLine ),
```
to set this flag programmatically.

MultiSelect
Possible values are True and False. If it is True, then the list box control allows selection of multiple items using the mouse. The default value is False (cannot select multiple items).
You can use the listBox::getMultiSelect/0-> predicate to get whether the control has this flag
```
        IsMultiSelect  = controlName : getMultiSelect (),
        IsMultiSelect  = true,
```
and the listBox::setMultiSelect/1 predicate
```
        IsMultiSelect  = true,
        controlName : setMultiSelect ( IsMultiSelect ),
```
to set this flag programmatically.

Password
Possible values are True and False. If it is True, then the edit control displays all typed-in characters as asterisks (*). The default value is False (displays typed characters "as-is").
You can use the editControl::getPassword/0-> predicate to get whether the control has this flag
```
        IsPassword  = controlName : getPassword (),
        IsPassword  = true,
```
and the editControl::setPassword/1 predicate

```
        isPassword  = true,
        controlName : setPassword ( IsPassword ),
```
to set this flag programmatically.

ReadOnly
Possible values are True and False. If it is True, then the edit control is in the read-only mode. The default value is False (editable).
You can use the editControl::getReadOnly/0-> predicate to get whether the control has this flag
```
        IsReadOnly  = controlName : getReadOnly (),
        IsReadOnly  = true,
```
and the editControl::setReadOnly/1 predicate
```
        IsReadOnly  = true,
        controlName : setReadOnly ( IsReadOnly ),
```
to set this flag programmatically.

Rows
Possible values are positive numbers. For list button and list edit controls it specifies the maximal number of rows, which can be displayed in the opened list box sub-control. The default value is 3.
You can use the listControl::getMaxDropDownRows/0-> predicate to get this number
```
        MaxDropDownRowNumber  = controlName : getMaxDropDownRows (),
```
and the listControl::setMaxDropDownRows/1 predicate

MaxDropDownRowNumber = 5,
controlName : setMaxDropDownRows (MaxDropDownRowNumber),
to set this number programmatically.

Sort
Possible values are True and False. If it is True, then the list control sorts strings alphabetically. The default value is True (sort).
You can use the listControl::getSort/0-> predicate to get whether the control has this flag
IsSort = controlName : getSort (),
IsSort = true,
and the listControl::setSort/1 predicate
IsSort = true,
controlName : setSort (IsSort),
to set this flag programmatically.

StaticScrollbar
Possible values are True and False. If it is True, then the list control shows a disabled vertical scroll bar for the list box when the box does not contain enough items to scroll. If it is False, then the scroll bar is hidden when the list box does not contain enough items. The default value is True (shows vertical scroll bar).
You can use the listControl::getStaticScrollbar/0-> predicate to get whether the control has this flag
IsStaticScrollbar = controlName : getStaticScrollbar (),
IsStaticScrollbar = true,
and the listControl::setStaticScrollbar/1 predicate
IsStaticScrollbar = true,
controlName : setStaticScrollbar (IsStaticScrollbar),
to set this flag programmatically.

Style
Push Buttons and group boxes can have several styles. The style determines some of the activities of the control. For Push Buttons the style can have the values: Cancel, OK, and Regular.
Cancel
Push buttons with the Cancel style are created by the button::newCancel/1 constructor.
When a dialog/form contains a push button with the Cancel style, then this dialog/form will provide standard handling of the Esc key - the dialog/form is simply closed without any additional actions.
OK
Push buttons with the OK style are created by the button::newOk/1 constructor.
When a dialog/form contains a push button with the Ok style, then this dialog/form will execute dialog/form validation check when this button is clicked. See " Dialog/Form Validation" tutorial (on the WEB) for more information.
Regular
Push buttons with the Regular style are created by the ordinary button::new/1 constructor. Such buttons do not initiate any special default handling.

For group boxes possible values for Style are Group Box, Simple Border, Horizontal Separator, No Border . These values define the border type of the group box control. The default is Simple Border.
You can use the groupBox::getBorderStyle/0-> predicate to get the border style

```
        GroupBoxBorderStyle  = controlName : getBorderStyle (),
```
and the groupBox::setBorderStyle/1 predicate
```
        GroupBoxBorderStyle  = groupBox(),
        controlName : setBorderStyle ( GroupBoxBorderStyle ),
```

to set the border style programmatically.

UseTabStops
Possible values are True and False. If it is True, then the list box control recognizes and expands TAB characters when drawing strings. The default value is False (does not expand).
You can use the control::getUseTabStops/0-> predicate to get whether the control has this flag
```
        IsUseTabStops  = controlName : getUseTabStops (),
        IsUseTabStops  = true,
```
and the control::setUseTabStops/1 predicate
```
        IsUseTabStops  = true,
        controlName : setUseTabStops ( IsUseTabStops ),
```
to set this flag programmatically.

VScroll
Possible values are True and False. They define whether the multi-line control gets a vertical scroll bar when it needs one. The default value for edit controls is True, for other types of controls it is False. This property has no effect on single-line edit controls.
You can use the editControl::getVScroll/0-> and listControl::getVScroll/0-> predicates to get whether the control has this flag
```
        IsVScrollSet  = controlName : getVScroll (),
        IsVScrollSet  = true,
```
and the editControl::setVScroll/1 and listControl::setVScroll/0 predicates
```
        IsVScrollSet  = true,
        controlName : setVScroll ( IsVScrollSet ),
```
to set this flag programmatically.

WantReturn
Possible values are True and False. If it is True, then a carriage return will be inserted as the part of an edited string, when the user presses the Enter key while entering text into a multi-line edit control. If it is False, then pressing the Enter key has the same effect as pressing the default (OK) push button, which has the latest specified Default = True property value. This property has no effect on single-line edit controls. The default value is False.
You can use the editControl::getWantReturn/0-> predicate to get whether the control has this flag
```
        IsWantReturn  = controlName : getWantReturn (),
        IsWantReturn  = true,
```
and the editControl::setWantReturn/1 predicate
```
        IsWantReturn  = true,
        controlName : setWantReturn ( IsWantReturn ),
```
to set this flag programmatically.

Wrap

Possible values are True and False. They define whether the text in the control will be wrapped to several lines. The default value is True (wrap).
You can use the textControl::getWrap/0-> predicate to get whether the control has this flag

```
        IsNoWrap  = textControlName : getWrap (),
        IsNoWrap  = true,
```

and the textControl::setWrap/1 predicate

```
        IsNoWrap  = false,
        textControlName  : setWrap ( IsNoWrap ),
```

to set this flag programmatically.

Appendix A2 List manipulating predicates

In the class “list” VIP offers a lot of predicates for manipulating lists. All these predicates act as functions. It means that when you call them, you must bind the result to a variable. E.g. the predicate append/2 is used to append a list to another list. The declaration of this predicate is

```
append : (Elem* Front, Elem* Tail)  -> Elem* List.
```

From this declaration you can see that it is a polymorphic predicate. That means that it takes any input, as long as it is a list. So you can use this predicate for integer lists, string lists, et cetera. The predicate mode is omitted, so it is procedure. The flow is omitted, so all the arguments are input. To call this predicate, use:

```
...
List = list::append(Front, Tail),
...
```

The variable List will contain the result, that is a list in which Tail is appended to Front. When you open the class list in your program, you can simply state

```
...
List = append(Front, Tail),
...
```

What follows here is a short description of the most important predicates in the class “list”. In the helpfile you can find the other available predicates.

---

list::append/2->

append : (Elem* Front, Elem* Tail) -> Elem* List
List is the result of appending Tail to Front.

---

list::appendList/1->

appendList : (Elem** ListList) -> Elem* List
AppendList/1 takes for an input a list of lists. The result List is the result of appending all the lists in ListList.

---

list::getMember_nd/1->

getMember_nd : (Elem* List) -> Elem Value nondeterm.
Returns the members Value of the list List.

---

list::isMember/2

isMember : (Elem Value, Elem* List) determ.
Succeeds if the element Value is member of List.

---

list::length/1->

length : (Elem* List) -> core::positive Length.
Returns the length of the List as a natural number>=0. (What else could you expect? :-))

---

list::maximum/1->

maximum : (Elem* List) -> Elem Item.
Returns the element with maximal value in List.

---

list::minimum/1->

minimum : (Elem* List)
   -> Elem Item.
Return the minimum element in List.

---

list::nDuplicates/2->

nDuplicates : (Elem Element, core::positive N) -> Elem* List.
Takes for an input a single Element and a positive number N and then creates a list that contains N duplicates of Element

---

list::nth/2->

nth : (core::positive Index, Elem* List) -> Elem Item procedure.
Returns the nth element in List.

---

list::remove/2->

remove : (Elem* List, Elem Value) -> Elem* ListWithoutValue
Removes the first occurence of Value from List.

---

list::removeAll/2->

removeAll : (Elem* List, Elem Value) -> Elem* ListWithNoValues
Removes the all occurences of Value from List.

---

list::removeDuplicates/1->

removeDuplicates : (Elem* List) -> Elem* ListWithNoDuplicates
Removes the all duplicates from List. Last occurences are retained, this causes the order in the resulting list.

---

list::reverse/1->

reverse : (Elem* List) -> Elem* Reverse
Reverse contains the elements of List in reversed order.

---

list::setNth/4

setNth : (core::positive Index, Elem* List, Elem Item, Elem* NewList) procedure (i,i,i,o).
Replaces the nth element in List with Item. Returns the NewList with the n'th element changed to Item.

---

list::sort/1->

sort : (Elem* List) -> Elem* SortedList
Sorts the List.

---

list::sort/2->

sort : (Elem* List, sortOrder Order) -> Elem* SortedList.
SortedList is the sorted version of List. Order specifies the sort order, it can be “ascending()” and “descending()”. Default is ascending

---

list::sortBy/2->

sortBy : (comparator{Elem} Comparator, Elem* List) -> Elem* SortedList.
SortedList is the sorted version of List. Comparator is an user defined comparator predicate.

---

list::sortBy/3->

sortBy : (comparator{Elem} Comparator, Elem* List, sortOrder Order) -> Elem* SortedList
SortedList is the sorted version of List. Comparator is an user defined comparator predicate Order specifies the sort order, it can be “ascending()” and “descending()”. Default is ascending

---

list::tryGetIndex/2->

tryGetIndex : (Elem Item, Elem* List) -> core::positive Index determ.
Gets the index number of Item in the List. This predicates fails if Item is not found.

Index