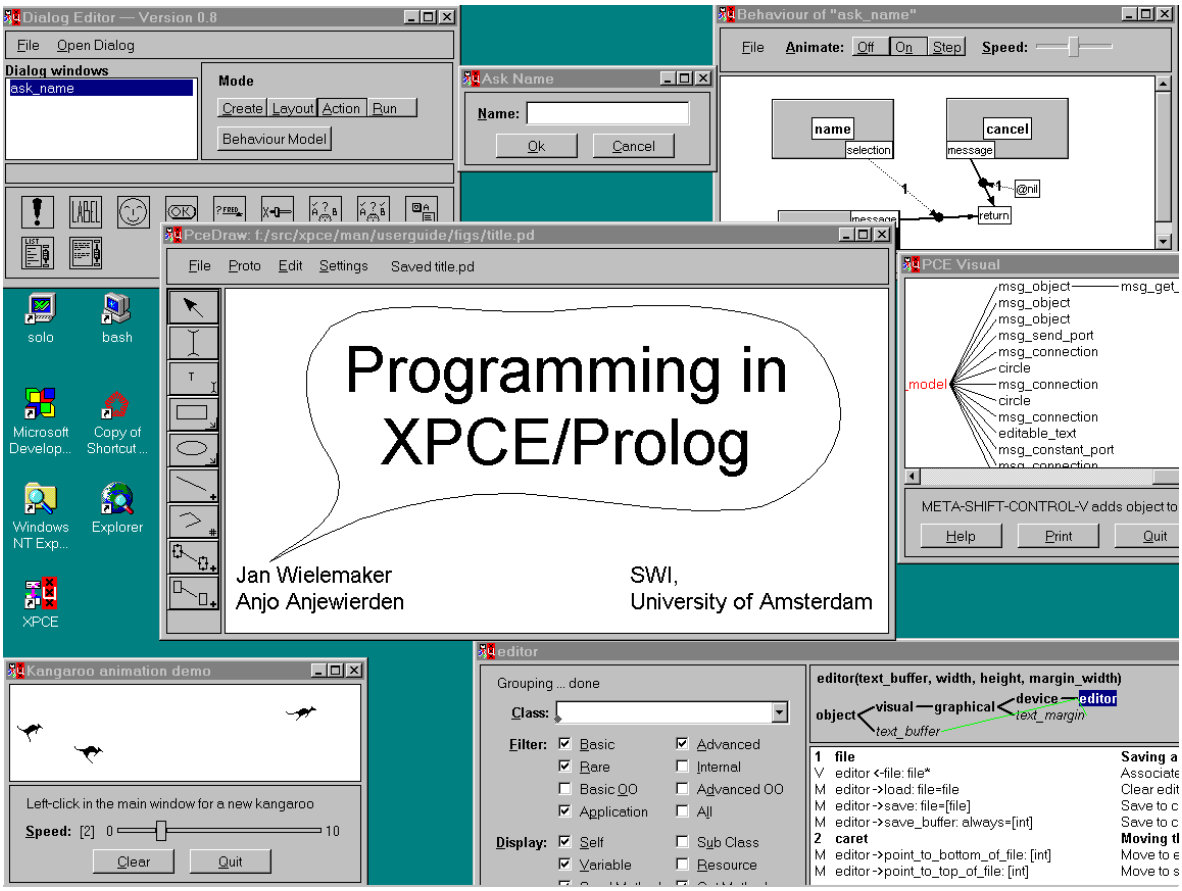

Dialog Editor — Version 0.8
File Open Dialog
Dialog windows
ask_name
Mode
Create Layout Action Run
Behaviour Model
Ask Name
Name:
Ok Cancel
Behaviour of "ask_name"
File Animate: Off On Step Speed:
name
selection
cancel
message
@nil
return
PceDraw: f:/src/xpce/man/userguide/figs/title.pd
File Proto Edit Settings Saved title.pd
Programming in XPCE/Prolog
Jan Wielemaker
Anjo Anjewierden
SWI,
University of Amsterdam
PCE Visual
model
msg_object
msg_get_
msg_send_port
msg_connection
circle
editable_text
msg_constant_port
META-SHIFT-CONTROL-V adds object to
Help Print Quit
solo
bash
Microsoft Develop...
Copy of Shortcut ...
Windows NT Exp...
Explorer
XPCE
Kangaroo animation demo
Left-click in the main window for a new kangaroo
Speed: [2] 0 10
Clear Quit
editor
Grouping ... done
Class:
Filter: Basic Advanced Rare Internal Basic OO Advanced OO Application All
Display: Self Sub Class Variable Resource
editor(text_buffer, width, height, margin_width)
object visual graphical device editor
text_buffer
text_margin
1 file Saving a
V editor <-file: file*
M editor ->load: file=file
M editor ->save: file=[file]
M editor ->save_buffer: always=[int]
2 caret Moving t
M editor ->point_to_bottom_of_file: [int]
M editor ->point_to_top_of_file: [int]

UNIVERSITEIT VAN AMSTERDAM

Dept. of Social Science Informatics (SWI)
Roetersstraat 15, 1018 WB Amsterdam
The Netherlands
http://www.swi.psy.uva.nl

Programming in XPCE/Prolog

Jan Wielemaker jan@swi.psy.uva.nl
Anjo Anjewierden anjo@swi.psy.uva.nl

XPCE/Prolog is a hybrid environment integrating logic programming and object-oriented programming for Graphical User Interfaces. Applications in XPCE/Prolog are fully compatible across the supported X11 and Win32 (Windows 95/98/ME, NT/2000/XP) platforms.

This document also applies to XPCE/Prolog 6.0.0 distributed as integrated packages to SWI-Prolog. Sources and binaries may be downloaded from http://www.swi-prolog.org

XPCE is distributed as Free Software with sufficient escapes to allow for producing non-free applications. The kernel is distributed under the *Lesser GNU Public License* (LGPL) and the Prolog sources under the *GNU Public License* (GPL) with explicit permission to generate non-free executables.

Product information, documentation and additional resources specific to XPCE are available from http://www.swi.psy.uva.nl/products/xpce/.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the (Lesser) GNU General Public License for more details.

Titlepage created using XPCE 4.8.10 on Windows-NT 4.0

Last updated February 2002 for XPCE version 6.0.0

Copyright © 1992-2002 University of Amsterdam

Contents

1 Introduction **1**
- 1.1 Organisation of the XPCE documentation . . . 1
- 1.2 Other sources of information . . . 2
- 1.3 Language interfaces . . . 2
- 1.4 Portability . . . 3
  - 1.4.1 Unix/X-windows . . . 3
  - 1.4.2 Win32 (Windows 95 and NT) . . . 3
- 1.5 Look-and-feel . . . 3
- 1.6 A brief history of (X)PCE . . . 4
- 1.7 About this manual . . . 4
- 1.8 Acknowledgements . . . 5

2 Getting started **7**
- 2.1 Starting XPCE/Prolog . . . 7
- 2.2 Prolog ... and what? . . . 7
  - 2.2.1 Creating objects: new . . . 8
  - 2.2.2 Modifying object state: send . . . 8
  - 2.2.3 Querying objects: get . . . 9
  - 2.2.4 Removing objects: free . . . 10
- 2.3 Optional arguments . . . 10
- 2.4 Named arguments . . . 11
- 2.5 Argument conversion . . . 11
- 2.6 Send and get with more arguments . . . 12
- 2.7 Notation . . . 12
- 2.8 Example: show files in directory . . . 12
- 2.9 Summary . . . 15

3 Using the online manual **17**
- 3.1 Overview . . . 17
- 3.2 Notational conventions . . . 18
  - 3.2.1 Argument types . . . 19
- 3.3 Guided tour . . . 20
  - 3.3.1 Class browser . . . 20
  - 3.3.2 Reading cards . . . 23
  - 3.3.3 Search tool . . . 24
  - 3.3.4 Class hierarchy . . . 24
- 3.4 Summary . . . 26

4 Dialog (controller) windows **27**
- 4.1 An example . . . 27

4.2 Built-in dialog items . . . 29
4.3 Layout in dialog windows . . . 29
4.3.1 Practical usage and problems . . . 31
4.4 Modal dialogs: prompting for answers . . . 32
4.4.1 Example: a simple editor for multiple fonts . . . 33
4.5 Editing attributes . . . 36
4.5.1 Example: editing attributes of a graphical . . . 37

5 Simple graphics 39
5.1 Graphical building blocks . . . 39
5.1.1 Available primitive graphical objects . . . 41
5.2 Compound graphicals . . . 41
5.3 Connecting graphical objects . . . 41
5.4 Constraints . . . 42
5.5 Activating graphicals using the mouse . . . 44
5.6 Summary . . . 45

6 XPCE and Prolog 47
6.1 XPCE is not Prolog! . . . 47
6.2 Dealing with Prolog data . . . 48
6.2.1 Life-time of Prolog terms in XPCE . . . 50

7 Defining classes 51
7.1 The class definition skeleton . . . 51
7.1.1 Definition of the template elements . . . 51
7.2 Accessing instance variables (slots) . . . 54
7.3 Refining and redefining methods . . . 56
7.3.1 General redefinitions . . . 57
7.3.2 Redefinition in graphical classes . . . 58
7.4 Handling default arguments . . . 61
7.5 Advanced topics . . . 62
7.5.1 More on type declarations . . . 62
7.5.2 Methods with variable number of arguments . . . 62
7.5.3 Implementation notes . . . 64

8 Class Variables 67
8.1 Accessing Class Variables . . . 67
8.2 Class variable and instance variables . . . 67
8.3 The ‘Defaults’ file . . . 68
8.4 Class variables in User Defined Classes . . . 69

9 Program resources 71

10 Programming techniques 73
10.1 Control-structure of XPCE/Prolog applications . . . 75
10.1.1 Event-driven applications . . . 75
10.1.2 XPCE and existing applications . . . 77
10.2 Executable objects . . . 79

10.2.1 Procedures . . . 79
10.2.2 Functions . . . 80
10.2.3 Example 1: Finding objects . . . 82
10.2.4 Example 2: Internal behaviour of dialog window . . . 82
10.3 Defining global named objects . . . 85
10.3.1 Using directives . . . 85
10.3.2 Inline testing . . . 85
10.3.3 The 'pce_global' directive . . . 86
10.3.4 Global objects for recognisers . . . 86
10.4 Using object references: "Who's Who?" . . . 89
10.4.1 Global named references . . . 89
10.4.2 Using the prolog database . . . 90
10.4.3 Using object-level attributes . . . 91
10.4.4 Using window and graphical behaviour . . . 92
10.4.5 Using user defined classes . . . 92
10.4.6 Summary . . . 93
10.5 Relating frames . . . 95
10.5.1 Class application . . . 95
10.5.2 Transient frames . . . 95
10.5.3 Modal operation . . . 96
10.6 Window layout in a frame . . . 97
10.6.1 Windows sizes and automatic adjustment . . . 98
10.6.2 Manipulating an open frame . . . 98
10.7 Informing the user . . . 101
10.7.1 Aim of the report mechanism . . . 101
10.7.2 The report interface . . . 101
10.7.3 Redefining report handling . . . 102
10.7.4 Example . . . 102
10.8 Errors . . . 105
10.8.1 Handling errors in the application . . . 105
10.8.2 Raising errors . . . 106
10.8.3 Repairable errors . . . 107
10.9 Specifying fonts . . . 109
10.9.1 Physical fonts . . . 109
10.9.2 Logical fonts . . . 110
10.10 Using images and cursors . . . 113
10.10.1 Colour handling . . . 113
10.10.2 Supported Image Formats . . . 114
10.11 Using hyper links to relate objects . . . 117
10.11.1 Programming existence dependencies . . . 117
10.11.2 Methods for handling hyper objects . . . 119
10.12 User defined graphicals . . . 121
10.12.1 (Re)defining the repaint method . . . 121
10.12.2 Example-I: a window with a grid . . . 122
10.12.3 Example-II: a shape with text . . . 124
10.13 Printing from XPCE applications . . . 127
10.13.1 Options for document generation . . . 127

11 Commonly used libraries **129**
11.1 Library “find_file” . . . 131
11.2 Showing help-balloons . . . 133
11.3 Dialog support libraries . . . 135
11.3.1 Reporting errors and warnings . . . 135
11.3.2 Toolbar support . . . 135
11.3.3 Example . . . 136
11.4 Library “pce_toc”: displaying hierarchies . . . 139
11.5 Tabular layout . . . 143
11.5.1 Using `format` . . . 143
11.5.2 Using `table` using the “tabular” library . . . 144
11.6 Plotting graphs and barcharts . . . 149
11.6.1 Painting axis . . . 149
11.6.2 Plotting graphs . . . 151
11.6.3 Drawing barcharts using “plot/barchart” . . . 153
11.7 Multi-lingual applications . . . 159
11.8 Drag and drop interface . . . 163
11.8.1 Related methods . . . 164
11.9 Playing WEB (HTTP) server . . . 167
11.9.1 Class `httpd` . . . 170
11.9.2 The “http/html_write” library . . . 172
11.10 Document rendering primitives . . . 179
11.10.1 The rendering library . . . 180
11.10.2 Predefined objects . . . 181
11.10.3 Class and method reference . . . 181
11.10.4 Using the “doc/emit” library . . . 186
11.11 Library “broadcast” for all your deliveries . . . 189

12 Development and debugging tools **193**
12.1 Object-base consistency . . . 193
12.2 Tracing methods . . . 193
12.3 Breaking (spy) on methods . . . 194
12.4 Visual hierarchy tool . . . 194
12.5 Inspector tool . . . 194

A The dialog editor **199**
A.1 Guided tour . . . 199
A.1.1 Creating the target dialog window . . . 199
A.1.2 Adding controls to the new window . . . 200
A.1.3 Defining the layout . . . 201
A.1.4 Specifying the behaviour . . . 201
A.1.5 Generating source code . . . 202
A.1.6 Linking the source code . . . 202
A.1.7 Summary . . . 205
A.2 Miscellaneous topics . . . 205
A.2.1 Specifying callback to prolog . . . 205
A.2.2 Advanced example of behaviour . . . 205

A.2.3 Specifying conditional actions . . . 208
A.2.4 Load and save formats . . . 208
A.3 Status and problems . . . 210
A.4 Summary and Conclusions . . . 210

B Notes on XPCE for MS-Windows 211
B.1 Currently unsupported features in the Win32 version . . . 211
B.2 Interprocess communication, extensions and interaction . . . 211
B.3 Accessing Windows Graphics Resources . . . 212
B.4 Accessing Windows Colours . . . 212
B.5 Accessing Windows Fonts . . . 212
B.6 Accessing Windows Cursors . . . 215

C XPCE/Prolog architecture 217
C.1 What is “Object-Oriented”? . . . 217
C.2 XPCE’s objects . . . 217
C.2.1 Classes . . . 218
C.3 Objects and integers . . . 218
C.4 Delegation . . . 218
C.5 Prolog . . . 220
C.6 Executable objects . . . 221
C.7 Summary . . . 221

D Interface predicate definition 223
D.1 Basic predicates . . . 223
D.1.1 Portable declaration of required library predicates . . . 228
D.2 Additional interface libraries . . . 228
D.2.1 Library “pce_util” . . . 228
D.2.2 Library “pce_debug” . . . 230
D.2.3 Accessing the XPCE manual . . . 231

E Memory management 233
E.1 Lifetime of an object . . . 233
E.2 Practical considerations . . . 234
E.3 Memory usage of objects . . . 234

F Commonly encountered problems 237

G Glossary 239

H Class summary descriptions 243

1 Introduction

XPCE is an object-oriented library for building Graphical User Interfaces (GUI's) for symbolic or strongly typed languages. It provides high level GUI specification primitives and dynamic modification of the program to allow for rapid development of interfaces. It integrates a graphical tool for the specification of interfaces, in addition to powerful and uniform mechanisms to facilitate automatic generation of GUI's.

XPCE is not a programming language in the traditional sense. Language constructs and objects of the system do not have a direct textual representation. The interface to the 'hosting' language defines what XPCE looks like from the programmers point of view. As a consequence, the programmer will first of all experience XPCE as a *library*.

XPCE however, does provide all semantic elements that can be found in many object-oriented programming languages: classes, objects, methods, instance-variables, inheritance, statements, conditions, iteration, etc.

All the above primitives are represented by first-class *objects* that may be created, modified, inspected and destroyed. This allows the programmer to extend the XPCE object-oriented system with new methods and classes from the host-language. In addition, procedures can be expressed as objects and then given to XPCE for execution.

The interface between XPCE and its hosting language is small, which makes XPCE especially a good GUI candidate for special-purpose languages.

The main target language for XPCE is *Prolog* and this document concentrates on XPCE/Prolog rather then XPCE/Lisp or XPCE/C++. XPCE/Prolog comes with a graphical programming environment that allows for quick browsing of the source-code, provides debugging tools and allows for the graphical construction of *dialog boxes* (graphical windows with *controllers*). XPCE's built-in editor is modelled after the standard (GNU-)Emacs editor and can be programmed in XPCE/Prolog.

1.1 Organisation of the XPCE documentation

This document describes the basics of XPCE and its relation to Prolog. Besides the written version, this document is also available as an HTML document from the URL below. The second URL may be used to download the entire WWW tree for installation on a local host.

```
http://www.swi.psy.uva.nl/projects/xpce/UserGuide/
ftp://ftp.swi.psy.uva.nl/xpce/HTML/UserGuide.tgz
```

This document provides the background material needed to understand the other documentation:

- The XPCE Reference Manual
  [Wielemaker & Anjewierden, 1993] The reference manual is available by means of the

Prolog predicate `manpce/0`. The reference manual provides detailed descriptions of all classes, methods, etc. which may be accessed from various viewpoints. See also chapter 3.

- PceDraw: An example of using XPCE
  [Wielemaker, 1992] This document contains the annotated sources of the drawing tool PceDraw. It illustrates the (graphical) functionality of XPCE and is useful as a source of examples.

- XPCE/Prolog Course Notes
  [Wielemaker, 1994] Course-notes, examples and exercises for programming XPCE/Prolog. The course-notes have a large overlap in contents with this user guide, but the material is more concise. If you are familiar with object-oriented systems, Prolog and graphical user interfaces the course-notes might be a quick alternative to this user guide.

1.2 Other sources of information

Various other information can be found on or through the XPCE WEB-home:

```
http://www.swi.psy.uva.nl/projects/xpce/
```

Utility programs, recent examples, documentation, etc. can be found in the primary XPCE anonymous ftp archive:

```
ftp://ftp.swi.psy.uva.nl/xpce/
```

There is a mailing list for exchanging information and problems between programmers as well as for us to announce new releases and developments. The address is `xpce@swi.psy.uva.nl`. Please send mail to `xpce-request@swi.psy.uva.nl` to subscribe or unsubscribe to this list. This E-mail address can also be used to communicate with the authors. The address `xpce-bugs@swi.psy.uva.nl` should be used for reporting bugs.

1.3 Language interfaces

The interface between XPCE and the application (host) language is very small. This feature makes it easy to connect XPCE to a new language. XPCE fits best with dynamically typed or strongly statically typed languages with type-conversion facilities that can be programmed. XPCE itself is dynamically typed. Cooperating with languages with the named properties avoid the need for explicitly programmed type-conversion. For a dynamically typed host-language such as Prolog or Lisp, the interface determines the type of the host-language construct passed and translates it into the corresponding XPCE object. For C++, the rules for translating C data structures to XPCE objects can be handled by the programmable type-casting mechanism of C++.

1.4 Portability

The XPCE virtual machine and built-in class library is written in standard ANSI-C and is portable to any machine offering a flat, sufficiently large, memory model (32 or 64 bits). XPCE's graphical classes (including windows, etc.) interface to XPCE *Virtual Windows System* (VWS). Currently there are VWS implementations for X11 and the Microsoft Win32 API. Please contact the authors if you are interested in other implementations.

1.4.1 Unix/X-windows

XPCE runs on most Unix/X11 platforms. Tested platforms include SunOs, Solaris, AIX, HPUX, IRIX, OSF/1 and Linux. Platform configuration is realised using GNU autoconf with an extensive test-suite.

1.4.2 Win32 (Windows 95 and NT)

The same binary version of XPCE runs on both Windows 95 and NT. Its functionality is very close to the Unix/X11 version, making applications source-code compatible between the two platforms. .

A detailed description of the differences between the Unix/X11 version and the Windows version as well as additions to the Windows version to access Windows-specific resources is in appendix B.

1.5 Look-and-feel

XPCE is not implemented on top of a standard UI library such as Motif, OpenWindows, or Win32. Instead, it is built on top of its own VWS defining primitives to create and manipulate windows, draw primitives such as lines, circles, text and handle user-events.

As a consequence, XPCE programs are fully compatible over the available platforms, except that some (almost exclusively obscure) features may have a different or have no effect on some implementations.

The implementation of all of XPCE on top of its primitive graphicals guarantees there are no platform-specific limitations in the manipulation and semantics of certain controllers. XPCE defines the look-and-feel for each of the controllers. As a consequence, XPCE controllers may not behave exactly the same as controllers of other applications in the same windowing environment.

All good things come at a price-tag and portability based on a virtual environment is no exception to this rule. XPCE builds high-level controllers (called dialog-items in its jargon) on top of the virtual machine and therefore bypasses the graphical libraries of the hosting system. The same technique is used by many other portable GUI toolkits, among which Java.

The visual feedback (look) and to some extend the reactions to user actions (feel) of the XPCE controllers is determined by XPCE's *defaults* file, located in ⟨*pcehome*⟩`/Defaults`. See section 8.

1.6 A brief history of (X)PCE

The “PCE Project” was started in 1985 by Anjo Anjewierden. His aim was to develop a high-level UI environment for (C-)Prolog. The requirements for this environment came from the “Thermodynamics Coach” project in which Paul Kamsteeg used PCE/Prolog to implement the UI for a courseware system for thermodynamics. This system included a ‘scratch-pad’ that allowed the student to create structured drawings of component configurations. The application had to be able to analyse the drawing made by the student.

PCE has been redesigned and largely re-implemented on a SUN workstation using Quintus Prolog and later SWI-Prolog [Wielemaker, 1996] in the Esprit project P1098 (KADS). This project used PCE to implement a knowledge engineering workbench called Shelley [Anjewierden *et al.*, 1990]. During this period PCE/Prolog has been used by various research groups to implement graphical interfaces for applications implemented in Prolog. Most of these interfaces stressed the use of direct-manipulation graphical interfaces. Feedback from these projects has made PCE generally useful and mature.

During the versions 4.0 to 4.5, XPCE was moved from SunView to X-windows and since 4.7 compatibility to the Win32 platform is maintained. In addition, the virtual machine has been made available to the application programmer, allowing for the definition of new XPCE classes. These versions have been used mainly for small internal case-studies to validate the new approach. Larger-scale external usage started from version 4.6 and introduced the vital requirement to reduce incompatible changes to the absolute minimum.

In version 5, the XPCE/Prolog interface was revisited, improving performance and making it possible to pass native Prolog data to XPCE classes defined in Prolog as well as associate native Prolog data with XPCE objects. Various new graphical primitives, among which HTML-4 like tables and graphical primitives for rendering markup containing a mixture of graphics and text.

As of XPCE 5.1, the license terms have been changed from a proprietary license schema to the open source GPL-2 licence.

As of XPCE 6.0, the licence terms have been changed from GPL to the more permissive LGPL for the XPCE kernel (compiled C-part) and GPL with an exception allowing for generating non-free applications with XPCE for the Prolog libraries. Please visit the SWI-Prolog home page at `http://www.swi-prolog.org` for details.

1.7 About this manual

qThis userguide introduces the basics of XPCE/Prolog and its development environment. Chapter 2, “Getting Started” explains the interface. Chapter 3, “Using the online manual” introduces the online documentation tools. These are introduced early, as many of the examples in this manual introduce classes and methods without explaining them. The online manual tool can be used to find the definitions of these constructs quickly. The chapter 5 and chapter 4, “Dialog (controller) windows” and “Simple Graphics” introduce the various controller and graphical primitives.

With the material of the above described chapters, the user is sufficiently informed to create simple GUI’s from predefined XPCE objects. The remaining chapters provide the background information and techniques that allow for the design of larger graphical systems.

Chapter 6, “The relation between XPCE and Prolog” is a brief intermezzo, explaining the relation between XPCE and Prolog data in more detail. Chapter 7, “Defining classes” explain the definition of new XPCE classes from Prolog and thus brings object-oriented programming to the user. Chapter 10, “Programming techniques” is an assorted collection of hints on how XPCE can be used to solve real-world problems elegantly. Chapter 11, “Commonly used libraries” documents some of the commonly used XPCE/Prolog libraries.

Chapter 12, “Development and debugging tools” introduces the XPCE debugger. The current debugger is powerful, but not very intuitive and requires a nice-looking front-end.

Of the appendices, appendix H is probably the most useful, providing a short description of each class and its relation to other classes. Many of the classes are accompanied with a small example.

1.8 Acknowledgements

The development of XPCE was started by Anjo Anjewierden. The package was then called PCE. He designed and implemented version 1 and 2. Version 3 is the result of a joint effort by Anjo Anjewierden and Jan Wielemaker.

XPCE-4, offering support for X-windows and user-defined classes, has been implemented by Jan Wielemaker. The implementation of user-defined classes was initiated when Jan Wielemaker was guest at SERC (Software Engineering Research Centre). Gert Florijn has contributed in the initial discussions on user-defined classes. Frans Heeman has been the first user.

The interface to SICStus Prolog has been implemented in cooperation with Stefan Andersson and Mats Carlsson from SICS.

The interface to Quintus Prolog was initiated by Paul-Holmes Higgins. The project was realised by James Little, Mike Vines and Simon Heywood from AIIL.

Luca Passani has bothered us with many questions, but was so kind to organise this material and make it available to other XPCE programmers in the form of a FAQ.

Gertjan van Heijst has commented on XPCE as well as earlier drafts of this documents.

(X)PCE is used by many people. They have often been puzzled by bugs, incompatibilities with older versions, etc. We would like to thank them for their patience and remarks.

2 Getting started

This section introduces programming the XPCE/Prolog environment: the entities (objects), referencing objects and manipulating objects. Most of the material is introduced with examples. A complete definition of the interface primitives is given in appendix D.

2.1 Starting XPCE/Prolog

XPCE is distributed as a library on top of the hosting Prolog system. For use with SWI-Prolog, this library is auto-loaded as soon as one of its predicates (such as `new/2`) is accessed or it can be loaded explicitly using

```
:- use_module(library(pce)).
```

In Unix XPCE/SWI-Prolog distribution the program `xpce` is a symbolic link to `pl` and causes the system to pull in and announce the XPCE library with the banner:

```
% xpce
XPCE 6.0.0, February 2002 for i686-gnu-linux-gnu and X11R6
Copyright (C) 1993-2002 University of Amsterdam.
XPCE comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
The host-language is SWI-Prolog version 5.0.0

For HELP on prolog, please type help. or apropos(topic).
         on xpce, please type manpce.

1 ?-
```

SWI-Prolog's prompt is "⟨*n*⟩ ?-" where ⟨*n*⟩ is the history-number of the command. The banner indicates the XPCE version. The indicated version is 5.1 and the patch-level is 0.

On MS-Windows, Prolog programs are normally loaded and started by double-clicking a `..pl` file. XPCE, being a normal library, does not change this. Note that XPCE can only be used fully with the GUI-based `plwin.exe`. Using the the console-based `plcon.exe` program only the non-GUI functionality of XPCE is accessible.

2.2 Prolog ... and what?

This section describes the four basic Prolog predicates used to control XPCE from Prolog. These four predicates map onto the basic functions of XPCE's virtual machine: creating, destroying, manipulating and querying *objects*, the basic entities of XPCE.

For those not familiar with this jargon, an object is an entity with a state and associated procedures, called *methods*. Objects may represent just about anything. In XPCE's world there are objects representing a position in a two-dimensional plane as well as an entire window on your screen. Each object belongs to a *class*. The class defines the constituents of the state as well as the procedures associated with the object. For example, a position in a two-dimensional plane is represented by an object of class `point`. The state of a point object consists of its X- and Y-coordinates. A point has methods to set the X- and Y-coordinate, mirror the point over a reference point, compute its distance to another point, etc.

2.2.1 Creating objects: new

The predicate `new/2` (new(?Reference, +NewTerm)) *creates* an object in the XPCE world and either assigns the given *reference* to it or unifies the first argument with a XPCE generated reference. An (object-) reference is a unique handle used in further communication with the object. Below are some examples (`?-` is the Prolog prompt):

```
1 ?- new(P, point(10,20)).
P = @772024

2 ?- new(@demo, dialog('Demo Window')).
```

The first example creates an instance of class point from the arguments '10' and '20'. The reference is represented in Prolog using the prefix operator `@/1`. For XPCE generated references the argument of this term is a XPCE generated integer value. These integers are guaranteed to be unique. The second example creates a dialog object. A dialog is a window that is specialised for displaying controllers such as buttons, text-entry-fields, etc. In this example we have specified the reference. Such a reference must be of the form `@Atom`. XPCE will associate the created object with this reference.[1]

As illustrated by the examples above, the second argument to `new/2` is a term. The principal functor denotes the name of the class of which an instance is created and the arguments are the initialisation parameters. The complete transformation rules are given in appendix D.

As stated before, an object has a state. At creation time, the initial state is defined by the class from which the object is created and the initialisation arguments. In our example, the point is assigned an x-value of 10 and and y-value of 20. The dialog is assigned the label 'Demo Window'. A dialog window has many *slots*[2] The example defines the 'label'. All the other slots are set to the default value described in the class definition.

2.2.2 Modifying object state: send

The state of an object may be manipulated using the predicate `send/2` (send(+Receiver, +Selector(...Args...))). The first argument of this predicate is an object reference. The second

[1] Normal applications use almost exclusively XPCE generated references. Many of the examples in this manual are typed from the terminal and Prolog specified references are easier to type.

[2] The attributes of an object state are called slots. In other languages they may be called *instance variables* or fields.

is a term. The principal functor of which is the name of the method to invoke (*selector*) and the arguments are arguments to the operation.

```
3 ?- send(@772024, x(15)).
4 ?- send(@demo, append(text_item(name))).
```

The first example invokes the *method* 'x' of the point object. It sets the instance variable x of the corresponding point object to the argument value. The second example invokes the method 'append' of class dialog. This method appends a UI component to the dialog window. The component is specified by the term 'text_item(name)', which is converted into an object just as the second argument of `new/2`. The query below opens the dialog window.

```
5 ?- send(@demo, open).
```

If everything is ok, a window as shown in figure 2.1 appears on your screen. The border (in the figure this is the title-bar displayed above the window) is determined by the window manager you are using. It should look the same as any other window on your terminal. If an error of any kind appears, please refer to appendix F.

Figure 2.1: Example Dialog Window

2.2.3 Querying objects: get

The next fundamental interface predicate is `get/3`. It is used to obtain information on the state of objects. The first two arguments are the same as for `send/2`. The last argument is unified with the return-value. The return value is normally an object reference, except for XPCE `name` objects, that are returned as a Prolog atom, XPCE integers (int) that are translated to Prolog integers and XPCE `real` objects, that are translated to Prolog floating point numbers. Examples:

```
6 ?- get(@772024, y, Y).
Y = 20
7 ?- get(@demo, display, D).
D = @display/display
8 ?- get(@772024, distance(point(100,100)), Distance).
Distance = 117
```

The first example just obtains the value of the 'y' instance variable. The second example returns the display object on which `@demo` is displayed. This is the reference to an object

of class display that represents your screen.[3] The last example again shows the creation of objects from the arguments to `send/2` and `get/3` and also shows that the returned value does not need to be a direct instance variable of the object. The return value is an integer representing the (rounded) distance between `@772024` and point(100,100).

The second example illustrates that `get/3` returns objects by their reference. This reference may be used for further queries. The example below computes the width and height of your screen.

```
9 ?- get(@display, size, Size),
     get(Size, width, W),
     get(Size, height, H).
Size = @4653322, W = 1152, H = 900
```

As a final example, type something in the text entry field and try the following:

```
10 ?- get(@demo, member(name), TextItem),
      get(TextItem, selection, Text).
TextItem = @573481/text_item, Text = hello
```

The first get operation requests a member of the dialog with the given name (‘name’). This will return the object reference of the text_item object appended to the dialog. The next request obtains the ‘selection’ of the text_item. This is the text typed in by the user.

2.2.4 Removing objects: free

The final principal interface predicate is `free/1`. Its argument is an object reference as returned by `new/2` or `get/3`. It will remove the object from the XPCE object base. Examples:

```
12 ?- free(@772024).
13 ?- free(@demo).
14 ?- free(@display).
No
```

The second example not only removed the dialog window object from the XPCE object base, it also removes the associated window from the screen. The last example illustrates that certain system objects have been protected against freeing.

2.3 Optional arguments

Arguments to XPCE methods are identified by their position. Many methods have both obligatory and optional arguments. If obligatory arguments are omitted XPCE will generate an error. If optional arguments are omitted XPCE fills the argument with the constant `@default`. The interpretation of `@default` is left to the implementation of the receiving method. See also chapter 3.

[3] Prolog would normally print ‘`@display`’. The library(`pce_portray`) defines a clause for the Prolog predicate `portray/1` that prints object references as ‘`@Reference/Class`’. This library is used throughout all the examples of this manual.

2.4 Named arguments

Some methods take a lot of arguments of which you generally only want to specify a few. A good example is the creation of a `style` object. A style is an object used to control the attributes of displayed text: font, fore- and background colour, underline, etc. Its →`initialise` method, serving the same role the *constructor* in C++, takes 7 arguments. Both calls below create a style object representing an underlined text-fragment:

```
1 ?- new(X, style(@default, @default, @default, @default, @on)).
2 ?- new(X, style(underline := @on)).
```

The names of arguments are specified in the reference manual. For example, the method '`area → set`', used to set one or more of the X-, Y-, H- and W-attributes of a rectangle, has the specification given below. For each argument that is specified as `@default`, the corresponding slot will not be changed.

```
area->set: x=[int], y=[int], width=[int], height=[int]
```

The following example illustrates the usage of this method:

```
1 ?- new(A, area),
     send(A, set(y := 10, height := 50)).
```

2.5 Argument conversion

Arguments to XPCE methods are *typed* and variables are dynamically typed. This combination is used for two purposes: automatic conversion and raising exceptions at the earliest possible point if conversion is not defined.

For example, the send-method 'colour' on class `graphical` specifies accepts a single argument of type 'colour'. A colour in XPCE is represented by a colour *object*. Colour objects may be created from their name. The natural way to specify a box should be coloured 'red' is:

```
...,
send(Box, colour(colour(red))),
...
```

Because the →`colour` method knows that it expects an instance of class colour, and because class colour defines the conversion from a name to a colour (see section 7.3.1), the following message achieves the same goal:

```
...,
send(Box, colour(red)),
...
```

Some other examples of classes defining type conversion are `font`, `image`, `name`, `string`, `real` and the non-object data item *int*. The following messages are thus valid:

```
...,
send(Box, x('10')),             % atom --> int
send(Box, selected(true)),      % atom --> boolean
send(Text, font(bold)),         % atom --> font
send(Text, string(10)),         % int  --> string
...
```

2.6 Send and get with more arguments

Though the principal predicates for invoking behaviour are `send/2` and `get/3`, XPCE provides an alternative using `send/[2-12]` and `get/[3-13]`. The following goals are all identical.

```
send(Box, width(100))                  send(Box, width, 100)
get(Point, distance(point(10,10), D)   get(Point, distance, point(10,1), D)
```

This alternative is provided for compatibility to pre-5.0 versions as well as to support users that dislike the new-style `send/2` and `get/3`. It is realised using `goal_expansion/2` and thus poses only a small compile-time overhead.

2.7 Notation

This manual, as well as all other XPCE documentation both printed and online uses some notational conventions. Instead of speaking of 'the send-method colour of class box', we write

'`box` → `colour`'

Similar, instead of 'the get-method height of class window', we write

'`window` ← `height`'

In some cases, the arguments and/or the return type of the method are specified:

'`box` → `colour`: colour'
'`window` ← `height` ⟶int'

2.8 Example: show files in directory

In this section we illustrate the above with some simple examples. We also show how the GUI can call procedures in the Prolog system.

The demo creates a list of files from a given directory and allows the user to view a file. Figure 2.2 shows the result.

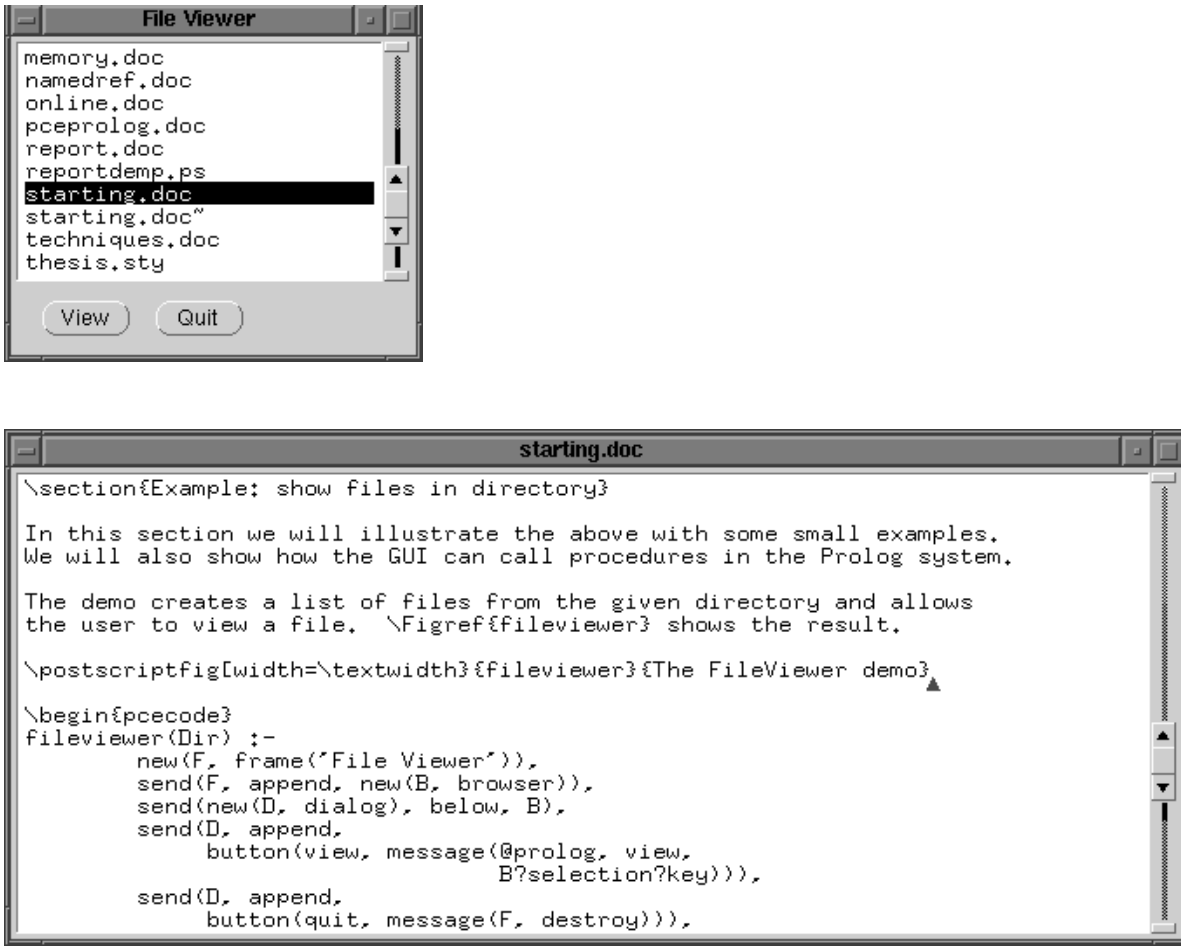


Figure 2.2: The FileViewer demo

```
1  fileviewer(Dir) :-
2          new(F, frame('File Viewer')),
3          send(F, append(new(B, browser))),
4          send(new(D, dialog), below(B)),
5          send(D, append(button(view,
6                                message(@prolog, view,
7                                        B?selection?key)))),
8          send(D, append(button(quit,
9                                message(F, destroy)))),
10         send(B, members(directory(Dir)?files)),
11         send(F, open).
12
13 view(F) :-
14         send(new(V, view(F)), open),
15         send(V, load(F)).
```

The main window of the application consists of a `frame` holding two specialised `window` instances. A `frame` is a collection of *tiled* windows. Any opened window in XPCE is enclosed in a `frame`. The windows are positioned using the methods →`above`, →`below`, →`right` and →`left`. The frame ensures that its member windows are properly aligned and handles resizing of the frame. See also section 10.6.

In line 3, a `browser` is →`append`ed to the frame object. A browser is a window specialised for displaying a list of objects. Next, a `dialog` is positioned below the browser. A dialog is a window specialised in handling the layout of controllers, or `dialog_item` objects as XPCE calls them.

Line 5 adds a `button` to the dialog. 'view' specifies the *name* of the button. XPCE defines a central mapping from 'dialog_item ⇆`name`' to the label displayed. The default mapping capitalises the name and replaces underscores with spaces. In section 11.7, we describe how this can be used to realise multi-lingual applications. The second argument of the button specifies the *action* associated with the button. A `message` is a dormant send-operation. When pressed, the button executes

```
send(@prolog, view, B?selection?key)
```

If a message is sent to `@prolog`, this calls a predicate with the name of the *selector* of the message and an arity that equals the number of arguments to the message (1 here).

The argument is a term with principal functor `?` and defines an 'obtainer', a dormant get-operation. It is defined as a Prolog infix operator of type `yfx`, priority 500. This implies that `B?selection?key` should be read as[4]

```
?(?(B, selection), key)
```

The result of the get-operation ←`selection` on a browser returns the `dict_item` object currently selected. Dict-items are the elements of a `dict`, the underlying data object of a browser. A `dict_item` consists of a ←`key` (its identifier), a ←`label` (the visual text) and optionally an associated ←`object`.

[4]Initial versions of XPCE lacked the obtainer. In this case the browser *B* would be passed to the predicate `view/1`, which would extract the current filename from the browser. Obtainers improve the readability and avoid the need to mix UI related code with application code.

Line 8 appends a second button to the dialog window. The dialog window will automatically align this one to the right of the first. The action sends a message directly to another XPCE object and →`destroys` the frame object if the quit button is pressed. Note that this will erase all UI objects associated with the frame. The garbage collector destroys all related objects.

Line 10 fills the `browser` with the files from the specified directory. The expression `directory(Dir)?files` defines an obtainer operating on an instance of class `directory`. The obtainer evaluates to a `chain`, XPCE's notion of a list, holding the names of all files in the directory. This expression is the same as:

```
...,
new(Dobj, directory(Dir)),
get(Dobj, files, Chain),
send(B, members, Chain),
...
```

Again, the garbage collector takes care of the directory and chain objects. The browser automatically converts the given names to `dict_item` objects.[5]

Finally, the frame is →`opened`. This will cause the frame to ask each of the windows to compute its desired size, after which the frame aligns the windows, decides on their final geometry and creates the Window-system counterpart.

The `view/1` callback predicate simply opens an instance of `view`, a window specialised in text-editing and loads the file into the view. The method '`view → load`' expects an instance of class `file`. Again, the type-conversion will deal with this.

2.9 Summary

XPCE's world consists of objects. An object is an entity with persistent state that belongs to a class. The XPCE/Prolog interface defines four basic predicates, `new/2` to create objects from a description and returns an object reference, `send/[2-12]` to change the state of an object and succeeds if the requested change could be made, `get/[3-13]` to request an object to compute a value and return it, and `free/1` to remove objects from the XPCE database.

Objects of the `message` are 'dormant' send operations. They may be activated by other objects (button, text_item, ...). In this case a send operation is started. Objects of class `?` are called *obtainer* and represent 'dormant' get operations. The '?' sign is defined as a prolog infix operator, allowing for constructs as:

```
send(Frame, height, Frame?display?height)
```

The object `@prolog` (class `host`) allows calling Prolog predicates from the XPCE environment.

[5] This conversion is implemented with class `dict_item`. In fact, the browser just specifies the desired type and the message passing kernel invokes the conversion method of class `dict_item`.

3 Using the online manual

In the previous sections we have introduced XPCE using examples. The language primitives of XPCE are simple, but XPCE as a whole is a massive library of very diverse classes with many methods. The current system contains about 160 classes defining over 2700 methods.

To help you finding your way around in the package as well as in libraries and private code loaded in XPCE, an integrated manual is provided. This manual extracts all information, except for a natural language description of the class, method or slot from the system and thus guarantees exact and consistent information on available classes, methods, slots, types, inheritance in the system.

The manual consists of a number of search tools using different entry points to the material. A successful query displays the summary information for the relevant hyper-cards. Clicking on the summary displays the cards themselves and hyper-links between the cards aid to quickly browse the environment of related material.

3.1 Overview

The online manual consists of a large set of tools to examine different aspects of the XPCE/Prolog environment and to navigate through the available material from different viewpoints.

The inheritance hierarchy Browsers/Class Hierarchy

The 'Class Hierarchy' tool allows the user to examine XPCE's class hierarchy. This tool reads the inheritance relations from the class objects and thus also visualises application or library classes. Figure C.2 is created using this tool.

The structure of a class Browsers/Class Browser

The most important tool is the 'Class Browser'. It provides the user with a view of material related to a class. As everything in XPCE is an object and thus an instance of a class this tool provides access to everything in XPCE, except for the Prolog interface.

Search Tool Browsers/Search

This tool provides full search capabilities on the entire manual contents, including combined search specifications.

Globally available object references Browsers/Global Objects

The XPCE environment provides predefined objects (`@pce`, `@prolog`, `@arg1`, etc.). The tool allows the user to find these objects.

Prolog interface predicates Browsers/Prolog Predicates

This tool documents all the XPCE/Prolog predicates.

Instances Tools/Inspector

This tool is part of the runtime support system. It allows you to inspect the persistent state associated with objects.

Structure of User Interface Tools/Visual Hierarchy

This tool provides a 'consists-of' view of all displayed visual objects. It provides a quick overview of the structure of an interface. It is a useful for finding object-references, examining the structure of an unknown UI and verifying that your program created the expected structure.

The manual itself (help) File/Help

The manual tools are documented by itself. Each tool has a 'Help' button that documents the tool.

XPCE **Demo programs** File/Demo Programs

The 'Demo Programs' entry of the 'File' menu starts an overview of the available demo programs. A demo can be started by double-clicking it. The sources of the demos may be found in ⟨*home*⟩`/prolog/demo`, where ⟨*home*⟩ refers to the XPCE installation directory, which may be obtained using

```
1 ?- get(@pce, home, Home).
Home = '/usr/local/lib/pl-4.0.0/xpce'
```

Note that the DemoBrowser allows to view the sources of the main file of a demo application immediately. Also consider using the VisualHierarchy and ClassBrowser to analyse the structure of the demo programs.

3.2 Notational conventions

The text shown by the online manual uses some notational conventions. The various overview tools indicate candidate documentation cards with a *summary line*. This line is of the form:

⟨*Identifier*⟩ ⟨*Formal Description*⟩ ["⟨*Summary*⟩"]

The 'Identifier' is a single letter indicating the nature of the documentation card. The defined identifiers are: **B**rowser (Manual Tool), **C**lass, **E**xample, **K**eyword, **M**ethod, **O**bject, **P**redicate, **R**esource, **T**opic and **V**ariable (instance-variable).

The 'Formal Description' is a short description derived from the described object itself:

V class - selector: type	Variable that cannot be accessed directly
V class <- selector: type	Variable that may be read, but not written
V class <->selector: type	Variable that may be read and written
V class ->selector: type	Variable that may only be written
M class ->selector: type ...	Send-Method with argument-types
M class <- selector: type ... -->type	Get-Method with argument-types returning value of type
R Class.attribute: type	Class-variable with type

The same notational conventions are used in the running text of a card. See section 3.3.2.

3.2.1 Argument types

XPCE is a partially typed language. Types may be defined for both method arguments and instance variables. A type is represented by an instance of class `type`. XPCE defines a conversion to create type objects from a textual representation. A full description of this conversion may be found in the online manual (method '`type` ← `convert`'). In this document we will summarise the most important types:

- int
  XPCE integer datum.
- ⟨*low*⟩..⟨*high*⟩, ⟨*low*⟩.., ..⟨*high*⟩
  Range of integers (including ⟨*low*⟩ and ⟨*high*⟩). The latter two constructs indicate one-side-unbound integer. Both ⟨*low*⟩ and ⟨*high*⟩ can also be floating point numbers, indication a 'real-range'.
- any
  Both integers and objects. Function objects will be evaluated. See section 10.2.2.
- ⟨*class-name*⟩
  Any instance of this class or one of its sub-classes. Class `object` is the root of the inheritance hierarchy. The type `object` is interpreted slightly different, as it does *not* accept instances of class `function` or its subclasses. This implies that the type `object` forces functions to be evaluated.
- [⟨*type*⟩]
  Either this type or `@default`. Trailing arguments to methods that accept `@default` may be omitted.
- ⟨*type*⟩*
  Either this type or `@nil`.
- ⟨*type*⟩ ...
  Methods with this type specification accept any number of arguments that satisfy ⟨*type*⟩.
- {⟨*atom1*⟩,⟨*atom2*⟩,...}
  Any of these name objects.

For example, the →`initialise` method of a graphical text object has type declaration:

```
[char_array], [{left,center,right}], [font]
```

The first argument is an instance of class `char_array`, the super-class of `name` and `string`. The second argument either 'left', 'center' or 'right' and the last argument is a font object. All arguments are between square brackets and may thus be omitted.

3.3 Guided tour

This section provides a 'guided tour' through the manual system. If you have XPCE/Prolog at hand, please start it and try the examples. For each of the central tools of the manual we will present a screendump in a typical situation and explain the purpose and some common ways to use the tool.

3.3.1 Class browser

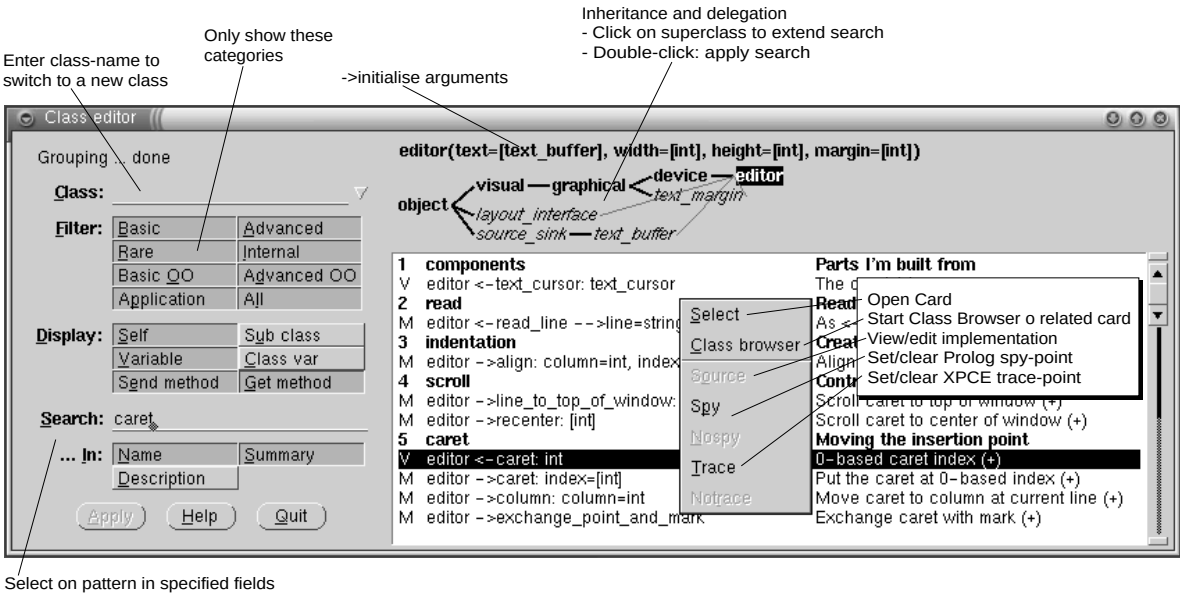


Figure 3.1: The Class Browser

The "Class Browser" is the central tool of the online manual. It provides an overview of the functionality of a class and options to limit the displayed information. Figure 3.1 shows this tool in a typical situation. In this example the user is interested in methods dealing with 'caret' in an `editor`.

The dialog to the left-side of the tool specifies *what* information is displayed. The top-right window displays the class together with the initialisation arguments (the arguments needed to create an instance of this class). Double-left-click on this text will open the description for →`initialise`.

Below this text a hierarchy is displayed that indicates the place in the inheritance hierarchy as well as the classes to which messages are *delegated* (see section C.4). The user can

select multiple classes only if there is delegation and the tree actually has branches. Use class `editor` or class view to explore these facilities. After the user has selected one or more classes, the Apply button makes the class-browser search for methods in all classes below the selected classes. If a method is found in multiple classes the class-browser will automatically display only the one method that will actually be used by this class.

The large right window displays a list of matching classes, variables, methods and class-variables. If an item is tagged with a "(+)" there is additional information that may be obtained by (double-) clicking the card to start the "Card Viewer" (see section 3.3.2).

The ClassBrowser dialog

The **Class** text_item (text-entry-field) may be used to switch to a new class. Note that this text_item implements completion (bound to the space-bar).

The **Filter** menu filters the candidate objects according to their categorisation. Selecting all switches off filtering, which is often useful in combination with **Search**. Clicking all again switches back to the old selection of categories. The meaning of the categories is:

- Basic
  Principal methods that are used very often. This is, together with *Application*, the default selection of this menu.
- Advanced
  Less often used and sometimes complicated methods.
- Rare
  Infrequently used methods. Note that does not mean they are complicated or do things you'd hardly ever want to use. For example, most of the caret-manipulation of class editor is in this category. It is essential and commonly used behaviour of the editor, but rarely used directly from program-control.
- Internal
  Behaviour that is not directly intended for public usage. It may be useful to understand how other methods interact. Try to avoid using these methods or variables in your code.
- Basic OO
  Methods intended to be redefined in user-defined classes. See chapter 7.
- Advanced OO
  Methods that may be redefined in user-defined classes, but for which this is far less common.
- Application
  Methods implemented in the host-language.

The **Display** menu determines the objects searched for. Self refers to the class itself, Sub Class refers to the direct sub classes of this class. The other fields refer to instance-variables, methods with send- and get-access and class-variables.

The **Search** and **... In** controllers limit the displayed cards to those that have the specified search string in one of the specified fields. While searching, the case of the characters is ignored (i.e. lower- and uppercase versions of the same letter match). Searching in the Name field is useful to find a particular method if the name (or part of it) is known.

Example queries to the classbrowser

Below we illustrate how some commonly asked questions may be answered with the class browser.

- What are variables of a bitmap?
  Select variable in the **Display** menu, clear **Search**, and set **Filter** to All. Then type 'bitmap' in **Class** and hit return. Note that by double-clicking on class `graphical` in the inheritance display not only the variables of class `bitmap` itself are shown, but also those of class `graphical`.
- How can I position the caret in an editor?
  The caret can only be changed using send-methods. Either the name or the summary is likely to have 'caret' as a substring. Thus, **Display** is set to Send Method, **Field** to Name and Summary, search 'caret'.

Methods with special meaning

This section describes the role of the 'special' methods. These are methods that are not used directly, but they define the behaviour of `new/2`, type conversion, etc. and knowing about them is therefore essential for understanding an XPCE class.

object → **initialise:** ⟨*Class-Defined*⟩
: The →`initialise` method of a class defines what happens when an instance of this class is created. It may be compared to the constructor in C++. Note that double-clicking the class description in the class-browser (top-right window) opens the reference card for the →`initialise` method. See also `new/2`, section 2.2.1.

object → **unlink**
: The →`unlink` method describes what happens when an instance of this class is removed from the object-base and may be compared to the C++ destructor.

object ← **lookup:** ⟨*Class-Defined*⟩ → **object**
: If defined, this method describes the lookup an already defined instance instead of object creation. For example

```
1 ?- new(X, font(screen, roman, 13)).
X = @screen_roman_13
2 ?- new(Y, font(screen, roman, 13)).
Y = @screen_roman_13
```

: The same instance of the reusable font instance is returned on a second attempt to create a font from the same parameters. Examples of classes with this capability are: `name`, `font`, `colour`, `image` and `modifier`.

object ← convert: ⟨*Class-Defined*⟩ **→ object**
Defines what can be converted into an instance of this type. If an instance of this class is requested by a type but another object is provided XPCE will call this method to translate the given argument into an instance of this class.

object → catch_all: ⟨*Class-Defined*⟩
The →`catch_all` method defines what happens with messages invoked on this object that are not implemented by any other method.

object ← catch_all: ⟨*Class-Defined*⟩ **→ any**
As →`catch_all`, but for get-operations.

3.3.2 Reading cards

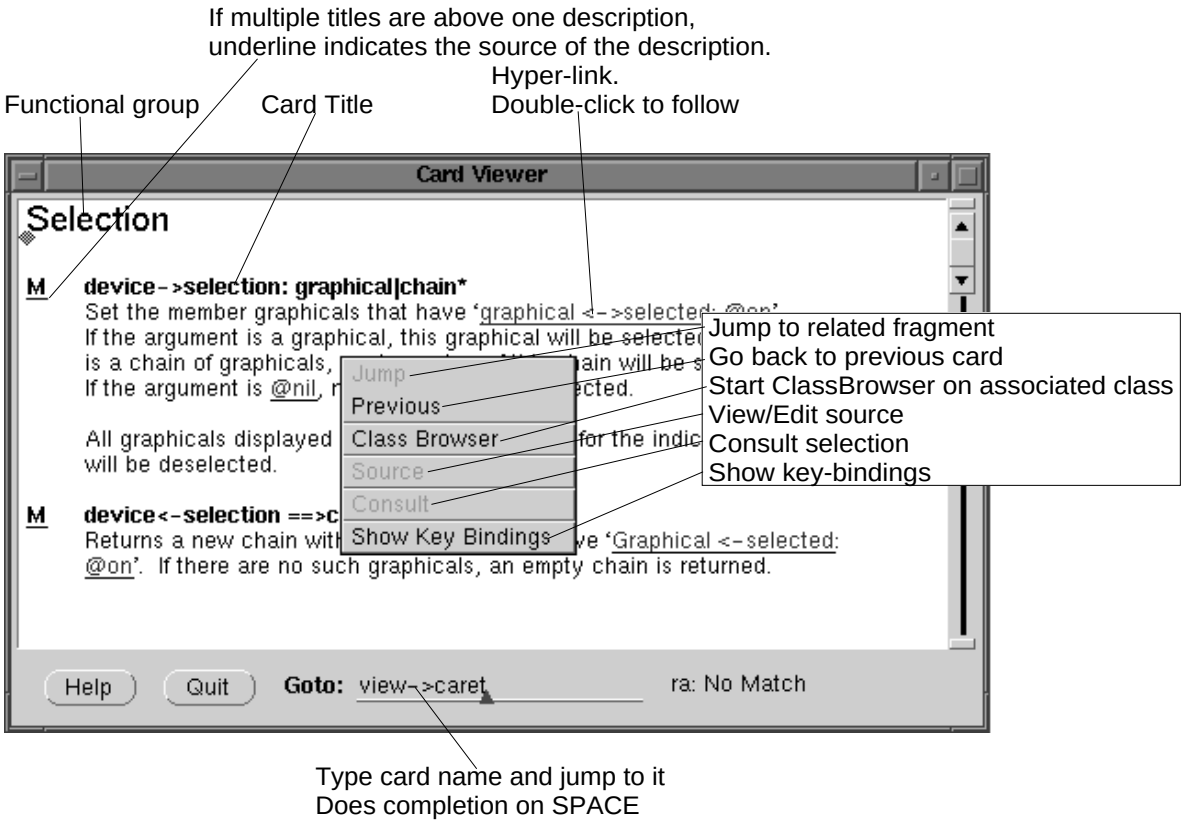


Figure 3.2: The Card Viewer

The other tools of the manual allow the user to *find* cards with documentation on the topic(s) the user is looking for. The information provided by the summary-lists often suffices for this purpose. Whenever a card is marked with a "(+)" in the summary list it may be opened by double-clicking it. This starts the "Card Viewer" tool. Figure 3.2 is a screendump of this tool showing the 'selection' group of class 'device'.

The "Card Viewer" displays the formal information and all available attributes from the card related to the displayed object (method, variable, class, ...). It uses patterns to deter-

mine relations to other manual material from the text. Each hit of these patterns is highlighted. When the user double-clicks on highlighted text the “Card Viewer” will jump to the related material.

If the user double-clicks a group-title in the ClassBrowser, all cards in the group will be displayed in the CardViewer. Some objects share their documentation with another object. Opening the card for such an object will show two titles above the card. The card from which the documentation originates will have an underlined type-indicator.

The **Goto** field allows for switching to a new card. The syntax for this field is similar to `manpce/1`, `tracepce/1` and `editpce/1` predicates description in section 12. It consists of a classname, followed by `->` to indicate a send-method, `<-` for a get-method and `-` to specify an instance-variable without considering associated methods.

The item performs *completion* bound to the space-bar. The first word is completed to a class-name. The second to a send-method, variable or get-method. Method completion considers inheritance and delegation.[1]

3.3.3 Search tool

The search tool is shown in figure 3.3. It allows the user to search through all XPCE manual cards in an efficient manner with queries similar to that what is found in WAIS tools. A search specification is an expression formed from the following primitives:

- Word
  Specifies all cards containing a word for which the search specification is the *prefix*. Case is ignored.
- <Word>
  Specifies all cards that contain the indicated word. Case is ignored.
- Expr1 `and` Expr2
  Specifies all cards satisfying both conditions.
- Expr1 `or` Expr2
  Specifies all cards satisfying either condition.

As a special shorthand, just specifying multiple words refers to all cards containing all these words.

If the user stops typing for more than a second, the system will parse the expression and display the number of matching cards.

The `browser` window on the left contains all words occurring anywhere in the manual. The window on the right is used to display the card summaries of all matching cards.

3.3.4 Class hierarchy

The “Class Hierachy” tool shown in figure 3.4 may be used to get an overview of XPCE’s class hierarchy or to find the (inheritance) relations of a particular class with other classes.

[1] Given the dynamic nature of delegation, the system cannot possibly determine all methods available through delegation. Consider a slot specified with type `graphical`. The system can infer it will surely be able to use behaviour defined at class `graphical`. If at runtime, the slot is filled with a `box`, all methods defined at class box will be available too.

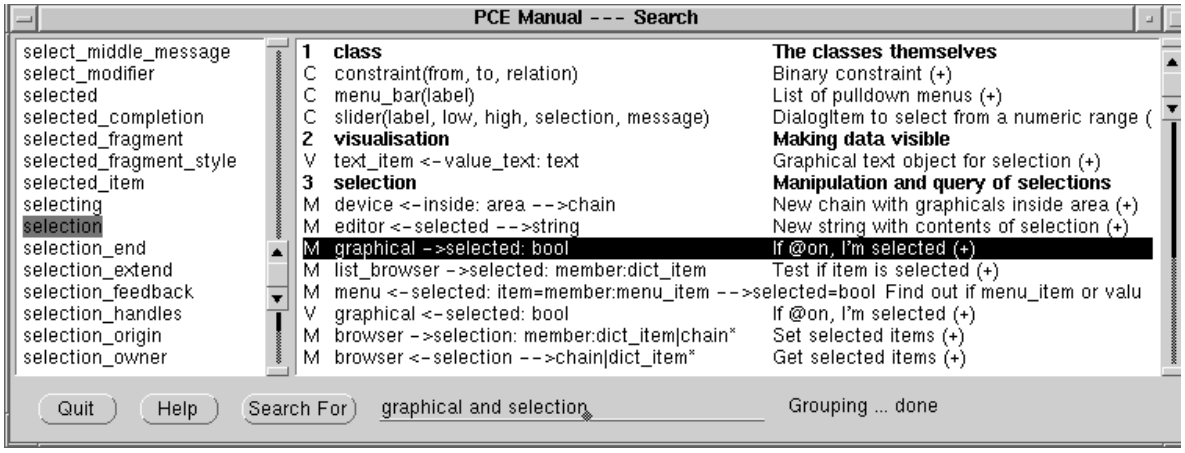


Figure 3.3: Manual search tool

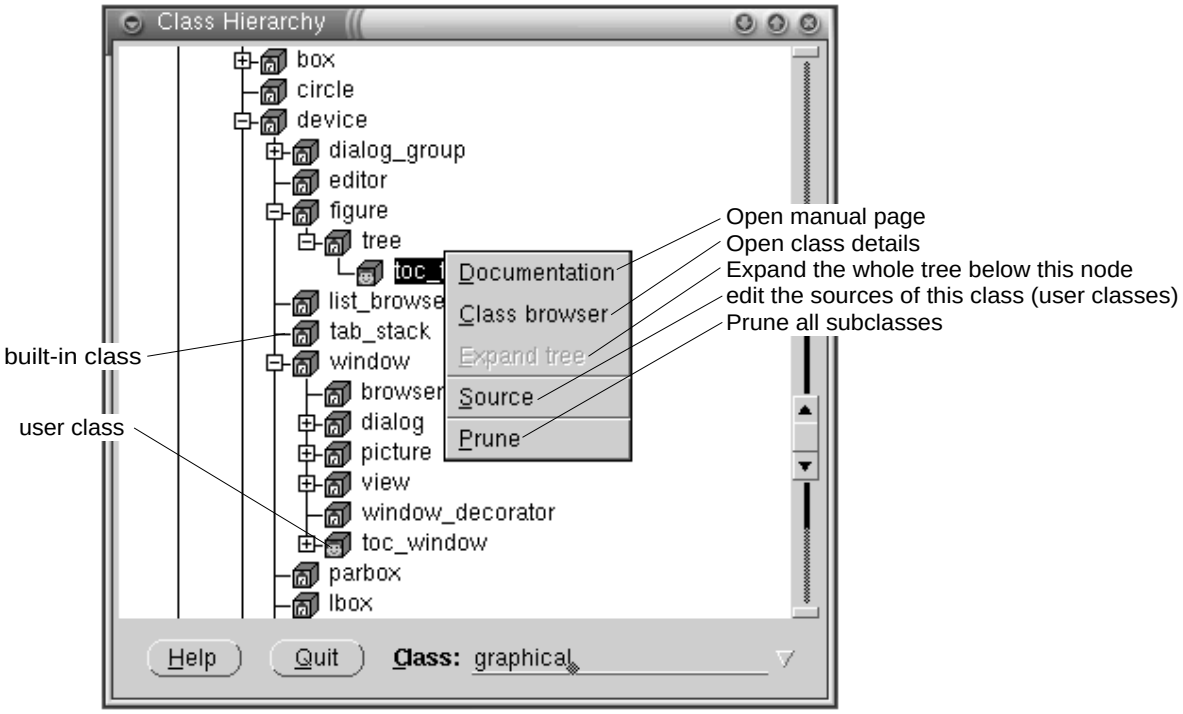


Figure 3.4: Class Hierachy Tool

Note that XPCE's inheritance hierarchy has a technical foundation rather than a conceptual. Super-classes are motivated by the need for code-sharing.

3.4 Summary

The online manuals integrate visualisation of XPCE's internal structure with a hyper-text system. This approach guarantees consistency between the documentation and the actual system and integrates overview and documentation of library and user-defined classes in one system.

The online manual tools provides various entry-points (classes, global objects, predicate overview, keywords, etc.) to obtain a list of *card summaries*. Cards may be opened from these summary lists to examine its contents.

4 Dialog (controller) windows

XPCE Dialog windows are normally used to display a number of controllers, named `dialog_items` in XPCE's jargon. Class `dialog` is a subclass of `window` with specialised methods for positioning controllers. Dialog items are graphical objects specialised for displaying and/or editing particular data. Figure 4.1 illustrates the inheritance relations relevant to dialog windows and the locations of the most important methods.

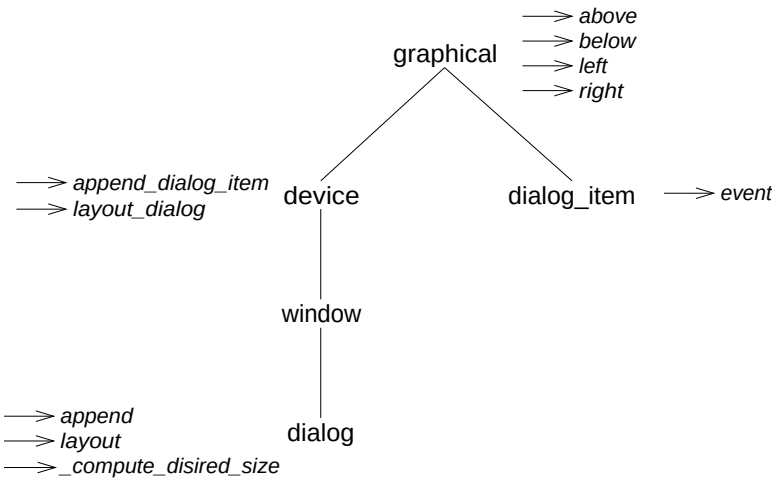


Figure 4.1: Dialog Inheritance Hierarchy

Dialogs can be created both by using the `new/2` and `send/[2-12]` operations as well as by using the Dialog Editor which is described in appendix A. This section describes the first mechanism. Reading this chapter will help you understanding the dialog editor.

4.1 An example

Before diving into the complexities we will illustrate normal usage through an example. The following Prolog predicate creates a dialog for entering information on an employee. The result, running on Windows-NT, is shown in figure 4.2.

```
1 ask_employee :-
2         new(Dialog, dialog('Define employee')),
3         send_list(Dialog, append,
4                   [ new(N1, text_item(first_name)),
5                     new(N2, text_item(family_name)),
6                     new(S,  new(S, menu(sex))),
```

```
                    new(A,  int_item(age, low := 18, high := 65)),
                    new(D,  menu(department, cycle)),
                    button(cancel, message(Dialog, destroy)),
                    button(enter, and(message(@prolog,
                                              assert_employee,
                                              N1?selection,
                                              N2?selection,
                                              S?selection,
                                              A?selection,
                                              D?selection),
                                      message(Dialog, destroy)))
                ]),
        send_list(S, append, [male, female]),
        send_list(D, append, [research, development, marketing]),
        send(Dialog, default_button, enter),
        send(Dialog, open).

assert_employee(FirstName, FamilyName, Sex, Age, Depth) :-
        format('Adding ~w ~w ~w, age ~w, working at ~w~n',
               [ Sex, FirstName, FamilyName, Age, Depth]).
```

This example shows the layout capabilities of `dialog` and its `dialog_item` objects. Simply appending items will place items vertically and group buttons in rows. Labels are properly aligned. The enter button defines a call-back on the predicate `assert_employee/5` using the values from the various controllers. Section 10.2 explains the use of `message` objects in detail.

Figure 4.2: Enter employee

button	Simple push-button. Executes ←message when pressed.
text_item	A text-entry field. Editable or non-editable, built-in type conversion (for example to enter a numerical value), completion using the space-bar if a value-set is provided.
int_item	Like a text_item, but providing properly sized field, buttons for one-up/down, type- and range-checking.
slider	Select numerical value in a range. Handles both integers and floating point values.
menu	Implements various styles of menus with different visual feedback. Realises radio-button, tick-box, combo-box and much more.
menu_bar	Row of pulldown (popup) menus. Normally displayed in a small dialog above the other windows in the frame.
label	Image or textual label. Normally not sensitive to user actions.
list_browser	Shows a list of items. List-browsers have been designed to handle lists with many items. Class browser is a window-based version.
editor	Powerful text-editor. Handles multiple and proportional fonts, text-attributes, fragment marking, etc. Class view is a window based version.
tab	Tagged sub-dialog, that may be combined with other tabs into a tab_stack, realising a tabbed controller-window. Often seen in modern applications to deal with many setting options.
tab_stack	Stack of tab objects.
dialog_group	Group of dialog items, possible with border and label.

Table 4.1: Built-in dialog items

4.2 Built-in dialog items

Table 4.1 provides an overview of the built-in dialog items. The XPCE/Prolog library defines various additional items as Prolog classes. See the file Overview in the library directory.

4.3 Layout in dialog windows

The layout inside a dialog window may be specified by two means, either using pixel-coordinates or using symbolic layout descriptions. The latter is strongly encouraged, both because it is generally much easier and because the layout will work properly if the end-user uses different preferences (see chapter 8) than the application developer.

The following methods can be used to define the layout of a dialog. All methods actually have both send- and get-versions. The methods listed only as '→send' methods are unlikely to be used as get-methods in application code.

dialog_item → above: dialog_item
dialog_item → below: dialog_item
dialog_item → left: dialog_item
dialog_item → rigth: dialog_item

These relations built a two-dimensional grid of dialog-items and determine the relative positioning of the dialog items. It suffices to relate each dialog item to one other item.

device → append_dialog_item: graphical, [{below,right,next_row}]
dialog → append: graphical, [{below,right,next_row}]
Append a dialog item relative to the last one appended. This method is the principal methods used to fill dialog windows. For the first item, the last argument is ignored. If the last argument is `below`, this item is placed below the previous one. If the argument is `right`, it is placed right of the previous one and if the argument is `next_row`, the item is placed below the first one of the current row of dialog items. If the last argument is `@default`, dialog objects are placed `next_row`, except for buttons, which are placed in rows, left to right.

dialog → gap: size
Defines the distance between rows and columns of items as well as the distance between the bounding box of all items and the window border.

dialog_item ⇆ reference: point
Point relative to the top-left corner that defines the *reference-point* of the dialog item. If two items are aligned horizontally or vertically, it are actually their reference points that are aligned.

dialog_item → alignment: {column,left,center,right}
This attribute controls how items are aligned left-to-right in their row. An item with →`alignment`: `column` will be alignment horizontally using the references of its upper or lower neighbour. Horizontally adjacent items with the same alignment will be flushed left, centered or flushed right if the alignment is one of `left`, `center` or `right`. The alignment value is normally specified as a class-variable and used to determine the layout of rows of `button` objects.

dialog_item → hor_stretch
0..100 After completing the initial layout, possibly remaining horizontal space is distributed proportionally over items that return a non-zero value for this attribute. By default, class `text_item` yields 100 for this value, normally extending text_items as far as possible to the right.

The methods above deal with the placement of items relative to each other. The methods below ensure that columns of items have properly aligned labels and values.

dialog_item ⇆ label_width: [0..]
If the item has a visible label, the label_width is the width of the box in which the label is printed. The dialog layout mechanism will align the labels of items that are placed above each other if ←`auto_label_align` is @on. The argument `@default` assigns the minimum width of the label, the width required by the text of the label.

dialog_item ⇆ label_format: {left,center,right}
Determines how the label is aligned in its box. The values are `left`, `center` and `right`. This value is normally defined by the look and feel.

dialog_item ⇆ value_width: [0..]
: If the item displays multiple values left-to-right (only class `menu` at the moment), '`dialog_item → value_width`' is used to negotiate equal width of the value-boxes similar to →`label_width` if ←`auto_value_align` is `@on`.

The methods listed below activate the layout mechanism. Normally, only '`device → layout_dialog`' needs to be called by the user.

dialog → layout: [size]
device → layout_dialog: gap=[size], size=[size], border=[size]
: Implements the dialog layout mechanism. '`Dialog → layout`' simply calls '`device → layout_dialog`' using '`dialog ← gap`'. '`Device → layout_dialog`' first uses the ←`above`, etc. attributes to build a two-dimensional array of items. Next, it will align the labels and value of items placed in the same column. Then it will determine the size and reference point for each of the items and determine the cell-size. It will then align all items vertically and afterwards horizontally, while considering the '`dialog_item ← alignment`'.

dialog → _compute_desired_size
: Sent from '`frame → fit`' to each of the member windows. For class `dialog`, this activates →`layout` and then computes the desired size of the window.

4.3.1 Practical usage and problems

Most of the above methods are only used rarely for fine-tuning the layout. Almost all dialog windows used in the development environment, demo applications and Prolog library simply use '`dialog → append`', sometimes specifying the last argument.

Two problems are currently not taken care of very well. Aligning multiple objects with a single third object can only be achieved using a sub-dialog in the form of a device and often requires some additional messages. The dialog of figure 4.3 is created using the following code:

```
1  layoutdemo1 :-
2          new(D, dialog('Layout Demo 1')),
3          send(D, append,
4               new(BTS, dialog_group(buttons, group))),
5          send(BTS, gap, size(0, 30)),
6          send(BTS, append, button(add)),
7          send(BTS, append, button(rename), below),
8          send(BTS, append, button(delete), below),
9          send(BTS, layout_dialog),
10         send(D, append, new(LB, list_browser), right),
11         send(D, append, new(TI, text_item(name, ''))),
12         send(LB, alignment, left),
13         send(D, layout),
14         send(LB, bottom_side, BTS?bottom_side),
15         send(LB, right_side, TI?right_side),
16         send(D, open).
```

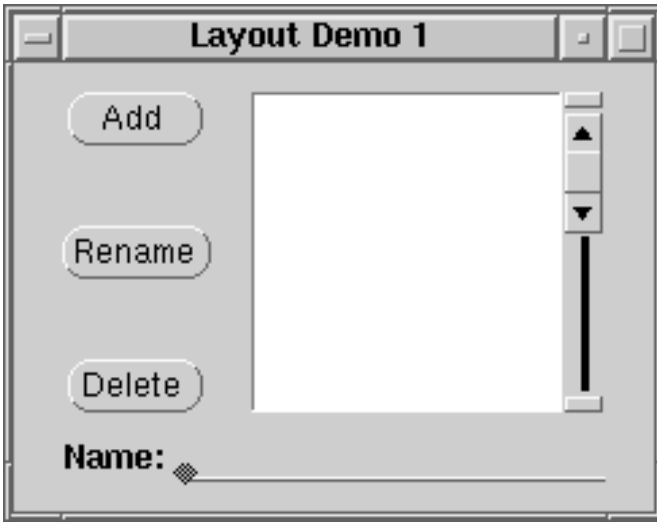


Figure 4.3: Aligning multiple items

In line 3, a `device` is added to the dialog to contain the stack of buttons. This device is sent an explicit →`layout_dialog` to position the buttons. Next, the list_browser is placed to the right of this stack and the text_item on the next row.

If you try this layout, the first column will hold the device and the text_item and the list_browser will be placed right of this column and thus right of the text_item. Using '`dialog_item` → `alignment`: left' enforces the list_browser to flush left towards the device. Now we enforce the layout and adjust the bottom and right sides of the list_browser to the device and text_item.

Dialog windows do not reposition their contents if the window is resized in the current implementation. If the window is enlarged, the items stay in the top-left corner. If the window is made smaller, part of the items may become invisible. Resizing can be implemented by the user by trapping the '`window` → `resize_message`'.

4.4 Modal dialogs: prompting for answers

A *modal* dialog is a dialog that is displayed and blocks the application until the user has finished answering the questions posed in the dialog. Modal dialogs are often used to prompt for values needed to create a new entity in the application or for changing settings.

Modal windows are implemented using the methods '`frame` ← `confirm`' and '`frame` → `return`'. '`Frame` ← `confirm`' invokes '`frame` → `open`' if the frame is not visible and then starts reading events and processing them. '`Frame` → `return`: value' causes '`frame` ← `confirm`' to return with the value passed as argument to →`return`. The following code is a very simple example opening a dialog and waiting for the user to enter a name and press RETURN or the Ok button.

```
ask_name(Name) :-
        new(D, dialog('Prompting for name')),
```

```
        send(D, append,
             new(TI, text_item(name, '''))),
        send(D, append,
             button(ok, message(D, return,
                                TI?selection))),
        send(D, append,
             button(cancel, message(D, return, @nil))),
        send(D, default_button, ok), % Ok: default button
        get(D, confirm, Answer),     % This blocks!
        send(D, destroy),
        Answer \== @nil,             % canceled
        Name = Answer.

?- ask_name(X).

X = 'Bob Worker'
```

See also section 10.5 for a discussion on how frames may be related, as well as alternatives for keeping control in Prolog.

4.4.1 Example: a simple editor for multiple fonts

The following example allows the user to select text in an editor and change its appearance. The application is shown in figure 4.4.

A typical XPCE/Prolog module header. Always make sure to load module library(pce) explicitly if you want to write modules portable to the various Prolog dialects supported by XPCE.

```
1  :- module(wysiwyg,
2            [ wysiwyg/1           % +File
3            ]).
4  :- use_module(library(pce)).
5  :- use_module(library(pce_style_item)).
```

Create the main window, consisting of a frame holding a dialog window with a button for defining new styles and a menu for setting the style of the selection. Both dialog items use call-back to `@prolog`.

```
6  wysiwyg(File) :-
7          new(Fr, frame(File)),
8          send(Fr, append, new(D, dialog)),
9          send(new(V, view), below, D),
10         send(V, font, normal),
11         send(D, append,
12              button(define_style,
13                     message(@prolog, define_style, Fr))),
14         send(D, append,
15              menu(style, toggle,
```

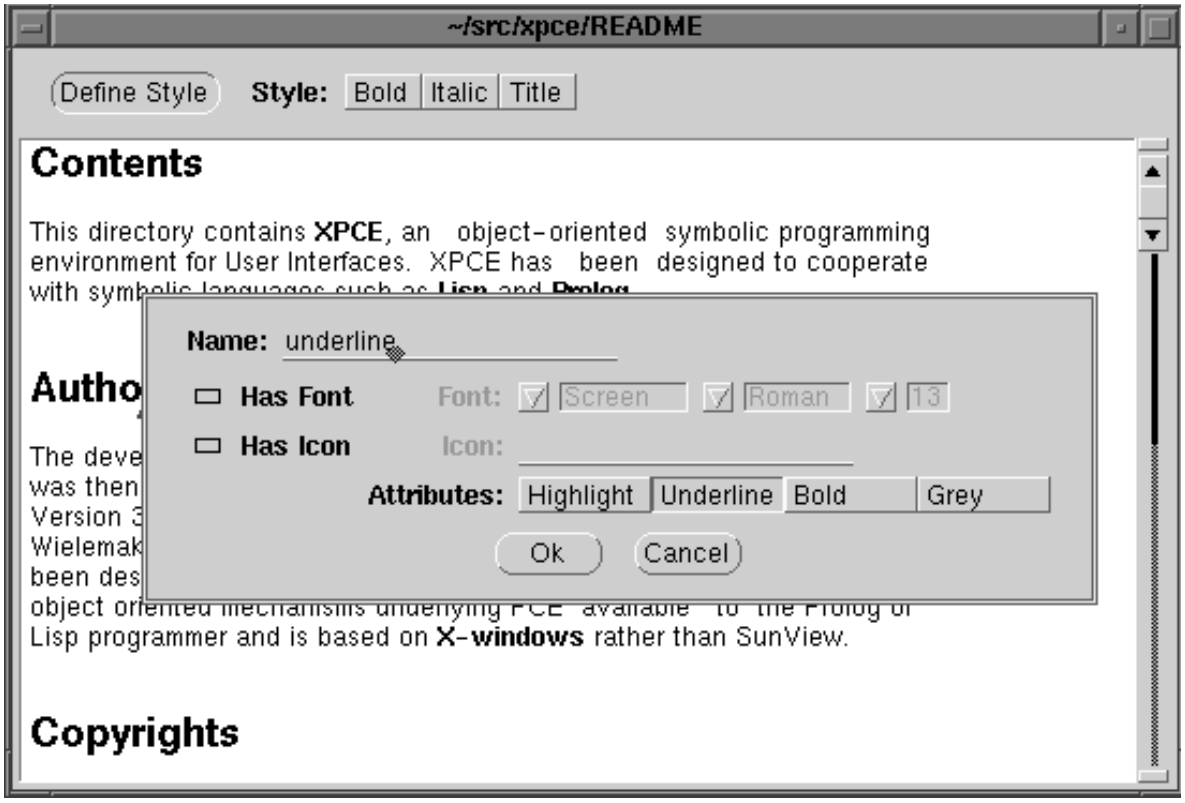


Figure 4.4: Very simple WYSIWYG editor

```
                    and(message(@prolog, set_style, Fr, @arg1),
                        message(V, selection, 0, 0),
                        message(@receiver, clear_selection))),
              right),
        append_style(Fr, bold,   style(font := bold)),
        append_style(Fr, italic, style(font := italic)),
        send(V, load, File),
        send(Fr, open).
```

Set the style for the current selection. Simply pick the selection start and end and make a fragment using the selection parameters and the style-name.

```
set_style(Fr, Style) :-
        get(Fr, member, view, V),
        get(V, selection, point(Start, End)),
        (   Start == End
        ->  send(Fr, report, warning, 'No selection')
        ;   get(V, text_buffer, TB),
            new(_, fragment(TB, Start, End-Start, Style))
        ).
```

Define a new style and add it to the menu and the view.

```
define_style(Fr) :-
        ask_style(Fr, Name, Style),
        append_style(Fr, Name, Style).

append_style(Fr, Name, Style) :-
        get(Fr, member, dialog, D),
        get(D, member, style, Menu),
        send(Menu, append, Name),
        send(Menu, active, @on),
        get(Fr, member, view, View),
        send(View, style, Name, Style).
```

Prompt for the style-name and style-object. Class style_item is defined in the library(pce_style_item). 'frame →`transient_for`' tells the window manager the dialog is a supporting frame for the main application. 'frame ←`confirm_centered`' opens the frame centered around the given location and starts processing events until 'frame →`return`' is activated.

```
ask_style(Fr, Name, Style) :-
        new(D, dialog('Define Style')),
        send(D, append,
             new(N, text_item(name, ''))),
        send(D, append,
             new(S, style_item(style))),
        send(D, append,
             button(ok, message(D, return, ok))),
        send(D, append,
             button(cancel, message(D, return, cancel))),
        send(D, default_button, ok),
        send(D, transient_for, Fr),
```

```
54          repeat,
55          get(D, confirm_centered, Fr?area?center, Answer),
56          (   Answer == ok
57          ->  get(N, selection, Name),
58              (   Name == ''
59              ->  send(D, report, error,
60                       'Please enter a name'),
61                  fail
62              ;   !,
63                  get(S, selection, Style),
64                  send(Style, lock_object, @on),
65                  send(D, destroy)
66              )
67          ;   !,
68              send(D, destroy),
69              fail
70          ).
```

4.5 Editing attributes

In the previous section, we discussed dialogs for entering values. Another typical use of dialog windows is to modify setting of the application, or more in general, edit attributes of existing entities in the application. These entities may both be represented as XPCE objects or as constructs in the host language (dynamic predicates or the recorded database in Prolog).

Such dialog windows first show the current settings. It allows for modifying the controls showing the various aspects of the current state and three buttons providing the following functions:

- Apply
  Apply the current controls, which implies invoking some behaviour on the application to realise the setting of the—modified— controls.
- Restore
  Reset the controls to the current status of the application.
- Cancel
  Destroy the dialog and do not modify the current settings of the application.

The following methods are defined on all primitive controls as well as on the dialog window faciliate the implementations of dialog windows as described above.

dialog_item → default: any|function
dialog_item → restore
For most dialog items, the ⇆`default` value is the second initialisation argument. Instead of a plain value, this can be a `function` object. The initial ←`selection` is set by evaluating this function. In addition, →`restore` will evaluate the function again and reset the selection.

dialog_item → apply: always:bool
Execute the →message of each dialog item for which 'dialog_item ←modified' yields @on. If the argument is @on, the modified flag is not checked.

dialog → apply
dialog → restore
Broadcasts →apply or →restore to each item in the dialog.

4.5.1 Example: editing attributes of a graphical

We will illustrate these methods described above in this example, which implements a dialog for editing the colour of the interior and thickness of the line around a graphical. Double-clicking on a graphical pops up a dialog window for changing these values. The result is show in figure 4.5.

```
1  colour(white).
2  colour(red).
3  colour(green).
4  colour(blue).
5  colour(black).
6
7  append_colour(M, C) :-
8          new(Img, pixmap(@nil, white, black, 32, 16)),
9          send(Img, fill, colour(C)),
10         send(M, append, menu_item(colour(C), label := Img)).
11
12 edit_graphical(Gr) :-
13         new(D, dialog(string('Edit graphical %s', Gr?name))),
14         send(D, append,
15              new(M, menu(colour, choice,
16                          message(Gr, fill_pattern, @arg1)))),
17         send(M, layout, horizontal),
18         forall(colour(C), append_colour(M, C)),
19         send(M, default, Gr?fill_pattern),
20         send(D, append, slider(pen, 0, 10, Gr?pen,
21                                message(Gr, pen, @arg1))),
22         send(D, append, button(apply)),
23         send(D, append, button(restore)),
24         send(D, append, button(quit, message(D, destroy))),
25         send(D, default_button, apply),
26         send(D, open).
27
28 attributedemo :-
29         send(new(P, picture('Attribute Demo')), open),
30         send(P, display,
31              new(B, box(100, 100)), point(20, 20)),
32         send(P, display,
33              new(E, ellipse(100, 50)), point(150, 20)),
34         send_list([B, E], fill_pattern, colour(white)),
35         new(C, click_gesture(left, '', double,
36                              message(@prolog, edit_graphical,
37                                      @receiver))),
```

```
38          send(B, recogniser, C),
39          send(E, recogniser, C).
```

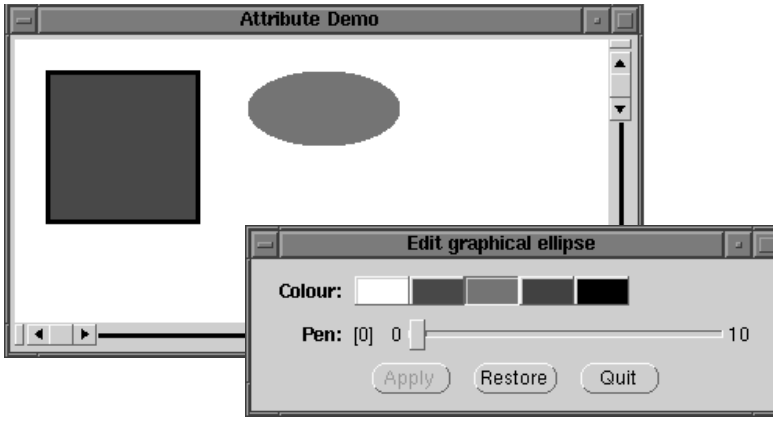


Figure 4.5: Attribute editor for graphical objects

5 Simple graphics

In chapter 2 we introduced the principal predicates of XPCE. For the examples we used controllers, because these are relatively easy to use. In this section we present the basic graphical components. These are more general and therefore can be applied in many more situations, but they are also more difficult to use.

This section only introduces the basics of graphics in XPCE. See also [Wielemaker, 1992]. The online manual and the demo programs provide more information on using XPCE's graphics.

5.1 Graphical building blocks

A `window` is the most generic window class of XPCE. Drawings are often displayed on a `picture`, which is a window with scrollbars. The drawing area of a window is two-dimensional and infinitely large (both positive and negative). The query below creates a picture and opens it on the screen.

```
1 ?- new(@p, picture('Demo Picture')),
     send(@p, open).
```

The following queries draw various primitive graphicals on this picture.

```
2 ?- send(@p, display,
          new(@bo, box(100,100))).
3 ?- send(@p, display,
          new(@ci, circle(50)), point(25,25)).
4 ?- send(@p, display,
          new(@bm, bitmap('32x32/books.xpm')), point(100,100)).
5 ?- send(@p, display,
          new(@tx, text('Hello')), point(120, 50)).
6 ?- send(@p, display,
          new(@bz, bezier_curve(point(50,100),
                                point(120,132),
                                point(50, 160),
                                point(120, 200)))).
```

XPCE's graphics infrastructure automatically takes care of the necessary repaint operations when graphical objects are manipulated. Try the queries below to appreciate this. The result is shown in figure 5.1.

```
7  ?- send(@bo, radius, 10).
8  ?- send(@ci, fill_pattern, colour(orange)).
9  ?- send(@tx, font, font(times, bold, 18)).
10 ?- send(@bz, arrows, both).
```

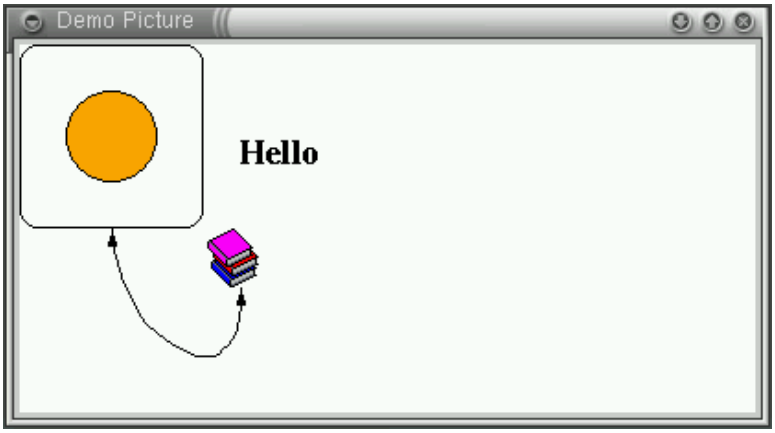


Figure 5.1: Example graphics

XPCE avoids unnecessary repaint operations and expensive computations involved in updating the screen. The screen is only updated *after* all available input has been processed or on an explicit request to update it. The following code illustrates this. Running `?- square_to_circle(@bo).` will show the box immediately as a circle without showing any of the intermediate results.

```
:- require([between/3, forall/2]).

square_to_circle(Box) :-
        get(Box, height, H),
        MaxRadius is H // 2,
        forall(between(0, MaxRadius, Radius),
               send(Box, radius, Radius)).
```

To get the intended animating behaviour, use '`graphical → flush`' to explicitly force redraw right now:

```
:- require([between/3, forall/2]).

square_to_circle(Box) :-
        get(Box, height, H),
        MaxRadius is H // 2,
        forall(between(0, MaxRadius, Radius),
               (   send(Box, radius, Radius),
                   send(Box, flush)
               )).
```

`arrow`	Arrow-head. Normally used implicitly by class `line`.
`bezier`	Bezier curve. Both quadratic and cubic Biezer curves are supported.
`bitmap`	Visualisation of an image. Both monochrome and full-colour images are supported. Images can have shape. See section 10.10.
`pixmap`	Subclass of bitmap only for coloured images.
`box`	Rectangle. Can be rounded and filled.
`circle`	Special case of ellipse.
`ellipse`	Elliptical shape. May be filled.
`arc`	Part of an ellipse. Can have arrows. Can show as pie-slice.
`line`	Straight line segment. Can have arrows.
`path`	Poly-line through multiple points. Can have arrows. Can be smooth.
`text`	Visualisation of a string in some font. Can have various attributes, can be clipped, formatted, etc.

Table 5.1: Primitive graphical objects

5.1.1 Available primitive graphical objects

An overview of the available primitive graphical classes is most easily obtained using the Class Hierarchy tool described in section 3.3.4. Table table 5.1 provides an overview of the primitive graphicals.

5.2 Compound graphicals

Often one would like to combine two or more primitive graphical objects into a single unit. This is achieved using class `device`. Below we create an icon, consisting of a bitmap and a textual label displayed below it.

```
9 ?- new(@ic, device),
     send(@ic, display, bitmap('happy.bm')),
     send(@ic, display, text('Happy'), point(0, 64)),
     send(@p, display, @ic, point(250, 20)).
```

A compound graphical may be treated as a unit. It may be moved, erased, coloured, etc. by sending a single message to the compound. Compound graphicals are normal graphicals and thus may de displayed on other compound graphicals, resulting in a consists-of hierarchy of nested graphicals. See also section 12.4. The classes related to compound graphical objects are shown in table 5.2.

5.3 Connecting graphical objects

The primary application domain of XPCE is handling graphical modelling languages. Drawing in such languages often entails connecting graphical objects using lines. Instead of adding an instance of `line` to the graphical device at the proper place, it is much better to declare two graphical objects to be connected. Class `connection` provides for this.

`device`	Most generic compound graphical object. The `window` is a subclass of `device` and all graphical operations are defined on class `device`.
`figure`	Subclass of `device`, provides clipping, background, containing rectangle, border and the possibility to show a subset of the displayed graphical objects.
`format`	A format object specifies a two-dimensional table layout. Formats may be associated to graphical devices using '`device` → `format`'.
`table`	The successor of `format` realises tabular layout compatible to the HTML-3 model for tables. See section 11.5.

Table 5.2: Compound graphical classes

To prepare an object for making connections, the object should first define `handles`. Below is a simple example. The link is a *reusable* object and therefore defined as a global reference. See section 10.3. The screendump is shown in figure 5.2.

```
:- pce_global(@in_out_link, make_in_out_link).

make_in_out_link(L) :-
        new(L, link(in, out, line(arrows := second))).

linked_box_demo :-
        new(P, picture('Linked Box demo')),
        send(P, open),
        send(P, display, new(B1, box(50,50)), point(20,20)),
        send(P, display, new(B2, box(25,25)), point(100,100)),
        send(B1, handle, handle(w, h/2, in)),
        send(B2, handle, handle(w/2, 0, out)),
        send_list([B1, B2], recogniser, new(move_gesture)),
        send(B1, connect, B2, @in_out_link).
```

If there are multiple handles of the same 'kind' on a graphical, a connection will automatically try to connect to the 'best' handle.

The classes related to making connections are summarised in table 5.3.

Note that, as class `connection` is a subclass of `graphical`, connections can be created between connections. Class `graphical` defines various methods to help reading the relations expressed with connections and/or refine the generic `connect_gesture`.

5.4 Constraints

XPCE allows the user to specify constraints between pairs of objects. In the example above we would like the text to be centered relative to the bitmap. This may be achieved using:

```
10 ?- get(@ic, member, bitmap, Bitmap),
      get(@ic, member, text, Text),
      new(_, constraint(Bitmap, Text, identity(center_x))).
```

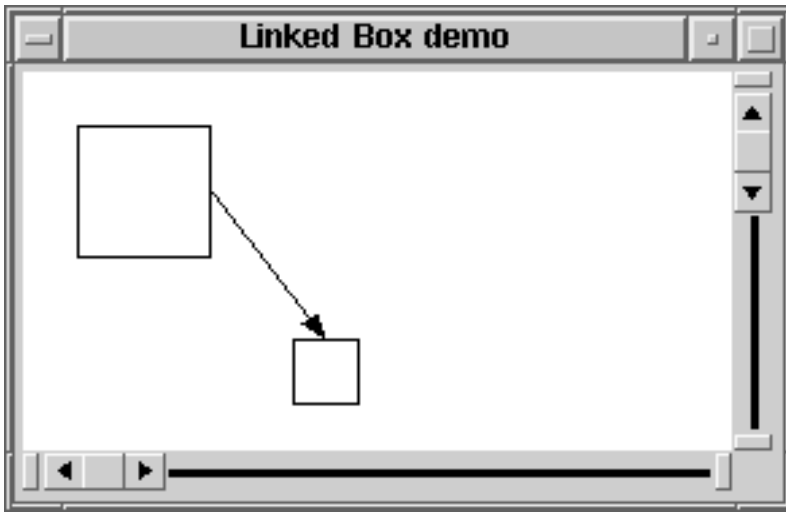


Figure 5.2: A connection between two boxes

`connection`	Subclass of class `line`. A connection can connect two graphicals on the same window that have handles. The line is automatically updated if either of the graphicals is moved, resized, changed from device, (un)displayed, hidden/exposed or destroyed.
`handle`	Defines the location, nature and name of a connection point for a connection. Handles can be attached to individual graphicals as well as to their class.
`link`	Defines the generic properties of a connection: the nature ('kind') of the handle at either side and the line attributes (arrows, pen and colour).
`connect_gesture`	Event-processing object (see section 5.5) used to connect two graphical objects.

Table 5.3: Classes used to define connections

handler	Binds a single event to a message.
handler_group	Combines multiple recognisers into a single.
key_binding	Maps keyboard sequences to commands.
click_gesture	Maps a mouse-click to a message. Allows to specify modifiers (alt/meta, control, shift), button and multi (single, double, triple).
connect_gesture	Connect two graphicals dragging from the first to the second.
move_gesture	Move graphical by dragging it.
move_outline_gesture	Move graphical by dragging an outline.
resize_gesture	Resize graphical by dragging a side or corner.
resize_outline_gesture	Resize graphical by dragging a side or corner of the outline.

Table 5.4: Recogniser classes

Each time either the bitmap or the text changes this constraint will invoke ←center_x on the changed object and →center_x with the return value on the other object. Class spatial defines more general geometrical constraints between graphicals.

Constraints are high-level, but potentially expensive means to specify graphical relations. An alternative is the redefinition of the →geometry method of (compound) graphical objects. See chapter 7.

5.5 Activating graphicals using the mouse

Recogniser objects enable detection of mouse- and keyboard activities. XPCE defines both primitive and complex recognisers. The first (called handler) processes a single event. The latter processes a gesture: sequence of events starting with a mouse-button-down up to the corresponding mouse-button-up. The following example allows us to move the icon by dragging with the middle mouse button:

```
11 ?- send(@ic, recogniser, new(move_gesture)).
```

The second example allows us to double-click on the icon. This is a common way to ‘open’ an icon. In the example we will just print ’hello’ in the Prolog window.

```
12 ?- send(@ic, recogniser,
           click_gesture(left, '', double,
                         message(@pce, write_ln, hello))).
```

The predefined recogniser classes are summarised in table 5.4. Besides the built-in recognisers, the XPCE/Prolog library defines various additional ones. See also section 11.8.

5.6 Summary

In this section we have introduced some of the graphics capabilities of XPCE. XPCE's graphics are built from primitive and compound graphicals. A compound graphical has its own coordinate system in which it can display any graphical object including other compound objects. Graphical objects can be connected to each others using a `connection`. This facility makes the generation of graphs relatively simple. It also makes it very simple to extract the graph represented by a drawing made by the user.

Graphical objects are made sensitive to mouse and keyboard activities by attaching `recogniser` objects to them. XPCE defines standard recognisers for various complex operations such as moving, resizing, popup-menu's, linking graphicals and clicking on graphicals.

6 XPCE and Prolog

XPCE and Prolog are very different systems based on a very different programming paradigm. XPCE objects have global state and use destructive assignment. XPCE programming constructs use both procedures (code objects and send-methods) and functions (function objects and get-methods). XPCE has no notion of non-determinism as Prolog has.

The hybrid XPCE/Prolog environment allows the user to express functionality both in Prolog and in XPCE. This chapter discusses representation of data and dealing with object-references in XPCE/Prolog.

6.1 XPCE is not Prolog!

Data managed by Prolog consists of logical variables, atoms, integers, floats and compound terms (including lists). XPCE has natural counterparts for atoms (a `name` object), integers (a XPCE int) and floating point numbers (a `real` object). Prolog logical variables and compound terms however have no direct counterpart in the XPCE environment. XPCE has variables (class `var`), but these obey totally different scoping and binding rules.

Where Prolog uses a compound term to represent data that belongs together (e.g. person(Name, Age, Address)), XPCE uses objects for this purpose:[1]

```
:- pce_begin_class(person(name, age, address), object).

variable(name,    name,   both, "Name of the person").
variable(age,     int,    both, "Age in years").
variable(address, string, both, "Full address").

initialise(P, Name:name, Age:int, Address:string) :->
        "Create from name, age and address"::
        send(P, name, Name),
        send(P, age, Age),
        send(P, address, Address).

:- pce_end_class.

1 ?- new(P, person(fred, 30, 'Long Street 45')).
P = @3664437/person
```

These two representations have very different properties:

[1] This example uses XPCE user-defined classes. The details of this mechanism do not matter for the argument in this section. User-defined classes are described in chapter 7.

- Equality
  Prolog cannot distinguish between ‘`person('Fred', 30, 'Long Street 45')`’ and a second instance of the same term. In XPCE two instances of the same class having the same state are different entities.

- Attributes
  Whereas an attribute (argument) of a Prolog term is either a logical variable or instantiated to a Prolog data object, an attribute of an object may be assigned to. The assignment is destructive.

- Types
  XPCE is a dynamically typed language and XPCE object attributes may have types. Prolog is untyped.

6.2 Dealing with Prolog data

By nature, XPCE data is not Prolog data. This implies that anything passed to a XPCE method must be converted from Prolog to something suitable for XPCE. A natural mapping with fast and automatic translation is defined for atoms, and numbers (both integers and floating point). As we have seen in section 2, compound terms are translated into instances using the functor-name as class-name.

In XPCE 5.0 we added the possibility to embed arbitrary Prolog data in an object. There are three cases where Prolog data is passed natively embedded in a instance of the class `prolog_term`.

- *Explicit usage of* `prolog`*(Data)*
  By tagging a Prolog term using the functor `prolog/1`, *Data* is embedded in an instance of `prolog_term`. This term is passed unaltered unless it is passed to a method that does not accept the type Any, in which case translation to an object is enforced.

- *When passed to a method typed* `Prolog`
  Prolog defined methods and instance-variables (see section 7) can define their type as `Prolog`. In this case the data is packed in a `prolog_term` object.

- *When passed to a method typed* `unchecked`
  A few methods in the system don’t do type-checking themselves.

We will explain the complications using examples. First we create a `code` object:

```
1 ?- new(@m, and(message(@prolog, write, @arg1),
                 message(@prolog, nl))).
```

This code object will print the provided argument in the Prolog window followed by a newline:

```
2 ?- send(@m, forward, hello).
hello
```

From this example one might expect that XPCE is transparent to Prolog data. This is true for integers, floats and atoms as these have a natural representation in both languages. However:

```
3 ?- send(@m, forward, chain(hello)).
@774516
4 ?- send(@m, forward, 3 + 4).
7
5 ?- send(@m, forward, [hello, world]).
@608322
```

In all these examples the argument is a Prolog compound term which —according to the definition of `send/3`— is translated into a XPCE instance of the class of the principal functor. In 3) this is an instance of class `chain`. In 4) this is an instance of class +. Class + however is a subclass of the XPCE class `function` and function objects are evaluated when given to a method that does not accept a function-type argument. Example 5) illustrates that a list is converted to a XPCE `chain`.

We can fix these problems using the `prolog/1` functor. Example 7) illustrates that also non-ground terms may be passed.

```
6 ?- send(@m, forward, prolog(chain(hello))).
chain(hello)
7 ?- send(@m, forward, prolog(X)).
_G335

X = _G335
```

Below is a another realistic example of this misconception.

```
1 ?- new(D, dialog('Bug')),
2    send(D, append, button(verbose,
3                           message(@prolog, assert,
4                                   verbose(on)))),
5    send(D, open).
6 [PCE warning: new: Unknown class: verbose
7         in: new(verbose(on)) ]
```

One correct solution for this task is below. An alternative is to call a predicate `set_verbose/0` that realises the assertion.

```
1 make_verbose_dialog :-
2         new(D, dialog('Correct')),
3         send(D, append,
4              button(verbose,
5                     message(@prolog, assert,
6                             prolog(verbose(on))))),
7         send(D, open).
```

6.2.1 Life-time of Prolog terms in XPCE

.

XPCE is connected to Prolog through the foreign language interface. Its interface predicates are passed Prolog terms by reference. Such a reference however is only valid during the execution of the foreign procedure. So, why does the example above work? As soon as the `send/3` in `make_verbose_dialog/0` returns the term-reference holding the term `verbose`(*on*) is no longer valid!

To solve this problem, `prolog_term` has two alternative representations. It is created from a term-reference. After the interface call (`send/3` in this case) returns, it checks whether it has created Prolog term objects. If it finds such an object that is not referenced, it destroys the object. If it finds an object that is referenced it records Prolog terms into the database and stores a reference to the recorded database record.

Summarising, Prolog terms are copied as soon as the method to which they are passed returns. Normally this is the case if a Prolog terms is used to fill an instance-variable in XPCE.

7 Defining classes

The user defined class interface provides a natural way to define new XPCE classes. It is both used to create higher level libraries that have the same interface as the built-in XPCE classes as to define entire applications. Many of the library modules and XPCE/Prolog demo programs are implemented as user-defined classes. The PceDraw demo is an elaborate example defined entirely in user-defined classes.

A user defined class lives in XPCE, just as any other XPCE class. There is no difference. Both use dynamic resolution of messages to method objects and then execute the method object. Both use the same object-management and storage facilities.

XPCE/Prolog user-defined classes have their methods implemented in Prolog. This provides a neat and transparent interface between the two systems.[1]

User defined classes are defined using Prolog syntax, where some operators have special meaning. The definition of an XPCE/Prolog class is enclosed in

```
:- pce_begin_class(<Class>, <Super> [, <Comment>]).
<Class definition>
:- pce_end_class.
```

Multiple classes may be defined in the same Prolog source file, but class definitions may not be nested.

7.1 The class definition skeleton

We introduce the syntax for user-defined classes using a skeleton. Except for the `pce_begin_class/[2,3]` and `pce_end_class/0`, everything in the skeleton is optional and may be repeated multiple times. The order of declarations is not important, but the order of the skeleton is the proposed order. An exception to this rule is the `pce_group/1` directive, that may be placed anywhere and defines the group-identifier for the declarations that follow. The skeleton is given in figure 7.1.

7.1.1 Definition of the template elements

:- pce_begin_class(*+[Meta:]Class, +Super, [+Summary]***)**

Start the definition of an XPCE user-defined class. This directive can appear anywhere

[1] XPCE defines four implementation techniques for methods. *C-function pointers* are used for almost all the built-in behaviour. *C++-function pointers* are used when classes are defined in C++ ([Wielemaker & Anjewierden, 1994]). Instances of `c_pointer` are left to the host object for interpretation and finally, `code` objects are executed.

```
:- pce_begin_class([⟨Meta⟩:]⟨Class⟩[({⟨TermName⟩})], ⟨Super⟩[, ⟨Summary⟩]).

:- use_class_template(⟨TemplateClass⟩).
:- send(@class, ⟨Selector⟩{, ⟨Arg⟩}).
:- pce_class_directive(⟨Goal⟩).

variable(⟨Name⟩, ⟨Type⟩[:= ⟨Value⟩], ⟨Access⟩ [, ⟨Summary⟩]).

delegate_to(⟨VarName⟩).

class_variable(⟨Name⟩, ⟨Type⟩, ⟨Default⟩ [, ⟨Summary⟩]).

handle(⟨X⟩, ⟨Y⟩, ⟨Kind⟩, ⟨Name⟩).

:- pce_group(⟨Group⟩).

⟨SendSelector⟩(⟨Receiver⟩{, ⟨Arg⟩[:[⟨AName⟩=]⟨Type⟩]}) :->
        [⟨Summary⟩::]
        ⟨PrologBody⟩.

⟨GetSelector⟩(⟨Receiver⟩{, ⟨Arg⟩[:[⟨AName⟩=]⟨Type⟩]}, ⟨RVal⟩[:⟨Type⟩]) :<-
        [⟨Summary⟩::]
        ⟨PrologBody⟩.

:- pce_end_class.
```

Figure 7.1: Skeleton for user-defined classes

in a Prolog source file. The definition must be closed using `pce_end_class/0` and definitions may not be nested. *Class* describes the class to be created. Besides giving the class-name, the meta-class (class of the class) may be specified. When omitted, the meta-class of the *Super* will be used, which is normally class `class`. An example of meta-class programming can be found in PceDraw's file `shape.pl`, see [Wielemaker, 1992].

The class-name may be followed by a list of *TermNames* that define the result of `object/2`. `object/2` unifies its second argument with a term whose functor is the name of the class and whose arguments are the result of a 'get' operation using the *TermName* as selector. For example, `point(x,y)` specifies that object(P, T) unifies *T* to a term point/2 with the ←x and ←y of the point instance as arguments. When omitted, the term-description of the super-class is inherited.

:- use_class_template(*TemplateClass***)**
Import a class template. See section 7.5.2.

:- send(@`class`, ...**)**
Directives like this may be used to invoke methods on the class under construction. This can be used to modify the class in ways that are not defined by this preprocessor. The following example tells the system that the 'visual' attribute of an imaginary user-defined class should not be saved to file when the object is saved using '`object` → `_save_in_file`'.

```
:- send(@class, save_style_variable, nil).
```

See also `pce_class_directive/1` and section 7.5.3.

:- pce_class_directive(+*:Goal*)
Define *Goal* to be a goal that manipulates the class instance directly. See section 7.5.3.

variable(*Name, Type, Access, [Summary]***)**
Define a new instance variable. *Name* is the name of the variable, which is local to the class and its subclasses. *Type* defines the type. See section 3.2.1 and section 7.5.1. The type may be postfixed with `:=` *Value*,[2] in which case the instance-allocation will fill the slot with the given value. See also section 7.3.1. *Access* defines which implicit *universal* methods will be associated with the variable. A universal method is defined to be a method that reads or writes the slot, without performing any additional actions. See also section 7.2.

delegate_to(*VariableName***)**
Declares the variable named *VariableName* to be a candidate for delegation. See section C.4.

class_variable(*Name, Type, Default, [Summary]***)**
Declare a class-variable for the class. Class-variables describe common properties for all instances of the class. The *Default* value for a class-variable can de defined in the `Defaults` file. See chapter 8 for details.

[2] Value must be an int, name or `constant` in the current implementation.

The *Default* entry describes the default value if there is no value specified in the `Defaults` file. Example:

```
class_variable(size, size, size(400,200), "De-
fault size of object").
```

handle(*X, Y, Kind, Name***)**
Equivalent to the expression below. See also section 5.3.

```
:- send(@class, handle, handle(X, Y, Kind, Name)).
```

:- pce_group(*GroupIdentifier***)**
Sets the '`behaviour ⇆ group`' attribute of any variable or method definition following this directive. Groups are used to organise methods by the ClassBrowser. Groups have no semantic implications. `:- pce_group(@default).` makes methods inherit their group from the method that is re(de)fined. If no method is re(de)fined, the group will be `miscellaneous`.

:- pce_end_class(*Class***)**
End the definition of the named *Class*. *Class* must be the same as the class-name used by the most recent `pce_begin_class/[2,3]`. This variation of `pce_end_class/0` provides better documentation and error correction.

:- pce_begin_class()
Close the definition of the most recently started class. See also `pce_end_class/1`.

Syntax details

Table table 7.1 describes the details of the non-terminals in the above skeleton in more detail. The notation is an incomplete BNF notation.

7.2 Accessing instance variables (slots)

The method '`object ⇆ slot`' is used to access slots directly, bypassing possible methods with the same name. Normally, it should only be used in →`initialise` (see below) and when defining a method with the same name as a variable. Below is a fragment where a `type` slot is displayed by a text object named `type` in a graphical object. This variable has access `get`, associating a universal method ←`type` that yields the current value of the slot. The implementation of →`type` uses the →`slot` method to write the argument in the ←`type` slot and subsequently performs the required side-effects. The ... indicate where the fragment is incomplete.

```
variable(type,  name,   get, "Epistemological type").

initialise(D, Type:name, ...) :->
        send_super(D, initialise),
```

⟨*Meta*⟩	::=	⟨*Name*⟩	Name of the class this class will be an instance of. Default is the meta-class of the super-class
⟨*Class*⟩	::=	⟨*Name*⟩	Name of the class to be defined
⟨*TermName*⟩	::=	⟨*Name*⟩	Selector name to fetch `object/2` argument. For example, a point is translated into point(⟨*X*⟩, ⟨*Y*⟩) and the description is `point(x,y)`
⟨*Super*⟩	::=	⟨*Name*⟩	Name of the super-class. `object` refers to the most general class
⟨*Summary*⟩	::=	"{⟨*Char*⟩}"	Summary description as appearing in the on-line manual. < 40 characters, no newlines, Prolog string
⟨*TemplateClass*⟩	::=	⟨*Name*⟩	Import a template class. See section 7.5.2
⟨*Selector*⟩	::=	⟨*Name*⟩	Name of a method
⟨*X*⟩	::=	⟨*IntExpr*⟩	See class `handle`
⟨*Y*⟩	::=	⟨*IntExpr*⟩	See class `handle`
⟨*Kind*⟩	::=	⟨*Name*⟩	Category indicator. See class `handle`
⟨*Access*⟩	::=	`both | get | send | none`	Defines the access right to this variable
⟨*VarName*⟩	::=	⟨*Name*⟩	Name of variable used for delegation
⟨*Group*⟩	::=	⟨*Name*⟩	Functional group of the following methods or variables. Used to organise the ClassBrowser
⟨*SendSelector*⟩	::=	⟨*Name*⟩	Name of send-method to define
⟨*GetSelector*⟩	::=	⟨*Name*⟩	Name of get-method to define
⟨*Receiver*⟩	::=	⟨*Variable*⟩	Prolog variable bound to the receiver
⟨*Arg*⟩	::=	⟨*Variable*⟩	Prolog variable bound to argument
⟨*RVal*⟩	::=	⟨*Variable*⟩	Prolog variable that should be bound to the return value
⟨*AName*⟩	::=	⟨*Name*⟩	XPCE name for named argument
⟨*Type*⟩			See section 3.2.1 and section 7.5.1
⟨*PrologBody*⟩			Ordinary Prolog code
⟨*Value*⟩			Initial value for the instance variable. At this moment, only using constants is supported (int, name, bool)

Table 7.1: Syntax details for User Defined Classes

```
        send(D, slot, type, Type),
        send(D, display, new(T, text(Type))),
        send(T, name, type),
        ...

type(D, Type:type) :->
        "Modify the epistemological type"::
        send(D, slot, type, Type),
        get(D, member, type, Text),
        send(Text, string, Type).
```

object → slot: name, unchecked
object ← slot: name → unchecked
Read or write slot without side-effects. The value will be converted to the type of the instance variable addressed. An error is raised if this conversion is not defined or if the slot does not exist.

7.3 Refining and redefining methods

Re(de)fining methods is a common technique in object-oriented programming. This section describes how methods can be re(de)fined and what methods have special meaning in XPCE and are commonly redefined.

The method definition for a re(de)fined method is exactly the same as for a new method. The redefined method will inherit its group (see `pce_group/1`) from the method of the super-class.

When refining a method we often want to call the method of our super-class. For this reason there are two additional interface predicates to access the behaviour of a specific class. In 99% of the cases we wish to invoke a method of the immediate super-class. For this reason the class-compiler realises compile-time rewrite of `send_super/[2-12]` and `get_super/[3-13]` to `send_class/2` and `get_class/3`.

send_class(+*Object, +Class, +Message***)**
Invoke *Message* on *Object* using the implementation defined with class *Class*. *Class* must be the actual class of *Object* or one of its super-classes or an error is raised.

get_class(+*Object, +Class, +Message, -Result***)**
This is the get-equivalent of `send_class/3`.

send_super(+*Object, +Message***)**
The class-compiler converts goals of this format to an appropriate `send_class/3` call. Note that it is not possible to provide predicates as an alternative to the compile-time expansion and therefore meta-calls cannot use `send_super/2`.

get_super(+*Object, +Message, -Result***)**
This is the get-equivalent of `send_super/2`.

Similar as the predicates `send/2` and `get/3` may be written as `send/[3-12]` and `get/[4-13]` this is possible for `send_super/2` and `get_super/3`. In addition the pre-5.0 '`object → send_super`' and '`object ← get_super`' are expanded to `send_class/2` and `get_class/3`. The following calls are all equivalent. The last one should not be used by new code.

```
1         send_super(Object, my_method(Arg1))
2         send_super(Object, my_method, Arg1)
3         send(Object, send_super, my_method, Arg1)
```

7.3.1 General redefinitions

The most commonly redefined methods are →`initialise` and →`unlink` to redefine object creation and destruction. **Note that none of these methods should ever be invoked directly on an object**, because the implementation often makes assumptions that are only true in the context they are normally invoked by the kernel.

object → initialise: ⟨*Class-Defined*⟩

Initialise a new instance of the class. The initialisation is not allowed to access behaviour or slots of the super-class without invoking the →`initialise` on th super-class. Omitting is a common source of errors, often leading to crashes.

The initialise method should initialise all slots declared in this class that have no specified value in the variable declaration and cannot have the value `@nil`. See also `checkpce/0`.

If →`initialise` fails, the exception `initialise_failed` will be raised, passing the instance and the argument vector. Afterwards, the (possible named) reference is destroyed and the object's slots are reset to `@nil`. Finally, the instance is deallocated. →`unlink` (see below) is not called. In general, it is not good programming style to let →`initialise` fail.

object → unlink

Called from the object-management system if the object is to be destroyed. This method *must* call →`unlink` of the super-class somewhere in the process. It is an error if →`unlink` fails.

This method is normally used to unlink the object from related objects. For example, `graphical` objects use it to remove themselves from their device if they are displayed. There is no need to reset slot-values as dereferencing the slot-values will be done by the object-management system after →`unlink` has finished.

→`unlink` is always called, whether the object was destroyed using →`free` or by the garbage-collector.

object ← convert: ⟨*Class-Defined*⟩ **→ Instance**

This get method converts another object into an object of this class. It is called by the type-checker. Suppose an object *X* is handed to the type checker for checking against this class. If *X* is not already an instance of this class or any of its subclasses, the type checker will:

- Check *X* against the ⟨*Class-Defined*⟩ type.
- Run this method, passing the (possibly converted) *X*.

The receiver is not defined during the execution of this method. The method should either fail or succeed and return an instance of the requested class or one of its super-classes. The argument vector consists of a single argument. The type-conversion system guarantees the argument is of the satisfied type. It is allowed, but not obligatory to use the method of the super-class.

For example, suppose we are defining a class *person*, who has a unique name. There is a table `@persons`, that maps the name onto the person. We would like to be able to pass the name rather then a person instance to a method argument with the type `person`. If no such person exist, a new person instance is created. Below is the implementation for this:

```
convert(_, Name:name, P:person) :<-
        "Lookup from @persons or create a new one"::
        (   get(@persons, member, Name, P)
        ->  true
        ;   new(P, person(Name))
        ).
```

See also ←`lookup` described below.

object ← lookup: ⟨*Class-Defined*⟩ → **Instance**

Called from the new() virtual machine operation to deal with *reusable* objects before →`initialise` is considered. The arguments are normally the same as for →`initialise`. If this method returns an instance, this will be the value returned by new(). If it fails, a new instance is allocated and →`initialised`.

7.3.2 Redefinition in graphical classes

The generic graphical class `graphical` is prepared to have several of its methods redefined in subclasses. This section describes the most important of these methods.

graphical → event: event

Called when a user-event needs to be dispatched. This message is initially sent to the window object receiving the event. Graphical devices (and thus windows) collect all graphicals for which 'graphical →`in_event_area`' succeeds. These are normally all graphicals that overlap with the current position of the pointer. It will sort these objects to their stacking order, the topmost object first. See '`device ← pointed`'. Next the device will use '`event → post`' to post the event to each of these graphicals until one accepts the event, after which the method immediately returns success. If none of the ←`pointed` objects is prepared to accept the event, '`graphical → event`' will be invoked, trying all he `recogniser` objects associated with this graphical.

Notably most subclasses of class `dialog_item`, the standard controllers, refine →`event`.

The method →event is commonly redefined in user-defined graphicals to make them sensitive to the mouse. The following fragment of a class definition makes it possible to resize and move instances.

```
:- pce_global(@resize_and_move_recogniser,
              new(handler_group(new(resize_gesture),
                                new(move_gesture)))).

event(Gr, Ev:event) :->
        "Make the object re-sizeable and movable"::
        (   send_super(Gr, event, Ev)
        ;   send(@resize_and_move_recogniser, event, Ev)
        ).
```

Note that the implementation first tries the super-class. If the super-class has no specific event-handling, this allows recognisers to be attached that overrule the re-size/move behaviour. Also, if it is a device, invoking the super-class behaviour will test components displayed on the device to be considered before the device as a whole.

It is not obligatory to use →event on the super-class and if it is used, no specific ordering is required. If there is no behaviour of the super-class that conflicts with your extension we recommend to try the super-class first, to ensure recognisers and local event-processing in graphicals displayed on a device with redefined event-processing are considered before your extensions.

Note the way recognisers are activated from event methods. The graphical object itself is not passed. Instead, 'recogniser → event' reads the receiver from 'event ← receiver' set by 'event → post'.

As a consequence, do not call 'graphical → event' directly. An event is directed to a graphical using 'event → post'. For example, the event-method of a device displaying an editable text object may decide to forward all button and keyboard events to the text. The following accomplishes this:

```
event(D, Ev:event) :->
        (   (   send(Ev, is_a, button)
            ;   send(Ev, is_a, keyboard)
            )
        ->  % assumes text is named 'text'
            get(D, member, text, Text),
            send(Ev, post, Text)
        ;   send_super(D, event, Ev)
        ).
```

graphical → geometry: X:[int], Y:[int], W:[int], H:[int]

Requests the receiver to position itself at the X, Y and to be $W \times H$ pixels in size. Any of these values may be @default, indicating that the specific parameter is not to be changed.

Redefining →geometry is the proper way to interfere with positioning or resizing as this is the central method called by all move and resize methods.

The example below takes the text-box to ensure proper geometry handling by this class. Note that (I) the size of a device is by definition the bounding box of all displayed graphicals and (II) the text must be centered again.

```
geometry(D, X:[int], Y:[int], W:[int], H:[int]) :->
        get(D, member, box, B),
        get(D, member, text, T),
        send(B, set, @default, @default, W, H),
        send(T, center, B?center),
        send_super(D, geometry, X, Y).
```

Note that the relation between the text and the box could also be maintained using a constraint object. The above implementation however is only executed when the geometry of the device is changed, while constraints will be executed whenever a message arrives on the box or text.

graphical → request_geometry: X:[int], Y:[int], W:[int], H:[int]
Is much like →geometry, except that the interpretation of the units is left to the graphical. For example editor will use the current font to translate W and H to pixels and then invoke →geometry. Not used very often.

graphical → compute
This method cooperates with →request_compute and may be used to delay expensive graphical operations. Suppose we have a graphical representation and a database object linked using a hyper like this:

```
new(_, hyper(Db, Gr, controller, model))
```

If the database object (model) is modified, it could use the following to inform all associated controllers about the change:

```
send(Db, send_hyper, controller, request_compute)
```

XPCE remembers that the state of this graphical is not consistent. If XPCE requires the graphical to be in a consistent state, either because it needs to paint the graphical or because it requires information about the geometry of the graphical, it will invoke the method →compute on the graphical.

This mechanism is used by graphicals that have a complicated structure and are difficult to update. An example in the built-in classes is class text_image, displaying the text of an editor. Any modification to the text in the displayed region of the text_image requires expensive computation to recompute the layout of the text. Suppose the →request_compute and →compute mechanism is not available. It this case, multiple modifications by the program to the text would require this expensive process to run several times. Now, after modifying the text, →request_compute is

invoked on the text_image. Whenever XPCE has processed all pending events, it will invoke →compute to the text_image and then repaint it.

The method below is a far to simple example, where the →compute method simply copies the name of the represented object into the text object displayed on the device →compute is defined on.

```
compute(C) :->
        "Update according to model"::
        get(C, get_hyper, model, name, Name),
        get(C, member, text, T),
        send(T, string, Name),
        send_super(C, compute).
```

graphical → _redraw_area: area

Called by the graphical repaint thread. Its task is to repaint itself. *Area* indicates the area in the device coordinate system that needs to be repainted. This area overlaps with the ←area of the device.

Exploitation of this method to realise new graphical primitives is explained in section 10.12.

7.4 Handling default arguments

The predicate default/3 provides a comfortable way to specify the meaning of default arguments. Future versions may incorporate the default value in the type object itself.

default(+*Argument, +Default, -Value*)

Used to specify and compute defaults for arguments. *Argument* is the actual argument passed to the method implementation, *Default* is any valid XPCE object description (reference, integer, real, atom or compound ground term describing an object, see send/[2-12]). *Default* can also be the term

```
resource(<Object>, <Name>)
```

In which case the ←resource_value: ⟨*Name*⟩ from ⟨*Object*⟩ will be used as default value. *Value* is unified with *Argument* if *Argument* is not @default and with *Default* otherwise.

The following is an example that sets the volume to a specified value or the value of the resource 'volume' if @default is passed as an argument.

```
resource(volume, 0..130,  75, "Volume in decibels").

volume(X, Vol:[0..130]) :->
        default(Vol, resource(X, volume), V),
        <set the volume here>.
```

7.5 Advanced topics

7.5.1 More on type declarations

The basic XPCE type-syntax is described in section 3.2.1 of this manual. Types are first-class reusable XPCE objects that are created from the type-declaration in arguments and variables. The conversion from the textual representation to the object is performed by XPCE itself (together with the resource syntax, one of the few places where XPCE defines syntax). All types can be specified as Prolog quoted atoms. For example:

```
mymethod(Me, A:'graphical|dict_item|0..') :->
        ...
```

For most cases however, this is not necessary. If the type is not an atom, the class-compiler will convert the Prolog term into an atom suitable for XPCE's type system. Hence, `[point]` will translate to the atom '[point]', which is interpreted by XPCE as "an instance of class `point` or the constant `@default`". The atoms `*` and `...` are defined as postfix operators, while `..` is an infix operator. This makes '`any ...`' a valid notation for "any number of anything" (see section 7.5.2 below) and '`0..5`' a valid expression for "an integer in the range 0 to 5 (including the boundaries).

Also, `[box|circle]` is a valid description for "an instance of `box` or `circle` or the constant `@default`. Note however that `[box|circle|ellipse]` is *not* valid Prolog syntax and should be written as `'[box|circle|ellipse]'`. Whenever you are in doubt, use quotes to prevent surprises.

7.5.2 Methods with variable number of arguments

Methods such as '`chain` → `initialise`' and '`string` → `format`' handle an arbitrary number of arguments. The argument declaration for such a method first defines a number (possibly zero) of 'normal' arguments. The last argument is postfixed with '`...`'. The arguments assigned to the 'vararg' type are passed in a Prolog list.

Below is a refinement of '`label` → `report`' that will colour the label depending on the nature of the message. The →`report` method takes two obligatory arguments, the *kind* of the report and a *format* string, as well as an undefined number of arguments required by the format specification.

```
1  :- pce_begin_class(coloured_reporter, label,
2                     "Coloured reporter label").
3
4  report(L, Kind:name, Format:char_array, Args:any ...) :->
5          Msg =.. [report, Kind, Format | Args],
6          send_super(L, Msg),
7          get(L, colour_from_report_category, Kind, Colour),
8          send(L, colour, Colour).
9
10 colour_from_report_category(L, Kind:name, Colour:colour) :<-
11         <left to the user>.
12
13 :- pce_end_class.
```

Using class templates

XPCE provides two alternatives to multiple inheritance. Delegation is discussed in section C.4. See also the directive `delegate_to/1` for user-defined class definitions. The *template* mechanism is much closer to real multiple inheritance. A template is a named partial class-definition that may be included in other classes. It behaves as if the source-code of the template definition was literally included at the place of the `use_class_template/1` directive.

In fact, the class-attributes (variables, method objects) are *copied*, while the implementation (the Prolog clauses) are *shared* between multiple references to the same template.

Templates itself form a hierarchy below class `template`, which is an immediate subclass of `object`. Including a template will make all variables and methods defined between the template class and class `template` available to the receiving class.

We illustrate the example below, making both editable boxes as editable ellipses. First we define the template class.

```
:- use_module(library(pce_template)).

:- pce_begin_class(editable_graphical, template).

:- pce_global(@editable_graphical_recogniser,
              make_editable_graphical_recogniser).

make_editable_graphical_recogniser(G) :-
        Gr = @arg1,
        new(Dev, Gr?device),
        new(P, popup),
        send_list(P, append,
                  [ menu_item(cut, message(Gr, free)),
                    menu_item(duplicate,
                              message(Dev, display, Gr?clone,
                                      ?(Gr?position, plus,
                                        point(10,10))))
                  ]),
        new(G, handler_group(new(resize_gesture),
                             new(move_gesture),
                             popup_gesture(P))).


event(G, Ev:event) :->
        (   send_super(G, event, Ev)
        ;   send(@editable_graphical_recogniser, event, Ev)
        ).
:- pce_end_class.
```

The main program can now be defined as:

```
:- require([use_class_template/1]).

:- pce_begin_class(editable_box, box).
:- use_class_template(editable_graphical).
:- pce_end_class.
```

```
:- pce_begin_class(editable_ellipse, ellipse).
:- use_class_template(editable_graphical).
:- pce_end_class.

test :-
        send(new(P, picture('Template Demo')), open),
        send(P, display,
             editable_box(100,50), point(20,20)),
        send(P, display,
             editable_ellipse(100, 50), point(20, 90)).
```

Note that `use_class_template/1` *imports* the definitions from the template in the current class. Thus, the following **will not extend** further on the '`editable_graphical` → `event`' definition, but instead **replace** this definition. Of course it is allowed to subclass the definition of editable_box above and further refine the event method in the subclass.

```
:- require([use_class_template/1]).

:- pce_begin_class(editable_box, box).
:- use_class_template(editable_graphical).

event(Gr, Ev:event) :->
        (   send_super(Gr, event, Ev)
        ;   ...
        ).
:- pce_end_class.
```

7.5.3 Implementation notes

The XPCE/Prolog class-compilation is defined using the Prolog preprocessing capabilities of `term_expansion/2`. While the class is compiled, Prolog gathers the expressions belonging to the class. The expansion of `:- pce_end_class(`*`Class`*`)` emits the actual code for the class.

The method implementation is realised by the predicates pce_principal:`send_implementation/3` and pce_principal:`get_implementation/4`. that take the form:

send_implementation(*MethodId, Method(Arg...), Object***)**
: Where *MethodId* is unique identifier for the class and method, *Method* is the method implemented, *Arg...* are the arguments accepted by the method and *Object* is the receiving object.

get_implementation(*MethodId, Method(Arg...), Object, -Result***)**
: This is the get-equivalent of `send_implementation/3`.

```
:- pce_begin_class(gnus, ...
gnu(X, A:int) :-> ...
gnats(X, A:name, B:int) :-> ...
```

is translated into

```
pce_principal:send_implementation('gnus$+$->gnu', gnu(A), O) :- ...
pce_principal:send_implementation('gnats$+$->gnu', gnats(A, B), O) :-
...
```

The remainder of the class specification is translated into a number of Prolog clauses describing the class. No XPCE class is created. If XPCE generates an `undefined_class` exception, it will scan for the class-description in the Prolog database and create the XPCE `class` instance. No methods are associated with the new class. Instead, all method binding is again based on exception handling.

Modifications to the class beyond what is provided by the preprocessing facilities (for example changing the '`variable → clone_style`') cannot be made by sending messages to the class inside the class definition body as this would address the not-yet-existing class. Instead, they should be embedded in the `pce_class_directive/1` directive.[3]. The *Goal* argument of `pce_class_directive/1` should refer to the class using the XPCE `var` object `@class`. When the class is realised the exception system will bind `@class` to the current class while running *Goal*. *Goal* is called from the Prolog module in which the class is defined.

The main reason for the above approach is to exploit the runtime-generation facilities of the hosting Prolog system to create fast-starting portable and (depending on the hosting Prolog's capabilities) stand-alone executables.

One of the consequences of the chosen approach is that the class-building directives are not accessible as predicates. There is no preprocessing support for the dynamic creation of classes and the programmer should thus fall back to raw manipulation of the XPCE class objects.

[3] To facilitate the translation of old code, the construct `:- send(@class, ...` is treated automatically as if it was embedded using `pce_class_directive/1`

8 Class Variables

Class variables act as read-only storage for class-constants. They are normally used for storing setting information, such as fonts, colours etc. For this reason, the default value for a `class_variable` is defined with the declaration of it, but this default my be overruled using the `Defaults` file. The system defaults file is located in the XPCE home directory (‘`@pce ← home`’). This file contains an *include* statement, including the file `~/.xpce/Defaults`,[1] which may be used by the developer and application user to specify defaults.

Many XPCE built-in classes define class-variables. These can be examined using the *ClassBrowser* (see section 3.3.1) from the online manual tools.

8.1 Accessing Class Variables

Class variables define *get*-behaviour and can be accessed using the normal `get/[3-13]` call. Class variables are the last type of behaviour checked when resolving a get-method. Below are the most commonly used methods to access class-variables.

object ← class_variable_value: name → any
: Return the value of the named class-variable. Fails silently if the class does not define the given class-variable.

class ← class_variable: name → class_variable
: Return the `class_variable` object with the given name. Fails silently if the class does not define the given class-variable.

class_variable ⇆ value: any
: Reads or writes the class-variable value. The argument is type-checked using ‘`class_variable ← type`’ if the value is written. Writing class-variables should be handled with care, as existing instances of the class are not notified of the change, and may not be prepared deal with changes of the class-variable value. `pce_image_directory/1` is an example of a predicate modifying the image.path class-variable.

8.2 Class variable and instance variables

Class-variables may be used as defaults for instance-variables that can be modified through the `Defaults` file. For example, class `text` defines both the instance- and class-variable

[1] On Windows systems, ~ expands to `\%HOME\%`, `\%USERPROFILE\%`, `\%HOMEDRIVE\%/\%HOMEPATH\%` or the root of the current drive. See `expand_file_name/2` of the SWI-Prolog manual.

font. The class-variable defines the default font, while the font can be modified explicitely at instance-initialisation, or using the →`font` method afterwards.

If a class defines both an instance- and a class-variable with the same name, object-allocation will use the `constant @class_default` for the initial value of the slot. First access will replace this value using the value from the class-variable.

Instance-variables supercede class-variables for *get*-behaviour. To access the class-variable explicitly, use the methods from section 8.1.

8.3 The 'Defaults' file

The `Defaults` file consists of statements. Each statement is on a separate line. A statement may be split over several lines by preceeding the newline with a backslash (\). The exclamation mark (`!`) is the line-comment character. All input from the `!` upto the following newline is replaced by a single space-character. Empty lines or lines containing only blank space are ignored.

Default files may include other default files using the statement

`#include` ⟨*file*⟩

Default statements are of the form:

⟨*class*⟩.⟨*class-variable*⟩: ⟨*value*⟩

Where ⟨*class*⟩ denotes the name of the class, or * to indicate the default applies for any class defining this class-variable. If both forms are found, the statement with the explicit class-name is used. ⟨*class-variable*⟩ denotes the class-variable name. ⟨*value*⟩ is the default value of the class-variable. The syntax for ⟨*value*⟩ is similar to the arguments of `send/[2-12]`. The (informal) syntax for ⟨*value*⟩ is below.

⟨*Any*⟩	::=	⟨*int*⟩
	|	⟨*float*⟩
	|	⟨*Name*⟩
	|	@⟨*Name*⟩
	|	⟨*Chain*⟩
	|	⟨*Object*⟩
⟨*Chain*⟩	::=	[⟨*Any*⟩ {, ⟨*Any*⟩}]
	|	[⟨*Blank*⟩]
⟨*Object*⟩	::=	⟨*ClassName*⟩( )
	|	⟨*ClassName*⟩(⟨*Any*⟩ {, ⟨*Any*⟩})
	|	⟨*PrefixOp*⟩ ⟨*Any*⟩
	|	⟨*Any*⟩ ⟨*InfixOp*⟩ ⟨*Any*⟩
	|	⟨*Any*⟩ ⟨*PostfixOp*⟩
	|	" ⟨*String*⟩ "
⟨*String*⟩	::=	{⟨*CharNotDoubleQuote*⟩|""}
⟨*Name*⟩	::=	⟨*Letter*⟩{⟨*Letter*⟩|⟨*Digit*⟩}
	|	' {⟨*CharNotSingleQuote*⟩|''} '

8.4 Class variables in User Defined Classes

Class-variables are declared, similar to instance-variables. through macro-expansion inside the :- `pce_begin_class/[2,3]` ...:- `pce_end_class/0` definition of a class. The syntax is:

`class_variable(`⟨*name*⟩`,` ⟨*type*⟩`,` ⟨*default*⟩`,` [⟨*summary*⟩]`).`

⟨*default*⟩ defines the value if not overruled in the `Defaults` file. It is a Prolog term describing an object similar to the arguments of `send/[2-12]`.

In the following example. there is a class with the property 'expert_level'. The program defines the default level to be novice. The user may change that in his/her personal `Defaults` file or change it while the application is running. As the value may change at runtime, there should be an instance- as well as a class-variable. Below is the skeleton for this code:

```
variable(expert_level, {novice,advanced,expert}, get,
         "Experience level of the user").

class_variable(expert_level, @default, novice).

expert_level(Obj, Level:{novice,advanced,expert}) :->
        send(Obj, slot, expert_level, Level),
        <handle changes>.

        ...,
        (   get(Obj, expert_level, expert)
        ->  ...
        ;   ...
        ),
        ...
```

9 Program resources

Resources, in the new sense of the word is data that is required by an application but cannot be expressed easily as program-code. Examples are image-files, help-files, and other files using non-Prolog syntax. Such files are declared by defining clauses for the predicate `resource/3`:

resource(*?Name, ?Class, ?PathSpec***)**
Define the file specified by *PathSpec* to contain data for the resource named *Name* of resource-class *Class*.

Name refers to the logical name of the resource, which is interpreted locally for a Prolog module. Declarations in the module `user` are visible as defaults from all other modules. *Class* defines the type of object to be expected in the file. Right now, they provide an additional name-space for resources. *PathSpec* is a file specification as acceptable to `absolute_file_name/[2,3]`.

Resources can be handled both from Prolog as for XPCE. From Prolog, this is achieved using `open_resource/3`:

open_resource(*+Name, ?Class, -Stream***)**
Opens the resource specified by *Name* and *Class*. If the latter is a variable, it will be unified to the class of the first resource found that has the specified *Name*. If successful, *Stream* becomes a handle to a binary input stream, providing access to the content of the resource.

The predicate `open_resource/3` first checks `resource/3`. If successful it will open the returned resource source-file. Otherwise it will look in the programs resource database. When creating a saved-state, the system saves the resource contents into the resource archive, but does not save the resource clauses.

This way, the development environment uses the files (and modifications to the `resource/3` declarations and/or files containing resource info thus immediately affect the running environment, while the runtime system quickly accesses the system resources.

From XPCE, resources are accessed using the class `resource`, which is located next to `file` below the common data-representation class `source_sink`. Many of the methods that require data accept instances of `source_sink`, making resources a suitable candidate.

Below is the preferred way to specify and use an icon.

```
resource(my_icon,       image,  image('my_icon.xpm')).
```

```
...,
send(Button, label, image(resource(my_icon))),
...,
```

The directive `pce_image_directory/1` adds the provided directory to the search-path for images (represented in the class-variable `image.path`), as well as to the `image/1` definition of `file_search_path/2`.

Please note that MS-Windows formatted image files can currently not be loaded through `resource` objects. The Windows API only provides functions to extract these objects from a single file, or nested as Windows resources in a `.dll` or `.exe` file.

Right now, it is advised to translate the images into `.xpm` format using the following simple command:

```
?- send(image('myicon.ico'), save, 'myicon,xpm', xpm).
```

This transformation is complete as the `.XPM` image format covers all aspects of the Microsoft image formats. For further details on image formats and manipulation, see section 10.10.

10 Programming techniques

This chapter is an assorted collection of techniques to programming problems that appear in many applications and for which support for generic solutions is present, but cannot easily be found using the online reference manual.

10.1 **Control-structure of PCE/Prolog applications**
Discusses the event-driven control-structure used by XPCE applications. Also describes how to avoid using this structure.

10.2 **Executable Objects**
Discusses XPCE code objects and their usage in controls, methods and as parameters.

10.3 **Defining global named objects**
Discusses handling of globally named system-wide objects.

10.4 **Using object references: "Who's Who?"**
Discusses mechanism available to find object references, avoiding global references and explicit storage of references.

10.5 **Relating frames**
Discusses combining frames in a application, transient frames and various styles of modal operation and event handling.

10.7 **Informing the user**
Discusses the generic reporting system used to inform the user of progress, status and errors.

10.8 **Errors**
Specifying, handling and generating errors.

10.9 **Specifying fonts**
Discusses the indirections in the font-specification mechanism.

10.10 **Using images and cursors**
Discusses images, icons and cursors.

10.11 **Using Hyper Links to Relate Objects**
Discusses a technique to relate objects.

10.12 **User Defined Graphicals**
Discusses the definition of new graphical primitives.

10.13 **Printing**
Discusses printing from XPCE applications.

10.1 Control-structure of XPCE/Prolog applications

This section deals with the control-structure of interactive applications written in XPCE/Prolog. Interactive graphical applications are very different from terminal oriented applications. Terminal oriented applications often have a top level control structure of the form:

```
go :-
        initialise,
        main_loop.

main_loop :-
        present_question,
        read_answer(Answer),
        process_answer(Answer),
        main_loop.
```

This schema is often refined with sub-loops dealing with question/answers in a specific context.

Many interactive graphical applications present various UI components simultaneously: the user is free on which component s/he wants to operate next. The users actions (keyboard-typing, mouse movement, and mouse-clicking) must be related to the correct UI component and interpreted accordingly in the application. This interpretation is much more complex than the interpretation of a stream of ASCII characters typed by the user.

10.1.1 Event-driven applications

One approach is to write a main-loop that reads events, locates the UI-component referred to and executes the appropriate actions. This loop, which must take care of repaint-requests, various local feedback procedures (changing the mouse-cursor, inverting objects, etc.), is complicated. The approach taken by most graphical programming systems including XPCE, is to move this loop into the infra-structure (i.e. into the XPCE kernel). The application programmer creates the desired UI-components and supplies code fragments that will be called by the main-loop when a certain event happens. This control-structure is called *event-driven* control. Consider a button:

```
1 ?- new(B, button(hello,
                   message(@pce, write_ln, hello))),
     send(B, open).
```

In this example the application creates and displays a button UI component and associates a code fragment (the message) to be executed when the button is pressed. The XPCE kernel will loop through the main event-loop. For each event it will locate the UI component that should handle the event. When the button has recognised a ‘click’ it will execute the code fragment attached to it. This behaviour is part of the definition of class `button`.

It is clear that this approach relieves the application programmer of many of the complications associated with event-processing. As a consequence, the ‘main-loop’ of a XPCE application is no longer in the application itself, but in the XPCE kernel. Below is an outline of the control structure of a XPCE/Prolog application:

```
go :-
        initialise_database,
        create_ui_components.

handle_help_pressed :-
        create_help_window.

handle_solve :-
        solve_the_problem,
        create_solution_window.

...
```

The predicate go will exit after it has initialised the application and created the UI components. Assuming the application window has a button invoking the predicate handle_help_pressed, XPCE will call this predicate when the user presses the help button.

Keeping control

The application code often wants to wait for the user to finish an interaction. In section 4.4, we have seen a simple way of programming this using '`frame ← confirm`'. In this section, we will provide some other options.

Message Queue One possibility is to fall back to the XPCE 1 and 2 compatibility, where `@prolog` implements a queue of messages. `@prolog` is an instance of class `host`. The relevant methods are:

host → call_back: bool
: The default is `@on`. In this case, a message to `@prolog` is translated into a predicate call on the Prolog engine. If `@off`, a message is appended to the '`host ← messages`' queue.

host → catch_all: Selector:name, Arg:any...
: If ←`call_back` equals `@on`, use the *Selector* to determine the predicate to call, and the arguments to construct the argument vector for the predicate. Call the predicate and succeed or fail according to success or failure of the Prolog predicate.

  If ←`call_back` equals `@off`, create a `message` of the form

  message(`@prolog`, *Selector*, *Arg* ...)

  and append this message to the ←`messages` queue.

host ← message→ message
: Return the ←`head` of the ←`messages` queue. If the queue is empty, ensure ←`call_back` is (temporary) set to `@off`, and dispatch events using '`@display → dispatch`' as long as the ←`messages` queue is empty.

Note that it is possible to create multiple instances of class `host`, to realise multiple message queues. It is not desirable to modify the `@prolog` host object, as other code may rely on the ←`call_back` properties of `@prolog`.

> **Warning** During normal operation, event processing guards the objects created that are not assigned to any other object and destroys all such objects after the event has completely been processed (see section E. Using the host message queue mechanism, the Prolog programmer becomes responsible for such objects. For example, the message object returned should be discarded using '`object` → `done`' after processing.

Explicit dispatching An alternative to the above, and the '`frame` ← `confirm`' mechanism is to dispatch the events yourself. This is realised using `send(@display, dispatch)`, described below. This mechanism is the base of all the others. It should be used to realise different interaction schemas than the default callback schema.

display → dispatch
Process events and return on any of the following conditions

- *event has been processed*
  Either a normal event, a timer or an input stream has been processed. The method fails in this case.
- *timeout*
  The timeout (see '`display_manager` → `dispatch`') has expired. The method fails in this case.
- *Input on the console*
  There is input from the Prolog window. The message succeeds in this case.

For example, the following processes events in call-back style until the fact `quit/0` is in the Prolog database:

```
1  :- dynamic
2          quit/0.
3
4  process_to_quit :-
5          repeat,
6              send(@display, dispatch),
7          quit, !.
```

10.1.2 XPCE and existing applications

Due to the different control-regime described in the previous section, traditional terminal oriented applications are not easily transformed into XPCE/Prolog graphical applications. Depending on the application, there are two ways to proceed.

The first is to keep the existing control-regime. This implies that the questions asked on the terminal will be replaced by modal dialog windows. The main loop will be:

```
go :-
        initialise_database,
        create_dialog(Dialog).

main_loop(Dialog) :-
        fill_dialog_with_next_question(Dialog),
        send(Dialog, fit),
        get(Dialog, confirm, Answer),
        process_answer(Answer),
        main_loop(Dialog).
```

This example reuses the same dialog window for all questions. It is trivial to change this loop to use a new dialog window for each question. Output from the program may be presented in other windows. The approach does not exploit the potentially larger freedom for the user that is possible in graphical user interfaces.

If the application could be viewed as a number of commands operating on some data-structure and this data-structure is stored on the Prolog heap using `assert/1` or `recorda/2` one could consider rewriting the toplevel control and provide a more flexible interface.

10.2 Executable objects

PCE defines *executable* objects. Executable objects (called code objects) can be compared to lambda functions in Lisp. They are used in the contexts given below.

- Definition of 'call-back' actions
  The most common usage of executable objects is to specify the action undertaken by interactive UI components when they are activated. We have seen examples of this in section 2.8.

- As parameter to a method
  Various methods take an executable object as a parameter to specify the action undertaken. Suppose we want to remove all graphicals in the selection of a graphical device. The method '`device` ← `selection`' returns a `chain` object containing all graphicals currently selected on the device. The implementation is:

  ```
  ...,
  send(Dev?graphicals, for_all, message(@arg1, free)).
  ...
  ```

  The method '`chain` → `for_all`' will execute the argument `message` object in a loop for all elements in the chain. It will bind `@arg1` to the currently processed element of the chain.

Two major groups of code objects are distinguished: *procedures* and *functions*. The first only returns status information similar to the send-operation. The latter returns a value or failure status information similar to the get-operation.

10.2.1 Procedures

Procedures perform an action when executed. Similar to Prolog, the execution of a procedure either returns successful or failed completion. Procedures are normally executed by the object or method they are handed to. For example, the procedure associated to a `button` object is executed when the button is pressed. The procedure handed to the '`chain` → `for_all`' method we have seen above is executed by this method for each element of the chain.

Procedures can also be executed explicitly using the method '`code` → `forward`'. This binds the argument-forwarding objects `@arg1`, ..., and then executes the procedure and restores the old bindings of argument-forwarding objects. For example:

```
1 ?- new(@m, message(@prolog, format, 'Hello ~w~n', @arg1)).
2 ?- send(@m, forward, world).
Hello world
```

- message(Receiver, Selector, Argument ...)
  A `message` object starts a send-operation when executed. The arguments of a message are either objects or *functions*. In the latter case these are evaluated before the

message itself is evaluated. If the evaluation of one of the argument functions fails, the message is not executed. The *Receiver* of a the message can be `@prolog` to invoke a predicate in the Prolog environment.

- and(Statement, ...)
  An `and` is a sequence of code objects. It fails if one of the members fails and succeeds otherwise. If a statement is a function, it will be evaluated. Functions fail only if they return the fail control-value. Notably, a function that returned the boolean `@off` (false) is considered to have succeeded.
- if(Condition, [Then], [Else])
  An `if` implements a branch. It first evaluates *Condition* and then either of *Then* or *Else*. The success of the entire if is determined by the success of the executed *Then* or *Else*. Either or both of these statements may be omitted. The construct `if(`*`Statement`*`)` may be used to force success of a code object.
- Conditions: (A == B, A \== B, A < B, ...)
  These executable objects are commonly used as conditions for the `if` object.

The online manual may be used to get an overview of the other available code objects. See section 3.3.4.

10.2.2 Functions

Functions are code objects which —when executed— evaluate to a value. They are commonly used as arguments to other code objects or as arguments to any method. A function is automatically evaluated iff:

- It appears as part of a code object that is executed
- Type checking demands execution

The most important function objects are:

- ?(Receiver, Selector, Argument, ...)
  Class `?` starts a get-operation when executed (= evaluated). The classname is pronounced as *obtainer*. All the arguments may be functions. These will be evaluated before the get-operation is performed. As with `message`, a failing evaluation of one of the arguments forces failure of the whole.

- var
  Class `var` describes a variable. The objects `@arg1`, `@arg2`, ..., `@arg10`, `@receiver` and `@event` are the most commonly used var objects.

  The objects `@arg1`, ... are the message *forwarding* arguments. Assume the following message.

  ```
  ?- new(@m, message(@pce, format,
                     'Hello %s\n', @arg1)).
  ```

  When executed, `@arg1` (a `function`) is first evaluated. The method '`code` → `forward`: argument, ...' binds these `var` objects locally:

```
?- send(@m, execute),
   send(@m, forward, world),
   send(@m, execute).
@default
world
@default
```

The objects `@receiver` and `@event` are functions referring to the receiver of the currently executing method and the current user-event. Var objects are rarely created explicitly by application programmers.

- arithmetic functions (+, -, *, /)
  These functions define *integer* arithmetic and are commonly used to deal with graphical computations in code objects.

- create(Class, InitArg, ...)
  This function evaluates to a new instance of *Class* from the given *InitArg* initialisation arguments. Its usage is best explained using an example. Consider we want to iterate over all boxes in a graphical device, writing a file with lines of the form

  ```
  BOX <name> at X, Y
  ```

  A naive approach and common beginners mistake is to write:

  ```
  ...,
  new(F, file(Output)),
  send(F, open, write),
  send(Dev?graphicals, for_all,
       if(message(@arg1, instance_of, box),
          message(F, append,
                  string('BOX %s at %d,%d\n',
                         @arg1?name,
                         @arg1?x,
                         @arg1?y)))),
  send(F, close),
  ...
  ```

  This example will yield a warning:

  ```
  [PCE error: get: No implementation for:
                        @default/constant <-x
          in: get(@default/constant, x)]
  PCE:  4 fail: get(@default/constant, x) ?
  ```

  The reason for this is that the *interface* will translate string(...) into an instance of class `string`, instead of the execution of the →`for_all` method's body creating different strings for each box displayed on the device. The correct solution is to use:

```
...,
new(F, file(Output)),
send(F, open, write),
send(Dev?graphicals, for_all,
     if(message(@arg1, instance_of, box),
        message(F, append,
                create(string,
                       'BOX %s at %d,%d\n',
                       @arg1?name,
                       @arg1?x,
                       @arg1?y)))),
send(F, close),
...
```

The construct 'create(...)' will create an instance of class `create`. '`message` → `execute`' will first evaluate all functions in the message and thus create an instance of class `string`.

10.2.3 Example 1: Finding objects

A common problem is to find objects, notably some specific graphical object on a window. If we want to find all instances of class `box` within a graphical `device`, we can use the call below, collecting all boxes on the device in a new `chain`.

```
...
get(Dev?graphicals, find_all.
    message(@arg1, instance_of, box),
    Boxes),
...
```

10.2.4 Example 2: Internal behaviour of dialog window

Code are most commonly used to specify the internal behaviour of dialog windows, such that the call-back predicate can concentrate on the real function. We have seen an example of this in section 2.8.

Below there is another example. *Data* is assumed to be an XPCE `string` object containing a PostScript^{tm} description of a graphical object as can be obtained using

```
...,
get(Graphical, postscript, PostScriptString),
...,
```

In this example both the internal dynamics of the dialog window (the label of the text-entry fields changes if the user switches from file to printer) and grabbing the arguments from the various dialog items is written using XPCE executable objects. Prolog will only be called to do the real work: printing the data to the requested destination.

Note that XPCE/Prolog does not **require** you to use XPCE executable objects this way. It is also possible to call Prolog from both the menu and the buttons, passing the dialog window as argument and write all behaviour in Prolog. We leave this as an exercise to the user.

```
postscript(Data) :-
        new(D, dialog('Print destination')),
        send(D, append, new(T, menu(destination, marked))),
        send_list(T, append, [printer, file]),
        send(T, layout, horizontal),
        send(D, append,
             new(A, text_item(printer_name, 'PostScript'))),
        send(T, message,
             if(T?selection == printer,
                message(A, label,
                        ?(A, label_name, printer_name)),
                message(A, label,
                        ?(A, label_name, file_name)))),
        send(D, append,
             button(ok, and(message(@prolog,
                                    print_postscript,
                                    T?selection,
                                    A?selection,
                                    Data),
                            message(D, destroy)))),
        send(D, append,
             button(cancel, message(D, destroy))),
        send(D, default_button, ok),
        send(D, open).

print_postscript(printer, Address, Data) :- !,
        new(F, file),
        send(F, open, write),
        send(F, append, Data),
        send(F, close),
        get(F, name, TmpFile),
        get(string('lpr -P%s %s', Address, TmpFile),
            value, Command),
        unix(shell(Command)),
        send(F, remove).
print_postscript(file, Address, Data) :-
        new(F, file(Address)),
        send(F, open, write),
        send(F, append, Data),
        send(F, close).
```

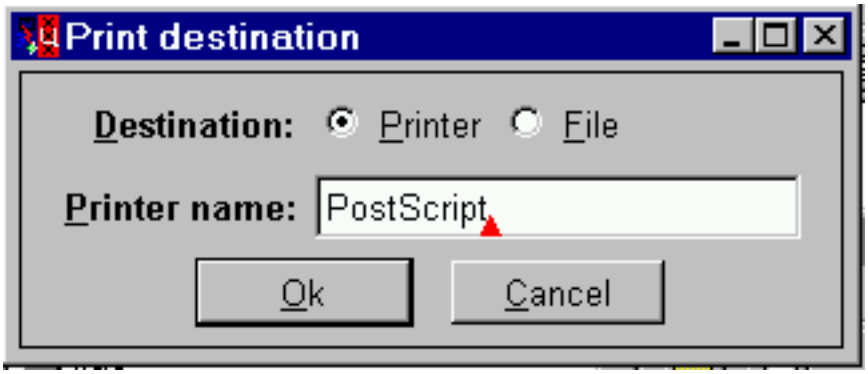


Figure 10.1: Print destination dialog using code objects for internal behaviour

10.3 Defining global named objects

As explained before, named references should be restricted to debugging and reusable objects. A couple of problems related to their creation and usage can be identified:

- Creating
  Objects need to be created before they can be used. Reusable objects generally are used from various places in the application. How are they best declared and when are they best created?
- Versioning
  Symbolic programming languages generally allow the programmer to change the program when it is running. This property makes them suitable for rapid-prototyping. Global objects are created from the Prolog system. It is desirable that the global object changes if the source-code that declares it has been changed.

Various alternatives to creating global objects have been tried. We will present some of the straight-forward approaches below and describe the (dis)advantages. Section 10.3.3 describes the `pce_global/2` directive to solve this problem. We will use a particular fill-pattern (image) as an example.

10.3.1 Using directives

Using a directive to create a reusable global object appears to be a logical way of dealing with them. This leads to the following code:

```
:- new(@stones_image, image(stones)).

        ...,
        send(Box, fill_pattern, @stones_image),
        ...
```

This code has a serious disadvantage. Whenever this file is reloaded after the Prolog code inside it has changed, the directive will be executed as well. The predicate `new/2` will generate a warning on an attempt to create an object with an existing name.

10.3.2 Inline testing

Another common approach is to test inline. For the example, this would look like:

```
        ...,
        (   object(@stones_image)
        ->  true
        ;   new(@stones_image, image(stones))
        ),
        send(Box, fill_pattern, @stones_image),
        ...
```

This approach is bad programming style. If `@stones_bitmap` is required at various places in the source files this construct needs to be repeated at various places.

10.3.3 The 'pce_global' directive

This approach is based on exception-handling. If PCE translates a named reference into an internal reference and the named reference does not exist it will raise an exception. The `pce_global/2` directive installs an exception handler dealing with a specific global reference. The example becomes:

```
:- pce_global(@stones_image, new(image(stones))).

        ...,
        send(Box, fill_pattern, @stones_image),
        ...
```

This directive applies some heuristics to take care of redefinitions when the file is reconsulted: if the definition is still the same it will not complain. If the definition has modified and the object is already created it will rename the object using 'object →`name_reference`'. A later reference to the object will trap the exception handler again, creating a new object according to the current specification. The directive prints diagnostics messages on redefinitions and other possible problems during compilation. See appendix D for details on `pce_global/2`.

10.3.4 Global objects for recognisers

Recogniser objects (see section 5.5) that make graphical objects sensitive to mouse events are often created with a global reference. Suppose an application requires box objects that can be moved, resized and that have a popup menu. The recogniser may be defined as:

```
:- pce_global(@mybox_recogniser, make_mybox_recogniser).

make_mybox_recogniser(R) :-
        Gr = @arg1,
        new(P, popup),
        send_list(P, append,
                  [ menu_item(cut, message(Gr, free))
                    ...
                  ]),
        new(R, handler_group(new(resize_gesture),
                             new(move_gesture),
                             popup_gesture(P))).
```

This recogniser object may be attached to the graphical objects either using '`graphical → recogniser`' or by redefining the '`graphical → event`' method. In the first case, the recogniser is an integral part of the graphical object and cannot be changed easily during the lifetime of the object. In the second case, the reference to the gesture is through the Prolog implementation of the method and replacing the global object will make all members of the class behave according to the new definition immediately.

If the latter approach is taken and the recogniser is under development, you may wish to use `free/1` to make sure the new definition is created:

```
:- free(@mybox_recogniser).
:- pce_global(@mybox_recogniser, make_mybox_recogniser).
```

10.4 Using object references: “Who’s Who?”

A user interface generally consists of a large amount of UI components. Some of these are used as input devices and some as output devices. Input devices generally activate functionality in the host system. Output devices are addressed by the host system to present results. Both input- and output devices may be related to entities within the application. For example, a particular icon may be the visualisation of a file in the computer’s file-system.

The application must be able to find the references to these UI components. Various techniques are available to keep track of objects in the user interface. Below we will discuss the following case:

> We want to create a frame, consisting of a dialog window and a picture window. The dialog contains a menu holding fill-patterns. The picture contains a box with a popup-menu that fills the interior of the box with the currently selected fill-pattern.

To reduce the code of the individual examples, the following predicate creating the fill-pattern menu is assumed to be available:

```
fill_pattern(@white_image).
fill_pattern(@grey12_image).
fill_pattern(@grey25_image).
fill_pattern(@grey50_image).
fill_pattern(@grey75_image).
fill_pattern(@black_image).

make_fill_pattern_menu(M) :-
        new(M, menu(fill_pattern, marked)),
        send(M, layout, horizontal),
        forall(fill_pattern(P),
               send(M, append, menu_item(P, @default, P))).
```

10.4.1 Global named references

Using this approach, we will call the menu `@fill_pattern_menu`. It leads to the following (minimal) program:

```
fill_1(P) :-
        new(D, dialog('Fill 1')),
        make_fill_pattern_menu(@fill_pattern_menu),
        send(D, append, @fill_pattern_menu),
        send(new(P, picture), below, D),
        send(D, open).

add_box_1(P) :-
        send(P, display, new(B, box(100,100)), point(20,20)),
        send(B, popup, new(Pop, popup)),
        send(Pop, append,
             menu_item(fill,
                       message(B, fill_pattern,
                               @fill_pattern_menu?selection))).
```

```
1 ?-    fill_1(P),
        add_box_1(P).
```

This approach is straightforward. Unfortunately it has various serious disadvantages:

- Name conflicts
  Large applications will have many objects whose references needs to be available. Unless the application provides a structure that can be used to generate meaningful names, one is likely to run into name conflicts quickly.
- Object life-time
  With named references, the application is responsible for destruction of the object. Thus, if a window holding named objects is freed, the named objects will not be destroyed. This has to be done explicitly. See also appendix E.
- No multiple instances
  The code above cannot be called more than once to create more than one such frame.

Global references are part of PCE to keep track of objects that are created once and will remain in existence during the entire PCE session. Examples are the predefined global objects `@pce`, `@prolog`, `@display`, etc. Other examples are reusable objects such as relations, messages, recognisers, methods, images to be used as fill-patterns, etc. See below:

```
:- pce_global(@center,
              new(spatial(xref=x+w/2, yref=y+h/2,
                          xref=x+w/2, yref=y+h/2)).
:- pce_global(@move_outline,
              new(move_outline_gesture)).
```

10.4.2 Using the prolog database

Dynamic predicates form another technique often used by novice PCE users. Using dynamic predicates the “label” would result in:

```
1  :- dynamic
2          fill_pattern_menu/1.
3
4  fill_2(P) :-
5          new(D, dialog('Fill 2')),
6          make_fill_pattern_menu(M),
7          send(D, append, M),
8          asserta(fill_pattern_menu(M)),
9          send(new(P, picture), below, D),
10         send(D, open).
11
12 add_box_2(P) :-
13         send(P, display, new(B, box(100,100)), point(20,20)),
14         send(B, popup, new(Pop, popup)),
```

```
        fill_pattern_menu(M),
        send(Pop, append,
             menu_item(fill,
                       message(B, fill_pattern,
                               M?selection))).
```

```
1 ?-    fill_2(P),
        add_box_2(P).
```

This is not a proper way to deal with references either. First of all, it does not really solve the danger of name conflicts unless one is using Prolog modules to establish storage of the dynamic predicates local to the module that uses them. More seriously, using implicit object references, PCE assumes it is allowed to destroy the object whenever no other PCE object has a reference to it. The `fill_pattern_menu/1` predicate then holds an invalid reference.

10.4.3 Using object-level attributes

PCE object-level attributes provide another approach:

```
fill_3(P) :-
        new(D, dialog('Fill 3')),
        make_fill_pattern_menu(M),
        send(D, append, M),
        send(new(P, picture), below, D),
        send(P, attribute, fill_pattern_menu, M),
        send(D, open).

add_box_3(P) :-
        send(P, display, new(B, box(100,100)), point(20,20)),
        send(B, popup, new(Pop, popup)),
        get(P, fill_pattern_menu, M),
        send(Pop, append,
             menu_item(fill,
                       message(B, fill_pattern,
                               M?selection))).
```

```
1 ?-    fill_3(P),
        add_box_3(P).
```

This approach is much better. There no longer is a potential name-conflict and PCE has access to all information it needs for proper memory management. Two disadvantages remain. First of all, the message object has a direct reference to ‘P’ and therefore the entire recogniser object cannot be shared by multiple graphical objects (reused). Second, the code for the box assumes the picture has an attribute fill_pattern_menu and this attribute refers to a menu holding fill-patterns.

10.4.4 Using window and graphical behaviour

All graphicals in PCE have a name, and graphical devices define the method ‘device ← member: name’ to find the (first) graphical with this name. The default name for a graphical is its class name. For dialog-items it is the label of the item. Using ←member results in:

```
fill_4(P) :-
        new(D, dialog('Fill 4')),
        make_fill_pattern_menu(M),
        send(D, append, M),
        send(new(P, picture), below, D),
        send(D, open).

:- pce_global(@fill_with_current_pattern,
              make_fill_with_current_pattern).

make_fill_with_current_pattern(G) :-
        new(G, popup),
        send(G, append,
             menu_item(fill,
                       message(Gr, fill_pattern,
                               ?(?(Gr?frame, member, dialog),
                                 member,
                                 fill_pattern)?selection))).

add_box_4(P) :-
        send(P, display, new(B, box(100,100)), point(20,20)),
        send(B, popup, @fill_with_current_pattern).


fill4 :-
        fill_4(P),
        add_box_4(P).
```

In this example we have made the recogniser generic. This saves both time and memory. Note however that this approach could be used in the previous example as well.

This example has largely the same (dis)advantages as the previous two. As an advantage, the attribute object may be omitted. The assumption here is that the frame the box is in contains a dialog which in turn contains a graphical object named ‘fill_pattern’ that implements a ←selection method yielding an image.

10.4.5 Using user defined classes

Using user-defined classes we can hide the implementation details and make objects depend on each other in a much more organised manner.

```
:- pce_begin_class(fill5, frame).

initialise(F) :->
        send(F, send_super, initialise, 'Fill 5'),
```

```
        send(F, append, new(D, dialog)),
        make_fill_pattern_menu(M),
        send(D, append, M),
        send(new(picture), below, D).

current_fill_pattern(F, P:image) :<-
        get(F, member, dialog, D),
        get(D, member, fill_pattern, M),
        get(M, selection, P).

draw_box(F) :->
        get(F, member, picture, P),
        send(P, display, fillbox(100,100), point(20,20)).

:- pce_end_class.

:- pce_begin_class(fillbox, box).

:- pce_global(@fillbox_recogniser, make_fillbox_recogniser).
make_fillbox_recogniser(G) :-
        Gr = @arg1,
        new(G, popup_gesture(new(P, popup))),
        send(P, append,
             menu_item(fill,
                       message(Gr, fill_pattern,
                               Gr?frame?current_fill_pattern))).

event(B, Ev:event) :->
        (   send(B, send_super, event, Ev)
        ;   send(@fillbox_recogniser, event, Ev)
        ).
:- pce_end_class.
```

```
1 ?- send(new(F, fill5), open),
     send(F, draw_box).
```

The fillbox now only assumes it is contained in an application window that defines ←`current_fill_pattern`, while the application (the frame) hides its internal window organisation using the methods ←`current_fill_pattern` and →`draw_box`.

10.4.6 Summary

Using global references or the Prolog database to keep track of instances in the UI is not the appropriate way. This approach quickly leads to name-conflicts, harms the memory management of PCE and makes it difficult to write reusable code.

Using attributes or user-defined classes to find (graphical) objects solves the name-conflict problems and allows PCE to perform proper memory management. It also allows multiple copies of these windows to run simultaneously. Using user-defined classes allows one to make the code more robust against later changes and allow low-level objects to be better reusable.

Large applications should carefully design the infra-structure to manage the structure of the UI components as well as the relation between UI objects and application entities.See [Wielemaker & Anjewierden, 1989].

Hyper objects as described in section 10.11 form an alternative to relate objects that is suitable if dependent objects cannot rely on each other's existence.

10.5 Relating frames

Applications may consist of multiple `frames`, either permanent, or temporary such as for opening a 'settings' window. This section discusses how frame references can be found, as well as how frames can force the user to deal with this frame first, in favour of all the other frames of the application: modal frames.

10.5.1 Class application

The class `application` is the key to this section. An `application` is a subclass of `visual`, and optionally located between `display` and `frame` in the visual consists-of hierarchy. It defines the '`application` ← `member`' method to located named frames, and thus supports locating frames similar to other graphicals as described in section 10.4.

A `frame` is made part of an `application` using the method '`frame` → `application`'. The application to which a frame is related may be changed. Frames are not required to be part of an application.

frame → application: application*

Changes the application object the receiver belongs too. Combining multiple frame objects in an application allows for finding frames as well as defining modal relations between frames. See '`frame` → `modal`'.

The application to which a frame belongs may be changed at any time. Using the `@nil` argument, to frame is detached from any application.

This method invokes '`application` → `delete`' to the application currently holding the frame (if any) and '`application` → `append`' to the application receiving the frame. These methods may be redefined if an application wants to keep track of its associated frames.

application ← member: name → frame

Return the frame for which '`frame` ← `name`' returns the argument name. See also '`device` ← `member`' and section 10.4.

10.5.2 Transient frames

The term *transient* window is taken from X11. A transient window (frame) is a frame that supports another frame. Transient windows are normally used for prompting. The related method is:

frame → transient_for: frame

Make the receiver a transient window for the argument. This notion is handed to the X11 window manager, but the support varies. Therefore XPCE ensures that:

- *The transient window stays on top*
  Whenever the main window is raised, the transient window is raised too, ensuring the transient window does not get hidden behind the main window.

- *Synchronise status change and destruction*
  Status change (see ‘`frame` → `status`: {unmapped,hidden,iconic,open}’) of the main window are forwarded to the transient windows. If the main window is destroyed, so is the transient window.

10.5.3 Modal operation

The method ‘`frame` → `modal`’ works in combination with class `application` and transient frames to define what frames are temporary insensitive to events, forcing the user to operate on the modal frame first.

frame → modal: {application,transient}*

Operate as a modal frame for all frames in the ←`application`, the frame I am ←`transient_for`, or none. A common sequence to display a modal dialog window centered on a frame is below. Note that, instead of ‘frame →`open_centered`’, one could also have used ‘`frame` ← `confirm_centered`’.

```
settings(Frame) :->
        "Open settings dialog"::
        new(D, dialog(settings)),
        send(D, transient_for, Frame),
        send(D, modal, transient),
        ...,
        <fill the dialog>,
        ...,
        send(D, open_centered, Frame?area?center).
```

Instead of using the center of a frame, the method can also use the location of `@event` to position itself. The code fragment for that is:

```
        ...,
        (   send(@event, instance_of, event)
        ->  get(@event, position, @display, EventPos)
        ;   EventPos = @default)
        ),
        send(D, open_centered, EventPos).
```

10.6 Window layout in a frame

A `frame` is the gateway between a collection of tiled `windows` (graphics, dialog) and the Window System. Any displayed window has a frame. The '`window → create`' method (invoked by '`window → open`') will create a default frame for a window if the window does not have a frame.

Windows in a frame are controlled by a hierarchy of `tile` objects. The leaves of this hierarchy manage an individual window, while the non-leaf tiles either manages a left-to-right list of sub-tiles or a top-to-bottom list. A non-leaf *horizontal* (left-to-right) tile forces all windows to be as high as the highest wishes to be and distributes the horizontal space over the sub-tiles, while a *vertical* tile forces all members to the same (widest) member and distributes the vertical space.

A tile hierarchy is built using the methods '`window → above`', etc. described below. Each window is born with a leaf-tile. If two windows are connected using →`left` or →`right`, a horizontal tile is created. Any other window associated to the left or right will be added as a member to this horizontal tile. A analogous story applies to vertically stacked windows.

If a window or collection of windows is placed left of a vertical tiled stack of windows, a horizontal tile is created.

Whenever a window receives one of the →`above`, →`left`, etc. messages, it will forward these messages to the root of the associated tile hierarchy. Assuming both windows are not yet related, the root tiles negotiate the alignment (combine, become member, create a new root).

Suppose we want to realise the window layout shown in figure 10.2. Now, look for windows that have equal width or height. This is the starting point, **W3** and **W4** in this example. So, the first statement is:

```
send(W3, above, W4)
```

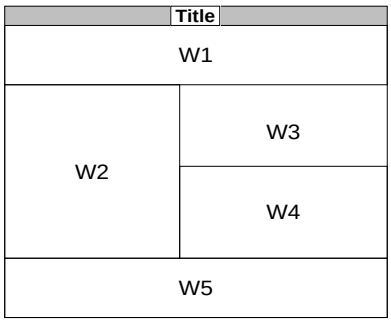


Figure 10.2: Demo window layout

Note that 'send(W4, below, W3)' is the same, except that if both **W3** and **W4** have a `frame` attached, the frame of the *argument* of the message will hold both windows, while the frame of the *receiver* is destroyed.

Now simply continue inwards-out:

```
send(W2, left, W3),
```

```
send(W1, above, W2),
send(W5, below, W2)
```

Note that 'send(W2, left, W4)' is *exactly* the same as positioning **W2** left of **W3**. The resulting tile hierarchy is shown in figure 10.3. The numbers indicate the order of creation of the various objects.

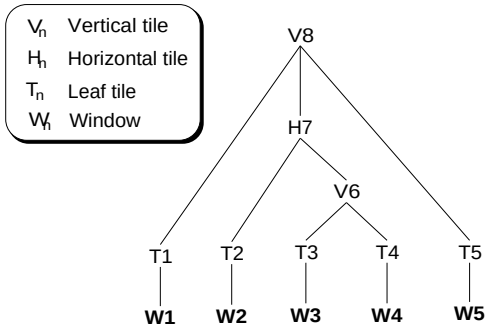


Figure 10.3: Tile hierarchy of example

10.6.1 Windows sizes and automatic adjustment

A `tile` defines six size parameters, described below.

tile → ideal_width: int
tile → ideal_height: int
These two parameters describe the *ideal* size of the tile. The initial values for these parameters are read from the window for which the tile is created. A non-leaf horizontal tile sets the ideal width to the sum of its members and the ideal height to the maximum of its members.

tile → hor_stretch: 0..100
tile → ver_stretch: 0..100
These two parameters describe how easy the window stretches (gets bigger). A non-leaf horizontal tile sets hor_stretch to the maximum of its members and ver_stretch to the minimum of its members.

tile → hor_shrink: 0..100
tile → ver_shrink: 0..100
Same, but deals with making the window/tile smaller than its ideal size.

The various built-in window types have the following defaults shown in table 10.1.

These rules will often suffice for simple applications. You may adjust the stretch and shrink parameters after creating the window, but before opening the window.

10.6.2 Manipulating an open frame

Windows may be added to an open frame using the →`above`, etc. message, specifying any of the member windows as argument. Windows may be deleted from a frame using

class	width	height	hor_shrink	ver_shrink	hor_stretch	ver_stretch
window	200	100	100	100	100	100
picture	400	200	100	100	100	100
dialog	200^{α}	100^{α}	0	0	0	0
browser	25^{β}	10^{β}	0	100	0	100
view	80^{β}	20^{β}	100	100	100	100

[a] If the dialog is not empty, the bounding box of the contents with the ←`gap` around it will be used.
[b] Interpreted as character units.

Table 10.1: The window types and their default sizes

'`frame` → `delete`'. If a window is added or deleted from an open window, the message '`frame` → `fit`' will be called to readjust the contents. This will normally change the size of the frame. If this is not desired, class `frame` must be sub-classed as below. Note that fixed_size should not be set to @on before the frame is open.

```
:- pce_begin_class(my_dynamic_frame, frame,
                   "Add fixed_size").

variable(fixed_size, bool := @off, both,
         "Do not resize on ->fit").

fit(F) :->
        "Request to fit the contents"::
        (   get(F, fixed_size, @on)
        ->  send(F, resize)
        ;   send(F, send_super, fit)
        ).
```

10.7 Informing the user

10.7.1 Aim of the report mechanism

Objects perform actions on behalf of the user. Sometimes it is desirable to inform the application user of progress, status or problems. In older applications, the action code often contained knowledge about the environment in which the object was embedded. This harms reusability and therefore XPCE provides a generic mechanism to deal with such messages.

10.7.2 The report interface

Whenever an object needs to inform the user, the programmer can use the '`object` → `report`' method. The definition of this method is below.

object → report: type, format, argument ...
: Inform the user. The message is constructed from *format* and the *argument* list. See '`string` → `format`' for a description of the C printf-like formatting syntax of XPCE. *type* describes the nature of the message and is used by the reporting mechanism to decide on the presentation-style. See section 10.7.2 for the available types and their meaning.

The object posting the report message will normally not be able to display the message. The main task of the implementations of this method in class `object`, `visual`, `frame` and `dialog` is to divert the report to an object that can present it. By default, this is either a `label` or the `display` object (`@display`).

The implementation of '`object` → `report`' will simply check whether there is a current event (see `@event`), which implicates the action was initiated by the user and then forward the →`report` message to the '`event` ← `receiver`', which is normally the controller activated by the user. If there is no current event, the action must be initiated by a user query to the host-language and therefore '`object` → `report`' will print its feedback to the host-language console using '`@pce` → `format`'.

The implementation of '`visual` → `report`' will simply forward the message to its containing visual object. If the message arrives at an instance of class `frame`, the implementation of '`frame` → `report`' will broadcast the message to each of its windows, but avoid sending it back to the window it came from. If this fails and the frame is a transient frame for another frame, the report is forward to the main frame. Class `dialog` will look for an object named `reporter`. If it can find such an object (typically a `label`), the report is sent to this object.

If all fails, the report will arrive at the `@display` object, which takes care of default handling. See section 10.7.2 on how the different report types are handled.

Information types

The report *type* indicates the semantic category of the report and defines how it is handled. The following report types are defined:

- status
  Information on progress, status change of the system, etc. such information should

not attract much attention. By default, XPCE will format the message at an appropriate location. If no such location can be found the message is ignored.

- inform
  The user requested information and this is the reply. Handled as `status`, but if no appropriate location can be found it will invoke '`@display → inform`', presenting a messagebox.

- progress
  Indicates progress in ongoing computation. Feedback is generally similar to `status`, but followed by a '`graphical → flush`' to take immediate effect. A sequence of `progress` reports should be closed with a `done` report.

- done
  Terminates a (sequence of) →`report`: progress messages.

- warning
  The user probably made a minor mistake for which a simple alert such as provided by '`graphical → alert`' suffices. If there is an appropriate location for the message it will be formatted there.

- error
  Something serious went wrong and the user needs to be informed of this. For example a file could not be read or written. If no appropriate location could be found '`@display → inform`' is used to bring the message to the attention of the user.

10.7.3 Redefining report handling

There are two aspects in which the reporting mechanism can be redefined. The first concerns the *location* and the other the *presentation* of the report. The *location* is changed by defining the method ←`report_to`. All the generic implementations of this method will first invoke ←`report_to` on itself. If this method yields an answer, the report is forwarded to this answer. For example, class `text_buffer` defines a ←`report_to` that forwards all reports to its associated editor object.

The *presentation* is changed by changing the implementation of '`label → report`' or '`display → report`. As this determines the look and feel of an application, applications should be reluctant using this.

10.7.4 Example

The typical way to exploit the report mechanism is by attaching a label named *reporter* to a dialog window of the applications frame. For example, an application consisting of a menubar, browser and a graphical window window could choose to make place for a small dialog for feedback at the bottom. The following fragment would build the window layout of such an application:

```
1 reportdemo :-
2         new(Frame, frame('Reporter demo')),
3         new(B, browser),
```

```
        send(new(picture), right, B),
        send(new(MD, dialog), above, B),
        send(new(RD, dialog), below, B),
        send(Frame, append, B),

        send(MD, append, new(MB, menu_bar)),
        send(MD, pen, 0),
        send(MD, gap, size(0,0)),
        send(MB, append, new(File, popup(file))),
        send_list(File, append,
                  [ menu_item(load,
                              message(@prolog, load, Frame),
                              end_group := @on),
                    menu_item(quit, message(Frame, destroy))
                  ]),

        send(RD, append, label(reporter)),
        send(RD, gap, size(5, 0)),
        send(Frame, open).
```

Now suppose the implementation of the ‘load’ action takes considerable time. The implementation below reads a file assuming each line in the file contains a word.

```
:- pce_autoload(finder, library(find_file)).
:- pce_global(@finder, new(finder)).

load(Frame) :-
        get(@finder, file, exists := @on, FileName),
        send(Frame, report, progress,
             'Loading %s ...', FileName),
        get(Frame, member, browser, Browser),
        new(File, file(FileName)),
        send(File, open, read),
        (   repeat,
                (   get(File, read_line, Line)
                ->  send(Line, strip),
                    send(Browser, append, Line),
                    fail
                ;   !,
                    send(File, close)
                )
        ),
        send(Frame, report, done).
```

The result is shown in figure 10.4.

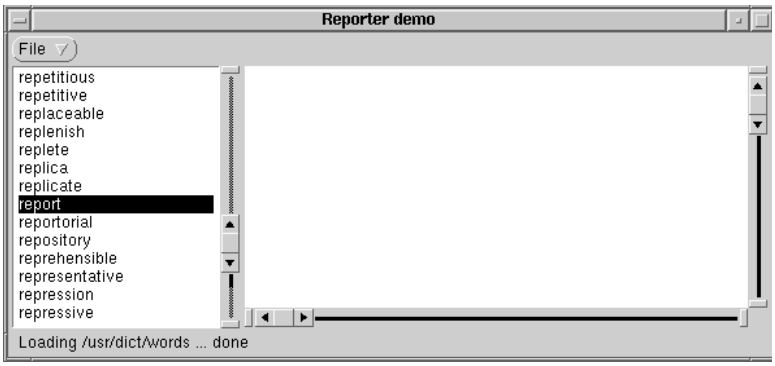


Figure 10.4: The ‘reporter’ demo

10.8 Errors

Errors are abnormalities that are detected during a programs execution. Errors may be caused by internal bugs in XPCE, bugs in the application program and finally by the application user making errors in operating the application (e.g. specifying a protected file).

Errors may also be discriminated according to their 'seriousness': If a certain font cannot be found it can easily be substituted by another. If a method expects an integer argument but the actual argument is a graphical object it is impossible to carry-out the operation. In such cases XPCE will normally trap the tracer. If the user decides to continue execution the method will return failure to its caller. Finally, some problems are categorised as 'fatal'. When such a problem is encountered XPCE does not know how execution can be continued.

All errors (except for some that may result from XPCE bugs during the boot phase of XPCE) are represented by an `error` object. An error object has the following properties:

- id
  Unique identifier name of the error. It may be used to generate errors; look-up error objects in the `@errors` database or catch errors (see `pce_catch_error/2`).
- kind
  The kind describes how serious the error is considered to be. The possible values are: *ignored* if the error is (currently) not regarded an error at all; *warning* if the error is to be reported, but no further action is required; *error* if the error is to be fixed. After printing the error the system will start the tracer, allowing a programmer to examine the problem context. Finally, *fatal* errors do not allow execution to be continued. The system will print context information and request Prolog to abort back to the Prolog interactive top level.
- feedback
  Determines how the error is to be reported. If `print` the error is printed in the Prolog window. If `report` the error is reported using the report mechanism described in section 10.7. The 'report' mechanism is for errors that may be caused by application users, for example file errors. If `throw` and there is a Prolog goal on the stack, the error is mapped to a Prolog exception. See below.

  In the runtime-system, all errors use feedback 'report'.
- format
  A format specification to construct a message text from the context arguments provided by the generator of the error.

The online manual "Errors Browser" may be used to examine the defined errors; change attributes of errors and get precise description of errors.

10.8.1 Handling errors in the application

Sometimes the application wants to anticipate on certain errors. Explicit testing of all conditions is a cumbersome solution to this problem. Therefore XPCE allows catching of errors by the application.

There are two mechanism available to do this. Regardless of the '`error` ← `feedback`' type of the error, all except fatal errors can be silenced using `pce_catch_error/2`:

pce_catch_error(+*ErrorSpec, :Goal***)**
Run *Goal* like once/1. It an error matching *ErrorSpec* is raised, this error is not reported, but stored in '@pce ← last_error'. *ErrorSpec* is the ←id of an error, a chain holding error-ids or @default. The latter implies none but fatal errors are reported.

The example below silently ignores errors from creating a backup of *File*. Note that the call does fail if backup raises an error.

```
...,
pce_catch_error(backup_file, send(File, backup)),
...
```

If the ←feedback of the error is throw and the error is not silenced with pce_catch_error/2 it is mapped to a Prolog exception of the form

error(pce*(Id, ContextArgs), Goal*)

For example:

```
?- catch(send(@pce, foobar), E, true).

E = error(pce(no_behaviour, [@pce/pce,  (->), foobar]),
          send(@pce/pce, foobar))
```

10.8.2 Raising errors

The application programmer may define new (application specific) errors. The error object is a normal XPCE object and may thus be created using new/2. An error is raised by invoking 'object → error'. The example below illustrates this:

```
:- new(_, error(no_user,
                '%N: Unknown user: %s',
                warning, inform)).


        ...,
        (   get(UserDatabase, user, Name)
        ->  ...
        ;   send(UserDatabase, error, no_user, Name)
        ),
        ...
```

Note that the names of errors should be unique. It is advised to reuse existing error-id's if possible.

10.8.3 Repairable errors

On trapping certain 'repairable' errors, XPCE will first raise an *exception*. Exceptions may be trapped by an *exception handler* which may solve the problem. If the exception-handler fails to solve the problem, XPCE will raise an *error*. See section 10.8.

Exceptions are raised by invoking '@pce → exception: id, arg ...'. Exception handlers are registered in the sheet '@pce → exception_handlers', which maps an exception-id onto a code object that handles the exception. The following illustrates this:

```
1 ?- [user].
|: add_user(Name) :- write(Name), nl.
^D

2 ?- send(@pce?exception_handlers, value,
          no_user,
          message(@prolog, add_user, @arg1)).

3 ?- send(@pce, exception, no_user, fred).
fred
```

The *context arguments* passed with an exception are defined by the code raising the exception. The currently defined exceptions are listed below. See also the online manual: 'pce → exception' and 'pce ← exception_handlers'.

- undefined_class
  An attempt is made to reference a non-existing class while doing one of the following: create an object; load an object from file using 'File ← object'; create a subclass. @arg1 is bound to the class-name. This trap is used by pce_autoload/2, as well as by the code that realises compiled classes.

- undefined_assoc
  An attempt is made to resolve a symbolic reference (i.e. @pce), but the reference is unknown. @arg1 is bound to the missing reference name. This trap is used by pce_global/2.

- redefined_assoc
  An attempt is made to create an object with the same symbolic reference as an already existing object. @arg1 is bound to the already existing reference name. This trap is used by library(pce_renew).

- initialisation_failed
  The →initialisation method for some instance failed. @arg1 is bound to the (partial) instance; @arg2, ... are bound to the arguments given to the new-operation.

10.9 Specifying fonts

XPCE's font specification is a two-stage process. In the first stage, XPCE physical fonts are mapped to fonts of the underlying windowing system. In this stage, fonts are identified by their *family*, *style* and *size*. For example

```
font(screen, roman, 13)
```

Refers to a fixed-width font designed for use on the screen that has normal weight, not slanted and 13-pixels high characters.

In the second stage, logical font-names are mapped to their physical implementation. At this level, fonts are identified by a single name from an extensible, but preferably small set.

See section B.5 for a description of Windows specific font issues.

10.9.1 Physical fonts

The default physical font set is built when the first font object is opened (i.e. its window counterpart is located and made available to the text-drawing functions). This set is created from class-variables on the display object. The first class-variable is display.font_families, which defines a chain with names of the font-families. The default value is:[1]

```
display.font_families: \
        [ screen_fonts, \
          courier_fonts, \
          helvetica_fonts, \
          times_fonts, \
          symbol_fonts
        ]
```

Each of these names refers to the name of another resource of class `display`, enumerating the members of this font family. The default value can be examined using the online manual. Below is the default value for the `screen_fonts` font-set for X11:

```
display.screen_fonts: \
        [ font(screen, roman, 10, "6x10"), \
          font(screen, roman, 12, "6x12"), \
          font(screen, roman, 13, "8x13"), \
          font(screen, roman, 14, "7x14"), \
          font(screen, roman, 15, "9x15"), \
          font(screen, bold, 13, "8x13bold"), \
          font(screen, bold, 14, "7x14bold"), \
          font(screen, bold, 15, "9x15bold") \
        ]
```

The set of predefined physical fonts can be examined using the FontViewer demo application accessible through the online manual tools.

[1] See section 8 for the default syntax.

Defining additional fonts

If an application needs additional fonts, such fonts can be declared using directives. The fourth initialisation argument of class `font` determines the window-system font that will be mapped. The syntax for this argument depends on the window-system used. For this Unix/X11 version it is a string consisting of 15 '-' separated fields. A font can be searched using `xfontsel(1)` or the much better GNOME-project `gfontsel(1)`.

For example, the 14-points 'courier new' TrueType font can be registered using:

```
:- initialization
   new(_, font(courier, roman, 14,
               '-winfonts-courier new-medium-r-normal-*-*-140-*-*-
m-*-iso8859-1')).
```

This specification has various drawbacks. For example, another library or application loaded on top of the same XPCE process may be using the symbol,roman,14 specification, but bound to another window-system font. A user may try to run your application on an environment that does not have this font. Part of these problems can be eliminated by binding the font to a logical font name. See also section 10.9.2.

```
:- initialization
   send(@display, font_alias,
        adobesymbol,
        font(symbol, roman, 14,
             '-*-symbol-*-*-*-*-14-*-*-*-*-*-adobe-*')).
```

The application will refer to this font using the font-alias. user has other preferences or the font is not available, the user may specify the font using the display.user_fonts class-variable described in section 10.9.2.

10.9.2 Logical fonts

.

It is not wise let your application code speak about physical fonts as the user or interface guidelines may prefer using a different font-palette. For this reason the display defines a mapping between logical font names and physical fonts. Applications are encouraged to use logical font names as much as possible and leave the assignment to physical fonts to the users preferences. XPCE predefines the following logical font-names. The value gives the default assignment for these fonts.

- normal font(helvetica, roman, 12)
  The default font. Normally a proportional roman font. Should be easy to read.
- bold font(helvetica, bold, 12)
  Bold version of the normal font.
- italic font(helvetica, oblique, 12)
  Slanted version of the normal font. Note that italic fonts should not be used for long text as italics is generally not easy to read on most displays.

- small font(helvetica, roman, 10)
  Small version of the normal font. To be used in notes, subscripts, etc. May not be so easy to read, so avoid using it for long texts.

- large font(helvetica, roman, 14)
  Slightly larger version of the normal font.

- boldlarge font(helvetica, bold, 14)
  Bold version of large.

- huge font(helvetica, roman, 18)
  Even larger font. To be used for titles, etc.

- boldhuge font(helvetica, bold, 18)
  Bold version of huge.

- fixed font(screen, roman, 13)
  Terminal font. To be used for code fragments, code editors, etc. Should be easy to read.

- tt font(screen, roman, 13)
  Same as `fixed`.

- boldtt font(screen, bold, 13)
  Bold terminal font.

- symbol font(symbol, roman, 12)
  Symbol font using the adobe symbol-font encoding. This font provides many mathematical symbols.

The end-user of an XPCE application can define the class-variable display.user_fonts to overrule fonts. The example below re-binds the most commonly used fonts to be slightly larger and choose from the Times font family rather than the Helvetica fonts.

```
display.user_fonts: \
        [ normal := font(times, roman, 14), \
          bold   := font(times, bold, 14), \
          italic := font(times, italic, 14) \
        ]
```

The mapping between logical font names and physical fonts is realised by the methods '`display` ⇆ `font_alias`' additional font aliases may be loaded using '`display` → `load_font_aliases`'.

Class `font`'s predefined conversion will translate names to font objects. This implies that for any method expecting a font object the programmer can specify the font-name instead. In those (rare) cases where a font needs to be passed, but the type-specification does not require this, the conversion must be done explicitly. The preferred way to make the conversion is using the font type object:

```
        ...,
        get(type(font), check, bold, BoldFont),
        ...,
```

10.10 Using images and cursors

Many today graphical user interfaces extensively use (iconic) images. There are many image formats, some for specific machines, some with specific goals in mind, such as optimal compression at the loss of accuracy or provide alternatives for different screen properties.

One of XPCE's aim is to provide portability between the supported platform. Therefore, we have chosen to support a few formats across all platforms, in addition to the most popular formats for each individual platform.

10.10.1 Colour handling

Colour handling is a hard task for todays computerprogrammer. There is a large variety in techniques, each providing their own advantages and disadvantages. XPCE doesn't aim for programmers that need to get the best performance and best results rendering colours, but for the programmer who wants a reasonable result at little effort.

As long as you are not using many colours, which is normally the case as long as you do not handle full-colour images, there is no problem. This is why this general topic is handled in the section on images.

Displays differ in the number of colours they can display simultaneously and whether this set can be changed or not. X11 defines 6 types of visuals. Luckily, these days only three models are popular.

- *8-bit colour-mapped*
  This is that hard one. It allows displaying 256 colours at the same time. Applications have to negotiate with each others and the windowing systems which colours are used at any given moment.

  It is hard to do this without some advice from the user. On the other hand, this format is popular because it leads to good graphical performance.

- *16-bit 'high-colour'*
  This schema is a low-colour-resolution version of true-colour, normally using 5-bit on the red and blue channels and 6 on the green channel. It is not very good showing perfect colours, nor showing colour gradients.

  It is as easy for the programmer as true-colour and still fairly efficient in terms of memory.

- *24/32 bit true-colour*
  This uses the full 8-bit resolution supported by most hardware on all three channels. The 32-bit version wastes one byte for each pixel, achieving comfortable alignment. Depending on the hardware, 32 bit colour is sometimes much faster.

We will further discuss 8-bit colour below. As handling this is totally different in X11 and MS-Windows we do this in two separate sections.

Colour-mapped displays on MS-Windows

In MS-Windows one has the choice to stick with the reserved 20 colours of the system palette or use a colourmap (palette, called by Microsoft).

If an application chooses to use a colourmap switching to this application causes the entire screen to be repainted using the application's colourmap. The idea is that the active application looks perfect and the other applications look a little distorted as they have to do their job using an imperfect colourmap.

By default, XPCE makes a `colour_map` that holds a copy of the reserved colours. As colours are required they are added to this map. This schema is suitable for applications using (small) icons and solid colours for graphics. When loading large colourful images the colourmap will get very big and optimising its mapping to the display slow and poor. In this case it is a good idea to use a fixed colourmap. See class `colour_map` for details.

When using XPCE with many full-colour images it is advised to use high-colour or true-colour modes.

Colour-mapped displays on X11/Unix

X11 provides colourmap sharing between applications. This avoids the flickering when changing applications, but limits the number of available colours. Even worse, depending on the other applications there can be a large difference in available colours. The alternative is to use a private colourmap, but unlike MS-Windows the other applications appear in totally random colours. XPCE does not support the use of private colourmaps therefore.

In practice, it is strongly advised to run X11 in 16, 24 or 32 bit mode when running multiple applications presenting colourful images. For example Netscape insists creating its own colourmap and starting Netscape after another application has consumed too many colours will simply fail.

10.10.2 Supported Image Formats

The table below illustrates the image format capabilities of each of the platforms. Shape support means that the format can indicate *transparent* areas. If such an image file is loaded, the resulting `image` object will have an '`image` ← `mask`' associated: a monochrome image of the same side that indicates where paint is to be applied. This is required for defining cursors (see 'cursor →`initialise`') from a single image file. *Hotspot* means the format can specify a location. If a Hotspot is found, the '`image` ← `hot_spot`' attribute is filled with it. A Hotspot is necessary for cursors, but can also be useful for other images.

Format	Colour	HotSpot	Shape	Unix/X11		Win32	
				load	save	load	save
Icons, Cursors and shaped images							
XPM	+	+	+	+	+	+	+
ICO	+	-	+	-	-	+	-
CUR	+	+	+	-	-	+	-
Rectangular monochrome images							
XBM	-	-	-	+	+	+	-
Large rectangular images							
JPEG	+	-	-	+	+	+	+
GIF	+	-	+	+	+	+	+
BMP	+	-	-	-	-	+	-
PNM	+	-	-	+	+	+	+

The XPM format (**X P**ix**M**ap) is the preferred format for platform-independent storage of images that are used by the application for cursors, icons and other nice pictures. The XPM format and supporting libraries are actively developed as a contributed package to X11.

Creating XPM files

Unix There are two basic ways to create XPM files. One is to convert from another format. On Unix, there are two popular conversion tools. The `xv` program is a good interactive tool for format conversion and applying graphical operations to images.

ImageMagic can be found at `http://www.simplesystems.org/ImageMagick/` and provides a comprehensive toolkit for converting images.

The `pixmap` program is a comprehensive icon editor, supporting all of XPM's features. The image tools mentioned here, as well as the XPM library sources and a FAQ dealing with XPM related issues can be found at `ftp://swi.psy.uva.nl/xpce/util/images/`

Windows XPCE supports the Windows native `.ICO`, `.CUR` and `.BMP` formats. Any editor, such as the resource editors that comes with most C(++) development environments can be used. When portability of the application becomes an issue, simply load the icons into XPCE, and write them in the XPM format using the 'image →`save`' method. See the skeleton below:

```
to_xpm(In, Out) :-
        new(I, image(In)),
        send(I, save, Out, xpm),
        free(I).
```

Note that the above mentioned ImageMagick toolkit is also available for MS-Windows.

Using Images

Images in any of the formats are recognised by many of XPCE's GUI classes. Table table 10.2 provides a brief list:

`bitmap`	A `bitmap` converts an image into a first class `graphical` object that can be displayed anywhere.
`cursor`	A `cursor` may be created of an image that has a mask and hot-spot.
‘`frame → icon`’	Sets the icon of the frame. The visual result depends on the window system and X11 window manager used. Using the Windows 95 or NT 4.0 shell, the image is displayed in the task-bar and top-left of the window.
‘`dialog_item → label`’	The label of all subclasses of class `dialog_item` can be an image.
‘`label → selection`’	A `label` can have an image as its visualisation.
‘`menu_item → selection`’ ‘`style → icon`’	The items of a menu can be an image. Allows association of images to lines in a `list_browser`, as well as marking `fragments` in an `editor`.

Table 10.2: GUI classes using `image` objects

10.11 Using hyper links to relate objects

A `hyper` is a binary relation between two objects. Hypers are, like `connection` objects, guarded automatically against destruction of one of the related objects. Special methods allow for easy communication between *hypered* objects.

Hypers form an adequate answer if objects need to be related that depend temporary and incidentally on each other. It is possible to be informed of the destruction of hypers, which enables a hypered object to keep track of its environment. Good examples for the usage of hypers are to express the relation between multiple *frame* objects working together to form a single application or maintaining the relation between an application object (persistent object, model) and its visualisation (controller).

Of course relations between objects can be maintained using instance-variables, but this process requires awareness from both related objects as well as significant bookkeeping.

10.11.1 Programming existence dependencies

The example of this section demonstrates a common existence relationship. If the ‘main’ object is destroyed, all related ‘part’ objects should be destroyed too, but if a part is destroyed, the main should not be destroyed. This semantic is expressed using a refinement of class `hyper` that can be found in library(`hyper`) of the XPCE/Prolog libraries.

```
:- pce_begin_class(partof_hyper, hyper,
                   "<-to is a part of <-from").

unlink_from(H) :->
        "->destroy the <-to part"::
        get(H, to, Part),
        (   object(Part),
            send(Part, has_send_method, destroy)
        ->  send(Part, destroy)
        ;   free(Part)
        ),
        free(H).

:- pce_end_class.
```

This hyper is demonstrated in the following application. We have an application for editing a graphical representation. The colour of the objects can be modified by double-clicking an object and selecting a colour in a dialog window. In this example we do not use a modal dialog and using the hyper serves two purposes. First of all it tells the dialog what object should be changed, but second, it ensures the dialog is destroyed if the box is.

```
:- use_module(library(hyper)).

:- pce_begin_class(link_demo, picture).

initialise(P) :->
        send_super(P, initialise, 'Link Demo'),
        send(P, recogniser,
             click_gesture(left, '', single,
```

```
                          message(P, add_box, @event?position))).

add_box(P, At:point) :->
        send(P, display, new(link_box), At).

:- pce_end_class(link_demo).

:- pce_begin_class(link_box, box).

handle(w/2, 0, link, north).
handle(w/2, h, link, south).
handle(0, h/2, link, west).
handle(w, h/2, link, east).

initialise(B) :->
        send_super(B, initialise, 100, 50),
        send_list(B, recogniser,
                  [ click_gesture(left, '', double,
                                  message(B, edit)),
                    new(connect_gesture),
                    new(move_gesture)
                  ]).

colour(red).
colour(green).
colour(blue).
colour(yellow).

edit(B) :->
        "Allow changing colour"::
        new(D, dialog('Select colour')),
        send(D, append, new(M, menu(colour, choice,
                                    message(?(D, hypered, box),
                                            fill_pattern,
                                            @arg1)))),
        (   colour(Colour),
            send(M, append,
                 menu_item(colour(Colour),
                           @default,
                           pixmap(@nil,
                                  background := Colour,
                                  width := 32,
                                  height := 16))),
            fail
        ;   true
        ),
        send(D, append, button(done, message(D, destroy))),
        new(_, partof_hyper(B, D, dialog, box)),
        get(B, display_position, PosB),
        get(PosB, plus, point(20,100), PosD),
        send(D, open, PosD).

:- pce_end_class(link_box).
```

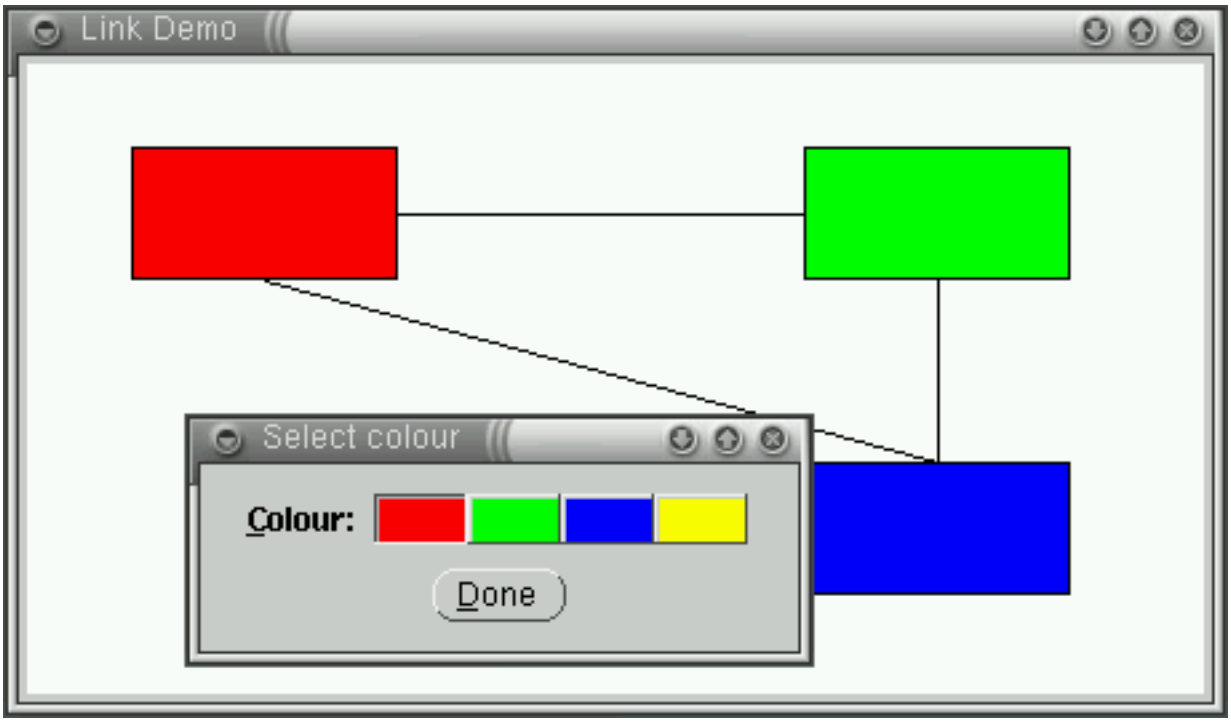


Figure 10.5: Using a hyper to link a window to an object

10.11.2 Methods for handling hyper objects

Methods on class hyper

hyper → initialise: F:object, T:object, FName:name, TName:[name]

Create a new hyper object. Seen from *F*, this hyper is called *FName*; seen from *T* it is called *TName*. The default for *TName* is *FName*.

hyper → unlink_from

Called by the object-management system when the ←`from` side of the hyper is being destroyed. May be refined.

hyper → unlink_to

Called by the object-management system when the ←`to` side of the hyper is being destroyed. May be refined.

Methods on class object

Below are the two most commonly used methods dealing with hypers and defined on class `object`. XPCE defines various other methods for deleting and inspecting the hyper structure. Use the online manual for details.

object → send_hyper: Name:[name], Selector:name, Arg:unchecked ...

Broadcast a send-operation to all (named) ←hypered objects. Similar to ←get_hyper, but does not stop if the method is received successfully. Succeeds if at least one hypered object accepted the message.

object ← hypered: Name:[name], Test:[code] → object

Find a hyper-related object. Name is the name of the hyper (seen from the side of the receiver). Test is an optional additional test. If present, this test is executed using the arguments given below. The first matching object is returned. See also ←all_hypers.

@arg1	This object
@arg2	The hyper object
@arg3	The object at the other end of the hyper

10.12 User defined graphicals

This section discusses various approaches for defining new graphical objects. XPCE offers three approaches for defining new graphical objects:

- Combining graphicals
  The simplest way to create a new graphical object is by combining multiple graphical objects on a graphical device. The following predicate creates a 'text-box':

```
text_box(TB, Text, Width, Height) :-
        new(TB, device),
        send(TB, display,
             new(B, box(Width, Height))),
        send(TB, display,
             new(T, text(Text, center, normal))),
        send(T, center, B?center).
```

  For some applications, this is a suitable and simple approach. However, it is not a very good approach to build a library of GUI objects or, more in general, to make *generic* and *reusable* new graphical objects. The above object does not handle resize properly, and the user has to know the internal structure to modify the object.

- Subclassing class device
  Since the introduction of user-defined classes (see section 7), sub-classing `device` is a common way to define new graphicals. Using this technique, '`device` → `initialise`' is refined to display the part of the compound graphical. '`device` → `event`' and '`device` → `geometry`' are normally redefined to define event-handling and resize of the new graphical object. See section 7.3.2 for details.

- (Re)defining the repaint method
  The method '`graphical` → `_redraw_area`' can be redefined to define the look of a graphical. We will discuss the advantages and disadvantages of this approach in this section and give some examples.

10.12.1 (Re)defining the repaint method

The most basic way to (re)define the look of a graphical object is by redefining the method that paints the graphical. This method is called →`_redraw_area`. The method →`_redraw_area` **cannot be called directly** by the user, but it is called by the graphical infra-structure whenever the graphical needs to be repainted. The definition of the method is below:

graphical → _redraw_area: Area:area
  This method is called by the repaint infra-structure of XPCE. Its task is to paint the graphical on the current graphical device. *Area* indicates the area —in the coordinate system of the device— that needs to be repainted. This area is guaranteed to overlap with the ←`area` of the graphical.

→draw	Paint other graphical
→clip	Clip to area or ←area of graphical
→unclip	Undo last →clip
→save_graphics_state	Save current pen and colours
→restore_graphics_state	Restore saved values
→graphics_state	Set graphics attributes
→draw_arc	Draw ellipse-part
→draw_box	Draw rectangle (rounded, filled, etc.)
→draw_fill	Fill/invert/clear rectangle
→draw_image	Paint (part of) image
→draw_line	Draw a line segment
→draw_poly	Draw a polygon
→draw_text	Draw string in font
→paint_selected	Paint visual feedback of →selected

Table 10.3: Methods for (re)defining →_redraw_area

It is not allowed for this method to paint outside the ←area of the receiver. There is no clipping (see →clip) to prevent this. If there is no cheap way to prevent this, bracket the graphical operations in →clip and →unclip, but be aware that setting and undoing the clip-region is an expensive operation. Note that is is **not** necessary to limit the applied paint only inside the given argument *Area*. The graphical infrastructure automatically clips all graphical operation to this area. In general, *Area* should only be considered to avoid large numbers of unnecessary drawing operations.

There are three sets of methods to implement the drawing job. The first is 'graphical → draw', that allows drawing other graphical objects in this place. The second are methods to manipulate the clipping and state of the graphical device. The last is a set of methods to realise primitive drawing operations, such as drawing lines, rectangles, images, text, etc. These methods can be used in any combination. It is allowed, but not obligatory, to call the →send_super method in order to invoke the default behaviour of the graphical. These methods are summarised in table 10.3. Full documentation is available from the online manual.

10.12.2 Example-I: a window with a grid

XPCE built-in class window does not provide a grid. Implementing a grid using graphical objects is difficult. The best approach would be to display a device on the window that provides the background and displays the lines of the grid. The resize and scroll messages need to be trapped to ensure the proper number of lines are displayed with the correct length. Furthermore, the code handling the inside of the window needs to be aware of the grid. It should ensure the grid is not exposed or deleted, etc.

It is much simpler to redefine the 'window → _redraw_area' method, paint the grid and then call the super-method. The code is below.

```
1 :- pce_begin_class(grid_picture, picture,
```

```
                    "Graphical window with optional grid'").'

variable(grid,     '1..|size*' := 20, get,
         "Size of the grid").
variable(grid_pen, pen,               get,
         "Pen used to draw the grid").

initialise(P, Lbl:[name], Size:[size], Disp:[display]) :->
        send(P, send_super, initialise, Lbl, Size, Disp),
        (   get(@display, visual_type, monochrome)
        ->  Texture = dotted, Colour = black
        ;   Texture = none,   Colour = grey90
        ),
        send(P, slot, grid_pen, pen(1, Texture, Colour)).

'_redraw_area'(P, A:area) :->
        "Draw a grid"::
        get(P, grid, Grid),
        (   Grid \== @nil
        ->  (   integer(Grid)
            ->  GX = Grid,
                GY = Grid
            ;   object(Grid, size(GX< GY))
            ),
            send(P, save_graphics_state),
            get(P, grid_pen, pen(Pen, Texture, Colour)),
            send(P, graphics_state, Pen, Texture, Colour),
            object(A, area(X, Y, W, H)),
            StartX is (X//GX) * GX,
            StartY is (Y//GY) * GY,
            Xlines is ((W + X - StartX)+GX-1)//GX,
            Ylines is ((H + Y - StartY)+GY-1)//GY,
            (   between(1, Xlines, Xline),
                    Xnow is StartX + (Xline-1)*GX,
                    send(P, draw_line, Xnow, Y, Xnow, Y+H),
                fail
            ;   true
            ),
            (   between(1, Ylines, Yline),
                    Ynow is StartY + (Yline-1)*GY,
                    send(P, draw_line, X, Ynow, X+W, Ynow),
                fail
            ;   true
            ),
            send(P, restore_graphics_state)
        ;   true
        ),
        send(P, send_super, '_redraw_area', A).


grid(P, Grid:'1..|size*') :->
        send(P, slot, grid, Grid),
        send(P, redraw).                % changed?
```

```
55
56 grid_pen(P, Penn:pen) :->
57         send(P, slot, grid_pen, Pen),
58         send(P, redraw).                % changed?
59
60 :- pce_end_class.
```

10.12.3 Example-II: a shape with text

The following example is yet another implementation of a shape filled with text. Redefining →_redraw_area has several advantages and disadvantages over the device based implementation:

- ++ Memory usage
  This approach uses considerably less memory than the combination of a device, box and text.
- -- Poor PostScript quality
  The current version of the system will generate PostScript for user-defined graphicals by painting the graphical on an image and translating the result in a PostScript image description.
- -- More rigid
  This version of the text-box does not have different colours for box and text, etc. Of course it is possible to implement a version with all thinkable attributes, but this is a lot of work.

  Implementing edit facilities for the text will be hard. The best approach would be to display a normal text object on top of the text-box and replace the ⇆string when editing is finished.

```
1  :- pce_begin_class(text_shape, graphical,
2                     "text with box or ellipse").
3
4  variable(string,        char_array,     get,
5           "Displayed string").
6  variable(font,          font,           get,
7           "Font used to display string").
8  variable(shape,         {box,ellipse},  get,
9           "Outline shape").
10
11 initialise(S, Str:string=char_array, Shape:shape={box,ellipse},
12            W:width=int, H:height=int, Font:[font]) :->
13         default(Font, normal, TheFont),
14         send(S, send_super, initialise, 0, 0, W, H),
15         send(S, slot, string, Str),
16         send(S, slot, shape, Shape),
17         send(S, slot, font, TheFont).
18
19 '_redraw_area'(S, _A:area) :->
20         get(S, area, area(X, Y, W, H)),
```

```
21          get(S, string, String),
22          get(S, font, Font),
23          get(S, shape, Shape),
24          send(S, clip),                  % text may be bigger
25          (   Shape == box
26          ->  send(S, draw_box, X, Y, W, H)
27          ;   send(S, draw_arc, X, Y, W, H)
28          ),
29          send(S, draw_text,
30               String, Font, X, Y, W, H,
31               center, center),
32          send(S, unclip),
33          send(S, send_super, redraw).
34
35  :- pce_end_class.
```

10.13 Printing from XPCE applications

You're wonderful application is finished ... but the users want **printing???**. What to do? A computer screen is not a set of paper-sheets and therefore there is no trivial answer to this question. It depends on the nature of the data that needs to be printed, and also on the operating system used.

In Unix , printing is achieved by producing a document in a format that can be understood by the print-spooler program, normally either plain text or PostScript. If formatted text and/or included graphics are required it is often desirable to produce input for a formatting program, call the formatter and send the result to the printer.

In Windows the printer is not driven by a document, but using a series of calls on a GDI (Graphical Device Interface) representing the printer. The good news of this is that whatever you can get on the screen you easily get on the printer. The bad news has been explained above: paper is not the same as your screen. It has pages, is generally worse in colour-handling but provides a much higher resolution. The users do not expect a series of screendumps from your applications. Most Windows applications however are WYSIWYG and there are no established standards for formatting applications.

10.13.1 Options for document generation

Below is a brief overview of the options available.

- *Generating PostScript*
  All XPCE `graphical` objects support the method ←`postscript` that creates an Adobe PostScript representation of the object. For most objects used in diagrams (lines, curves, text), the produced PostScript is clean PostScript ready for perfect scaling. The remaining objects (for example a menu or button) are translated into an `image` which is then translated to PostScript. Such objects scale poorly.

  This facility is useful for creating high-quality diagrams that can be imported in a text created on an external application. This is fairly portable, but using Microsoft applications you must have a PostScript printer attached and there is no previewing. On Windows platform there is no well-supported automated way to print a PostScript file unless you can asks your users to install a PostScript viewer such as Alladin GsView.

- *Generating plain text*
  If you need to produce a listing, you can generate a plain ISO-Latin-1 (or other 8-bit character set) string and print this. On Unix this is achieved by sending the text to the printer-spooler. On Windows you can save the data to a temporary file and start the command `notepad /p "file"`. This route provides no support for graphics or any kind of advanced formatting.

- *Generating markup*
  Translating project data to output on a page is a craft that is understood by text-manipulation programs that accept a high-level input specification such as Troff, LaTeX or an SGML variant. If you can expect your users to have or install a particular

package and this package can deal with PostScript graphics this is the ideal way to generate good-looking documents from your application. On Unix these tools are widely available and installed on most machines. Most of them are available on Windows, but not installed on most systms.

The library(`http/html_write`) library described in section 11.9.2 provides a good infra-structure for emitting documents in HTML or another SGML or XML dialect. There is no such library for LaTeX, but this can be designed using the same principles.

Using HTML, the application can be transformed into a web-server using the infrastructure described in section 11.9. The user can use standard web-technologies to process the page. Unfortunately well-established web technology does not support vector-drawings, though the emerging SVG technology may change that.

- *Generating a Windows meta-file*
  Windows metafiles are implemented in the class `win_metafile`, providing both input and output of metafiles. Such files preserve the vector properties of XPCE graphicals and can be imported in most native Windows applications. Unfortunately the scaling properties, especially of text, are much worse than PostScript.

  An example of exporting Windows Metafiles is in PceDraw in the file library(`draw/canvas`).

- *Printing to a Windows printer*
  Using class `win_printer`, the user can open a device on which graphicals can be painted and that can be advanced to the next page. This technique only works on Windows and requires you to program all details of the page. For WYSIWYG objects such as most drawings, this techinique is appropriate, but rendering textual documents require the implementation of a formatter, where you are responsible for page headers and footers, alignment, etc.

  Formatting text in sections, paragraphs lists, etc. is provided by the XPCE document-rendering classes described in section 11.10. These classes cannot handle pagination though. Another alternative is the use of `editor` and friends, drawing the `text_image` on a page while traversing through the document.

  An example of printing using `win_printer` is in PceDraw in the file library(`draw/canvas`).

11 Commonly used libraries

In this chapter we document some of the libraries from the ⟨*pcehome*⟩`/prolog/lib` XPCE/Prolog library. The libraries described here are only the commonly used ones. For more information check the file `Overview` in the library directory and the source-code of the library.

11.1 **Asking a filename**
This library defines the object `@finder`, instance of `finder`. The finder allows for asking filenames.

11.2 **Show help-balloon**
The library library(`help_message`) registers balloon-text with graphical objects.

11.3 **Dialog utilities**
Toolbars and reporting facilities. Includes example code for an application framework.

11.4 **Table-of-content like hierarchies**
This library extends class `tree`, displaying a modern-style hierarchy inside a window. This library is used for displaying the VisualHierachy and ClassHierarchy tools of the XPCE manual toolkit. See chapter 3.

11.5 **Tabular layout**
Primitives for dealing with tables.

11.6 **Plotting graphs and barcharts**
This section describes a number of libraries providing primitives for drawing graphs and barcharts.

11.7 **Multi-lingual applications**
Discusses support for multi-lingual applications.

11.8 **Drag and Drop Interface**
This library allows for drapping objects within one XPCE application.

11.9 **Playing WEB (HTTP) server**
Class `httpd` is a subclass of `socket` that deals with the HTTP protocol. It allows XPCE to act as a web-server.

11.10 **Document rendering**
Primitives for rendering mixed text/graphics, handling fonts, alignment, tables and other common text-layout primitives.

11.11 Library “broadcast” for all your deliveries

The library(`broadcast`) library is a pure-Prolog library sending messages around in your application. It is especially useful for blackboard-like architectures.

11.1 Library “find_file”

The library library(find_file) defines the class finder, representing a modal dialog window for entering a filename. This class defines the method ‘finder ← file’:

finder ← file: Exists:[bool], Ext:[name—chain], Dir:[directory], Def:[file] → name

Ask the user for a file. If *Exists* is @on, only existing files can be returned. *Ext* is either an atom denoting a single extension or a chain of allowed extensions. Extensions can be specified with or without the leading dot. I.e. both pl and '.pl' are valid specifications of files ending in '.pl'. *Dir* is the directory to start from. When omitted, this is the last directory visited by this instance of finder or the current working directory. *Def* is the default file-name returned. When omitted there is no default.

Below is the typical declaration and usage. In this example we ask for an existing Prolog file.

```
:- pce_autoload(finder, library(find_file)).
:- pce_global(@finder, new(finder)).

        ...,
        get(@finder, file, @on, pl, PlFile),
        ...,
```

The Windows version of XPCE implements the method ‘display ← win_file_name’, using the Win32 API standard functions GetOpenFileName() or GetSaveFileName() to obtain a file-name for saving or loading. If this method is defined, ‘finder ← file’ uses it, showing the users familiar dialog.

Names for the extensions (as ‘Prolog file’ rather than ’*.pl’) can be defined by extending the multifile predicate pce_finder:file_type/2. See the library source for the standard definition.

11.2 Showing help-balloons

The library library(help_message) provides support for displaying *balloons*. After loading this library, a loc_still event causes the system to look for a graphical implementing the ←help_message method. If ←help_message: tag yields a string, the library shows a little window with the message. Moving the pointer out of the area of the graphical or pressing a button causes the feedback window to disappear.

This tenchnique is commonly used in modern interfaces to provide feedback on the functions behind icons. See also section 11.3.2.

In addition to registering the global event-handler, the library defines →help_message to the classes visual, graphical and menu.

visual → help_message: {tag,summary}, string*
: Register *string* as tag (balloon) or extensive help message (summary) for the receiving object. At the moment summary is not used.

visual ← help_message: {tag,summary}, event → string
: This message is defined to return the help-message registered using →help_message. User-defined classes may consider redefining this method to generate the help-message on-the-fly.

Here is a typical usage for this library.

```
1 :- use_module(library(help_message)).
2 resource(print, image, image('16x16/print.xpm')).
3
4         ...
5         send(X, append, new(B, button(print))),
6         send(B, label, image(resource(print))),
7         send(B, help_message, tag, 'Print document'),
```

11.3 Dialog support libraries

This section deals with a number of classes from the library to simplify the creation of dialog windows.

11.3.1 Reporting errors and warnings

Error, and warning and informational messages are raised using the →`report` or →`error` method defined on all XPCE objects. Basic error and message handling is described in section 10.8. The library library(`pce_report`) defines the classes `reporter` and `report_dialog`.

- `reporter`
  This is a refinement of class `label`, displayed using the fashionable lowered 3D-style. In addition, it redefines the →`report` message to colour error messages red.

- `report_dialog`
  This is a trivial subclass of `dialog`, displaying a `reporter` and constraining this reporter to occupy the entire window.

An example using these classes is in section 11.3.2.

11.3.2 Toolbar support

The library library(`toolbar`) defines the classes `tool_bar`, `tool_button` and `tool_status_button` to simplify the definition of tool-bars.

tool_bar → initialise: Client:object*,
Orientation:[{horizontal,vertical}]
Create a `tool_bar` for which the buttons execute actions on *Client* (see class `tool_button` for details). By default the buttons are placed left-to-right, but using `vertical` *Orientation* they can be stacked top-to-bottom.

tool_bar → append: Button:tool_button | {gap}
Append a tool-button to the bar or, using the name `gap`, make a small gap to separate logical groups of buttons.

tool_bar → activate
Send →`activate` to all member buttons, reflecting whether they are ready to accept commands or 'grayed-out'.

tool_button → initialise: Action:name | code, Label:name | image,
Balloon:[name | string], Condition:[code]*
Define a button for '`tool_bar → append`'. *Action* is the action to execute. If this is a plain atom, this method without arguments is invoked on the '`tool_bar ← client`'. If it is a code object this code is simply executed. *Label* is the label. Normally for tool-bars this will be an `image` object. *Balloon* defines the text for the popup-window if the user rests the pointer long enough on the button. It it is a name, this balloon is subject to '`name ← label_name`' (see section 11.7), otherwise it is passed literally. Finally, if

Condition is present it is evaluated by →`activate` to determine the activation-state of the button.

tool_button → activate
If ←`condition` is present, evaluate it and send →`active`.

tool_button → active: Active:bool
If `@off`, deactivate the button and provide visual feedback for this.

A `tool_status_button` is toggled between depressed state and normal state on each click. If it has an atomic ←`action` it will send *action*: `@on` to the client when going to depressed state and *action*:`@off` when returning to normal state. If the ←`action` is a code object this boolean will for forwarded over the code object. See section 10.2.

11.3.3 Example

The example below uses these classes as well as class `menu_bar` to arrive at a typical modern application layout.

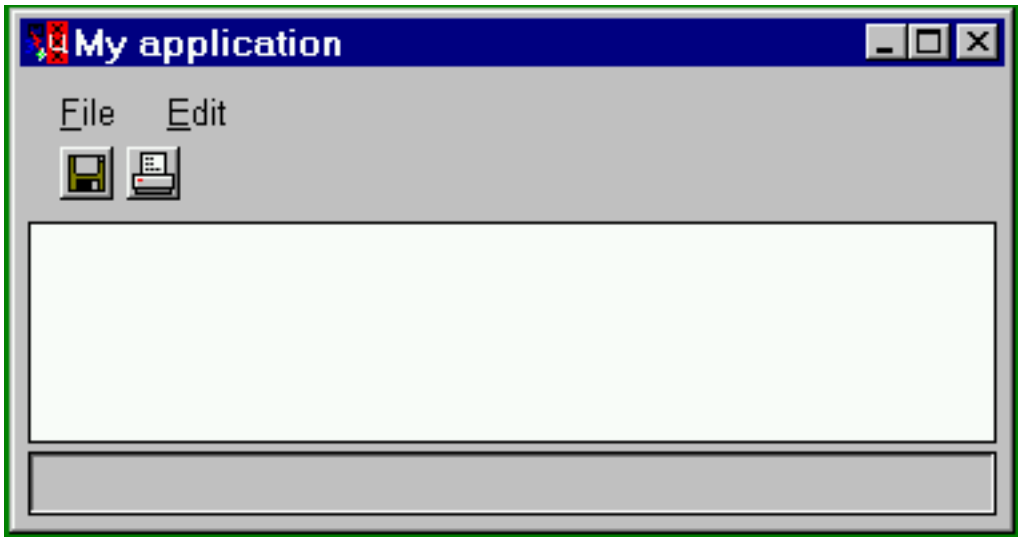


Figure 11.1: Simple application framework

```
%       Pull in the classes

:- pce_autoload(report_dialog, library(pce_report)).
:- pce_autoload(tool_bar, library(toolbar)).
:- pce_autoload(finder, library(find_file)).
:- pce_global(@finder, new(finder)).

%       Define icons as program resources

resource(printer,       image,  image('16x16/print.xpm')).
resource(floppy,        image,  image('16x16/save.xpm')).

```

```
%       Define the application as a subclass of frame.

:- pce_begin_class(myapp, frame,
                   "Frame representing the application").

initialise(MyApp) :->
        send_super(MyApp, initialise, 'My application'),
        send(MyApp, append, new(D, dialog)),
        send(D, pen, 0),
        send(D, gap, size(5, 5)),
        send(D, append, new(menu_bar)),
        send(D, append, new(tool_bar(MyApp))),
        send(MyApp, fill_menu_bar),
        send(MyApp, fill_tool_bar),
        send(new(W, myapp_workspace), below, D),
        send(new(report_dialog), below, W).

fill_menu_bar(F) :->
        get(F, member, dialog, D),
        get(D, member, menu_bar, MB),
        send_list(MB, append,
                  [ new(File, popup(file)),
                    new(_Edit, popup(edit))
                  ]),

        send_list(File, append,
                  [ menu_item(load,
                              message(F, load),
                              end_group := @on),
                    menu_item(print,
                              message(F, print))
                  ]).

fill_tool_bar(F) :->
        get(F, member, dialog, D),
        get(D, member, tool_bar, TB),
        send_list(TB, append,
                  [ tool_button(load,
                                resource(floppy),
                                load),
                    gap,                % skip a little
                    tool_button(print,
                                resource(printer),
                                print)
                  ]).

print(MyApp) :->
        "Print the document"::
        send(MyApp, report, progress, 'Printing ...'),
        get(MyApp, member, myapp_workspace, WS),
        send(WS, print),
        send(MyApp, report, progress, done).

```

```
load(MyApp) :->
        "Ask a file and load it"::
        get(@finder, file, @on, myp, File),
        get(MyApp, member, myapp_workspace, WS),
        send(WS, load, File).

:- pce_end_class(myapp).


%       dummy class for the work-area of your application

:- pce_begin_class(myapp_workspace, window).

:- pce_end_class(myapp_workspace).
```

11.4 Library “pce_toc”: displaying hierarchies

The table-of-content library defines a `window` displaying a `tree` in an explorer-like style. This library is programmed by refining its base-class `toc_window`. We will introduce this library using an example exploring the filesystem. A screendump of this application is in figure 11.2.

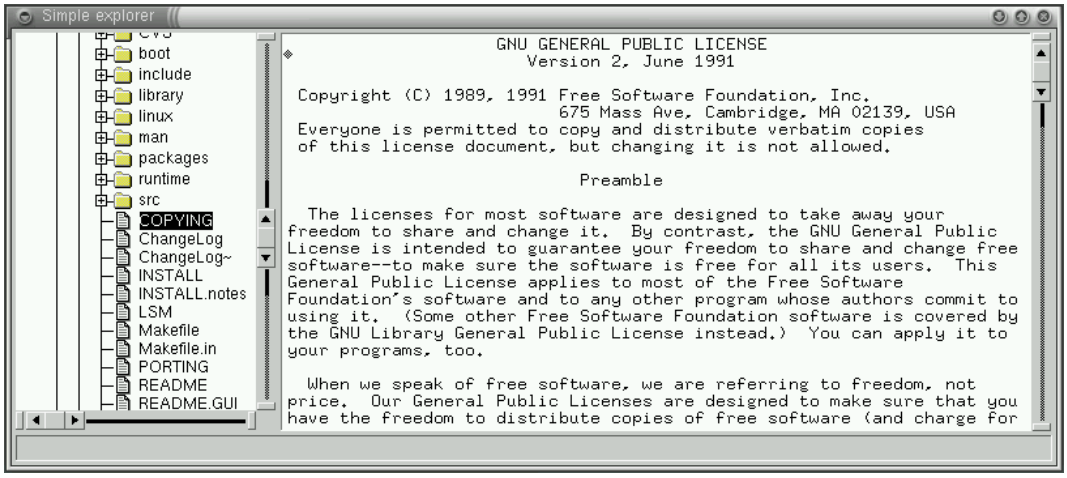


Figure 11.2: Exploring the filesystem

```
1  :- pce_autoload(toc_window, library(pce_toc)).
2  :- pce_autoload(report_dialog, library(pce_report)).
3
4  :- pce_begin_class(explorer, frame, "Explore the filesystem").
5
6  initialise(E, Dir:directory) :->
7          "Explore from directory"::
8          send_super(E, initialise, 'Simple explorer'),
9          send(E, append, new(DH, directory_hierarchy(Dir))),
10         send(new(view), right, DH),
11         send(new(report_dialog), below, DH).
12
13 open_node(E, Node:file) :->
14         "Show file content of opened node"::
15         get(E, member, view, View),
16         send(View, load, Node).
17
18 :- pce_end_class.
19
20
21 :- pce_begin_class(directory_hierarchy, toc_window,
22                    "Browser for a directory-hierarchy").
23
24 initialise(FB, Root:directory) :->
25         send(FB, send_super, initialise),
26         get(Root, name, Name),
27         send(FB, root, toc_folder(Name, Root)).
```

```
28
29 expand_node(FB, D:directory) :->
30         "Called if a node is to be expanded"::
31         new(SubDirsNames, chain),
32         new(FileNames, chain),
33         send(D, scan, FileNames, SubDirsNames),
34
35         get(SubDirsNames, map, ?(D, directory, @arg1), SubDirs),
36         send(SubDirs, for_all,
37              message(FB, son, D,
38                      create(toc_folder, @arg1?name, @arg1))),
39         get(FileNames, map, ?(D, file, @arg1), SubFiles),
40         send(SubFiles, for_all,
41              message(FB, son, D,
42                      create(toc_file, @arg1?base_name, @arg1))).
43
44 open_node(FB, Node:file) :->
45         "Called if a file is double-clicked"::
46         send(FB?frame, open_node, Node).
47
48 :- pce_end_class.
```

Programming is achieved by subclassing `toc_window` and in some cases the support classes `toc_folder` and `toc_file`, representing expandable and leaf-nodes.

Each node is assigned an *identifier*, a unique reference to the node. In the example below we used `file` and `directory` objects for this purpose. The identifier is the second argument to the creation of the node. When omitted, the node is turned into an identifier of itself. This distinction is used to hide the existence of graphical node objects for users of the basic functionality.

Below we describe the important methods of this package. We start with the virtual methods on class `toc_window` that should be refined by most applications.

toc_window → expand_node: Id:any
: The user clicked the `[+]` sign or double-clicked a `toc_folder`. This method is normally refined to add sub-nodes for *Id* to the current node using '`toc_window → son`'. If the implementation of `toc_window` is activated at the end the window will scroll such that as much as possible of the subtree below *Id* is visible.

toc_window → open_node: Id:any
: Called on double-click on a `toc_file` node. The implementation of `toc_window` is empty.

toc_window → select_node: Id:any
: Called after single-click on `toc_folder` or `toc_file`. Note that double-clicking activates both →`select_node` and →`open_node` and therefore the action following select_node should execute quickly.

toc_window ← popup: Id:any → Popup:popup
: This method is called on a right-down. If it returns a `popup` object this is displayed.

The methods below are used for general querying and manipulation of the hierarchy.

toc_window ← selection→ ChainOfNodes
Returns a `chain` holding the `node` objects that are currently selected.

toc_window ← node: Id:any → Node:toc_node
Map a node-identifier to a node. Fails silently if this identifier is not in the tree.

toc_window → root: Root:toc_folder
Assign the hierarchy a (new) root.

toc_window → son: Parent:any, Son:toc_node
Make a new node below the node representing *Parent*. If the node is a leaf, *Son* is a subclass of `toc_file`, otherwise it is a subclass of `toc_folder`.

toc_window → expand_root
Expands the root-node. This is normally called from →`initialise` to get a sensible initial hierarchy.

The classes `toc_folder` and `toc_file` are summarised below. Subclassing may be used to modify interaction and/or store additional information with the node.

toc_folder → initialise: Label:char_array, Id:[any],
CollapsedImg:[image], ExpandedImg:[image],
CanExpand:[bool]
Create an expandable node. *Id* defaults to the node object itself and the two images to the standard opened/closed folder images. Folders assume they can be expanded, *CanExpand* may be set to `@off` to indicate ‘an empty folder’.

toc_file → indicate: Label:char_array, Id:[any], Img:[image]
Create a ‘file’-node from its *Label*, *Id* (defaults to the created node object) and *Image* (defaults to a ‘document’ icon).

11.5 Tabular layout

XPCE provides various mechanisms for two-dimensional layout.

- *Controller layout using* ‘`device → layout_dialog`’
  This method is used by the classes `dialog` and `dialog_group`. It knows about layout requirements in controller windows, such as alignment of label- and value-width in stacked controllers. etc. Layout of controllers is described in chapter 4.

- *Simple tabular layout using* `format`
  In instance of class `format` can be attached to a `device` using ‘`device → format`’. This causes the device to place its graphicals according to the specification in the format object. This technique is frequently used to label images, place images in an image browser, etc. See section 11.5.1.

- *Full table support using* `table`
  An instance of `table` can be associated with a `device` to realise modern tables using the same primitives as defined in HTML-3: row- and column spanning, alignment, spacing, rules and borders, etc. The library library(`tabular`) provides a user-friendly front-end for most of the functionality of class `table`.

11.5.1 Using `format`

Class `format` can be seen as a poor-mens version of `table`. On the other hand, there are two cases that still make it a valuable solution. One is to deal with simple compound graphicals, such as a bitmap with a label displayed below it. The other is for browsing collections of graphical objects such as images.

The class `icon` below displays a label-text below an image.

```
1  :- pce_begin_class(icon, device).
2
3  :- pce_global(@icon_format, make_icon_format).
4
5  make_icon_format(F) :-
6          new(F, format(horizontal, 1, @on)),
7          send(F, adjustment, vector(center)),
8          send(F, row_sep, 2).
9
10 initialise(Icon, Img:image, Label:name) :->
11         send_super(Icon, initialise),
12         send(Icon, format, @icon_format),
13         send(Icon, display, bitmap(Img)),
14         send(Icon, display, text(Label, center)).
15
16 :- pce_end_class.
```

An example using `format` for distribution graphicals over a window is the library library(`pce_image_browser`).

11.5.2 Using `table` using the "tabular" library

The class `table` acts much the same way as class `format` in the sense that it is attached to a `device` and modifies the layout management of this device. For this purpose it uses an interface defined in XPCE version 5.0 and realised using the methods '`device` ⇆ `layout_manager`' and '`graphical` ⇆ `layout_interface`'. Figure 11.3 gives an overview of the classes realising tabular layout.

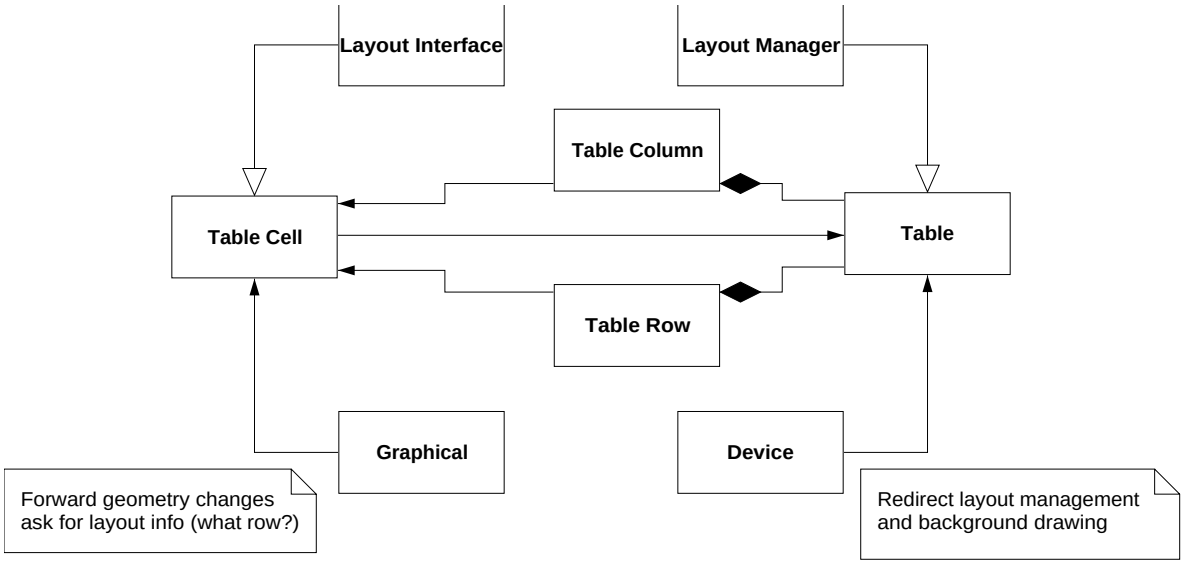


Figure 11.3: Layout manager interface for tables

The advantage of the approach using layout manager objects is that they can easily be associated with any subclass of `device`, such as a `window`. The disadvantage is that the communication gets more difficult due to the different objects involved. This complication is hidden in the XPCE/Prolog class `tabular`, a subclass of `device` with an associated `table` and methods for guiding the messages for common usage.

tabular → initialise
Create a device with associated table.

tabular → append: Label:name | graphical, Font:[font],
HAlign:[{left,center,right}],
Valign[{top,center,bottom}], Colspan:[1..],
Rowspan:[1..], Background:[colour],
Colour:[colour]
Append a new cell. Cells are places left-to-right. The arguments are:

- Label
  Defines the content. It this is a `name`, a `text` is created. Otherwise the graphical is immediately placed in the table.
- Font
  Defines the `font` if a `text` is created from a *Label* of type `name`.

- HAlign
  Horizontal alignment. When omitted, the value from the corresponding `table_column` is used.
- VAlign
  Vertical alignment. When omitted, the value from the corresponding `table_row` is used.
- Colspan
  Number of columns spanned. Default is 1.
- Rowspan
  Number of rows spanned. Default is 1.
- Background
  Colour or pattern used to fill the background of the cell. When omitted, the value from row or column is used or the background is left untouched.
- Colour
  Defines the default foreground colour when painting the cell's graphical. When omitted, the row, column and finally device are tried.

tabular → append_label_button: Field:name
This method appends a button that is nicely aligned with the cell. If the button is depressed it activates →`sort_rows`, providing the column index and the row below the row holding the button.

tabular → sort_rows: Col:int, Row:int
A virtual method with a body that prints an informative message. It is called from a button installed using →`append_label_button` and receives the column to sort-on as well as the first row to sort.

tabular ← table→ Table
Returns the `table` serving as ←`layout_manager`.

tabular → table_width: Width:int
Force the table to perform the layout in the indicated width. Initially the width of a `tabular` is defined by the content. Setting the width forces the `table` to negotiate with its columns and then force the width of the columns.

tabular → event: Event:event
This refinement of '`device → event`' determines the cell in which the event occurs. If this cell has a '`cell ⇆ note_mark`' attached and the graphical defines the method →`on_mark_clicked`, the event is checked against the mark-image. Otherwise the event is forwarded to the graphical inside the cell, even it it doesn't occur in the area of the graphical, making small (text-)objects sensitive to all events in the cell. Finally, this method checks for attempts to drag the column-borders, changing the layout of the table.

As `tabular` *delegates* all messages not understood to the ←`table`, the messages of this class are also available. Below are the most important ones.

table → next_row: EndGroup:[bool]
Start the next row in the table. If *EndGroup* is @on, the just-finished row is marked to end a row-group. See also →`rules`.

table → border: Border:0..
Defines the thickness of border and rule-lines. Default is 0, not drawing any lines.

table → frame: {**void,above,below,hsides,vsides,box**}
Defines which parts of the box around the table are painted (if ←`border` > 0). The terminology is from HTML-3.

table → rules: {**none,groups,rows,cols,all**}
Defines which lines between rows/columns are painted (if ←`border` > 0). The terminology is from HTML-3.

table → cell_padding: Padding:int|size
Defines the space around the content of a cell. If this is an integer this space is the same in horizontal and vertical directions. If it is a `size` these can be specified independently.

table → cell_spacing: Spacing:int|size
Defines the distance between the cells. Same rules as for →`cell_padding` applies. In some cases pretty effects can be achieved setting this value to minus the ←`border`.

Below we build a small example.

```
:- use_module(library(tabular)).
:- use_module(library(autowin)).

make_table :-
        new(P, auto_sized_picture('Table with merged cells')),
        send(P, display, new(T, tabular)),
        send(T, border, 1),
        send(T, cell_spacing, -1),
        send(T, rules, all),
        send_list(T,
                  [ append(new(graphical), rowspan := 2),
                    append('Length/mm', bold, center, colspan := 3),
                    next_row,
                    append('body', bold, center),
                    append('tail', bold, center),
                    append('ears', bold, center),
                    next_row,
                    append('males', bold),
                    append('31.4'),
                    append('23.7'),
                    append('3.8'),
                    next_row,
                    append('females', bold),
                    append('29.6'),
                    append('20.8'),
                    append('3.4')
```

```
27                      ]),
28          send(P, open).
```

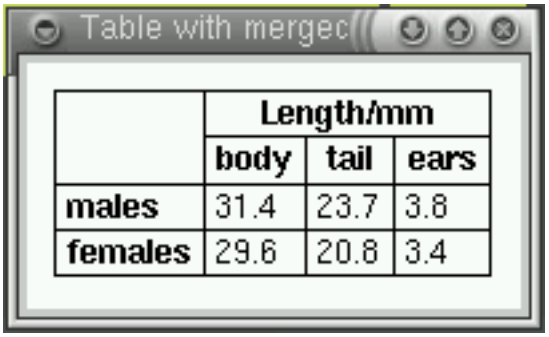


Figure 11.4: Small table with row/column spanning

11.6 Plotting graphs and barcharts

This section describes three libraries residing in ⟨*pcehome*⟩`/prolog/lib/plot` to deal with plotting graphs and barcharts.

11.6.1 Painting axis

The library library(`plot/axis`) defines the class `plot_axis` to draw an X- or Y-axis. The class deals with computing the layout, placing rule-marks, values and labels as well as translation between coordinates and real values. Normally this class is used together with `plotter`, `plot_axis` does not rely on other library classes and may therefore be used independent of the remainder of the plotting infrastructure.

We start with a small example from the library itself, creating the picture below.

```
?- [library('plot/axis')].
% library('plot/axis') compiled into plot_axis 0.03 sec, 27,012 bytes

?- send(new(P, picture), open),
           send(P, display,
                plot_axis(x, 0, 100, @default, 400, point(40, 320))),
           send(P, display,
                plot_axis(y, 0, 500, @default, 300, point(40, 320))).
```

Below is a reference to the important methods of this class. The sources to the class itself are a good example of complicated and advanced layout computations and delaying of these until they are really needed.

plot_axis → initialise: type=x,y, low=int|real, high=int|real,
step=[int|real], length=[int], origin=[point]

Create a new axis. *type* defines whether it is an X- or Y-axis. The axis represents values in the range [*low*...*high*]. If *step* is specified, a rule-mark with value is placed at these intervals. Otherwise the library computes its marking dynamically. The *length* argument specifies the length of the axis in pixels, the default is 200 and finally the *origin* defines the pixel-location of the origin.

plot_axis → label: graphical*

Label to position near the end of the axis. This is a graphical to provide full flexibility.

plot_axis → format: [name]

Define the printf()-format for rendering the values printed along the axis.

plot_axis ← location: int|real → int

Determine the coordinate in the device's coordinate system representing the given value. See also '`plotter ← translate`'.

plot_axis ←
value_from_coordinate: int → int|real

The inverse of ←`location`, returning the value along the axis from a pixel coordinate.

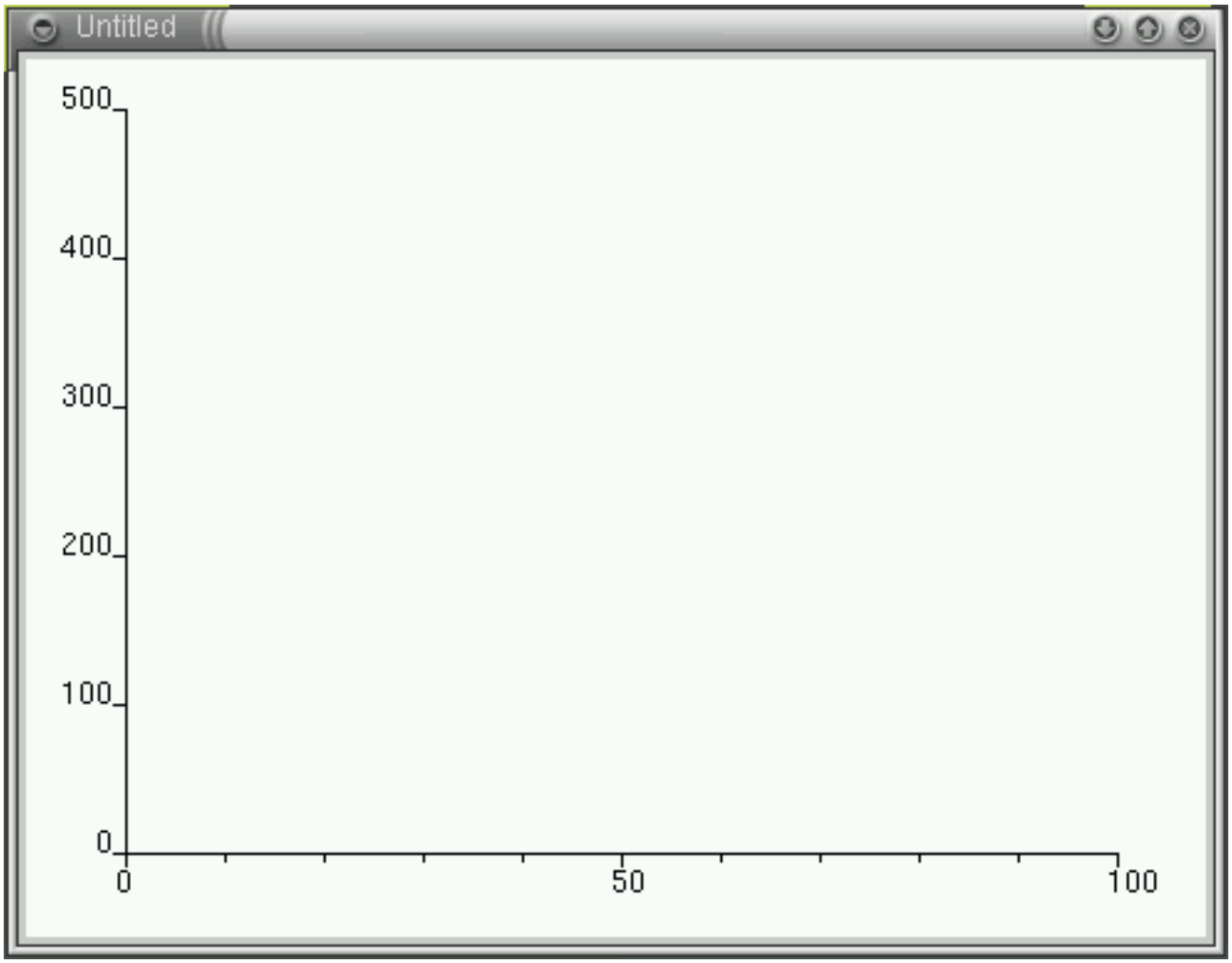


Figure 11.5: A picture showing two axis

Besides the principal methods below, the following methods are available for changing attributes of an existing axis: →`origin`, →`low`, →`high`, →`step`, →`small_step` (interval for rule-marks without a value), →`length` and →`type`: {x,y}.

11.6.2 Plotting graphs

The library library(`plot/plotter`) defines the classes `plotter` and `plot_graph` for displaying graphs. Class `plotter` is a subclass of `device`. The example below plots the function $Y = sine(X)$

```
:- use_module(library('plot/plotter')).
:- use_module(library(autowin)).

plot_function :-
        plot_function(X:sin(X)).

plot_function(Template) :-
        To is 2*pi,
        PlotStep is To/100,
        Step is pi/4,
        new(W, auto_sized_picture('Plotter demo')),
        send(W, display, new(P, plotter)),
        send(P, axis, new(X, plot_axis(x, 0, To, Step, 300))),
        send(P, axis, plot_axis(y, -1, 1, @default, 200)),
        send(X, format, '%.2f'),
        send(P, graph, new(G, plot_graph)),
        plot_function(0, To, PlotStep, Template, G),
        send(W, open).

plot_function(X, To, _, _, _) :-
        X >= To, !.
plot_function(X, To, Step, Template, G) :-
        copy_term(Template, X:Func),
        Y is Func,
        send(G, append, X, Y),
        NewX is X + Step,
        plot_function(NewX, To, Step, Template, G).
```

plotter → axis: plot_axis
Associate a `plot_axis`. Before using the plotter both an X and Y axis must be associated. Associating an axis that already exists causes the existing axis to be destroyed.

plotter → graph: plot_graph
Append a graph. Multiple graphs can be displayed on the same plotter.

plotter → clear
Remove all graphs. The X- and Y-axis are not removed.

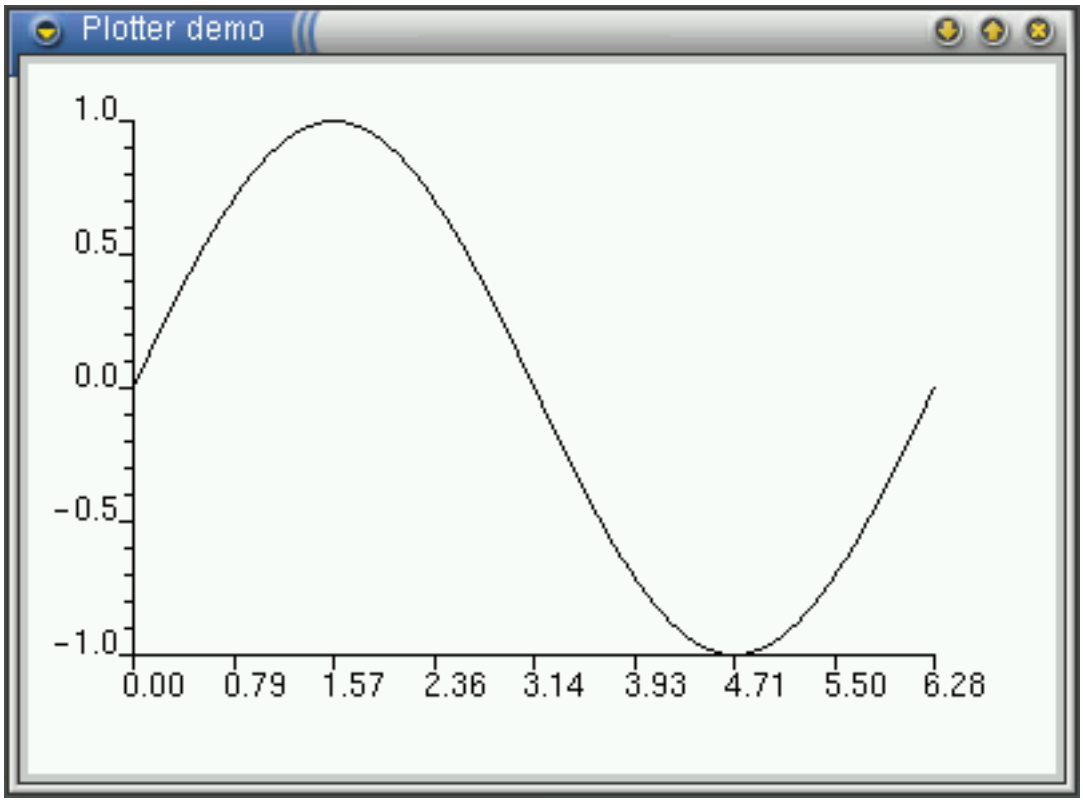


Figure 11.6: Plotter showing sine function

plotter ← translate: X:int|real, Y:int|real → point
Translate a coordinate in the value-space to physical coordinates.

plotter ← value_from_x: int → int|real
Translate an X-coordinate to a value.

plotter ← value_from_y: int → int|real
Translate an Y-coordinate to a value.

Graphs themselves are instances of class `plot_graph`, a subclass of `path`. Instead of normal `point` objects, the points are represented using the subclass `plot_point` that attaches the real values to the physical coordinates. Methods:

plot_graph → initialise: type=[{poly,smooth,points_only}],
mark=[image]*
The *type* argument denotes the interpolation used. Using `poly` (default), straight lines are drawn between the points. Using `smooth`, the curve is interpolated (see `path` for details) and using `points_only`, no lines is painted, just the marks. Using the *mark* argument the user may specify marks to be drawn at each control-point.

plot_graph → append: x=int|real, y=int|real
Append a control-point using the coordinate-system of the axis of the plotter.

11.6.3 Drawing barcharts using "plot/barchart"

The library(`plot/barchart`) library draws simple bar-charts. It is based on the `plotter` and `plot_axis` classes, adding simple bars, grouped bars and stacked bars. Below is an example from library(`plot/demo`) showing all active XPCE, classes, where active is defined that more than 250 instances are created. The code, except for the calculation parts is show below.

```
1  barchart :-
2          barchart(vertical).
3  barchart(HV) :-
4          new(W, picture),
5          active_classes(Classes),
6          length(Classes, N),
7          required_scale(Classes, Scale),
8          send(W, display, new(BC, bar_chart(HV, 0, Scale, 200, N))),
9          forall(member(class(Name, Created, Freed), Classes),
10                send(BC, append,
11                     bar_group(Name,
12                               bar(created, Created, green),
13                               bar(freed, Freed, red)))),
14         send(W, open).
```

bar_chart → initialise: orientation={horizontal,vertical}, low=real,
high=real, scale_length=[0..], nbars=[0..]
Initialise a `bar_chart`, a subclass of `plotter` for displaying bar-charts. The *orientation* indicates whether the bars are vertical or horizontal. The *low* and *high* arguments

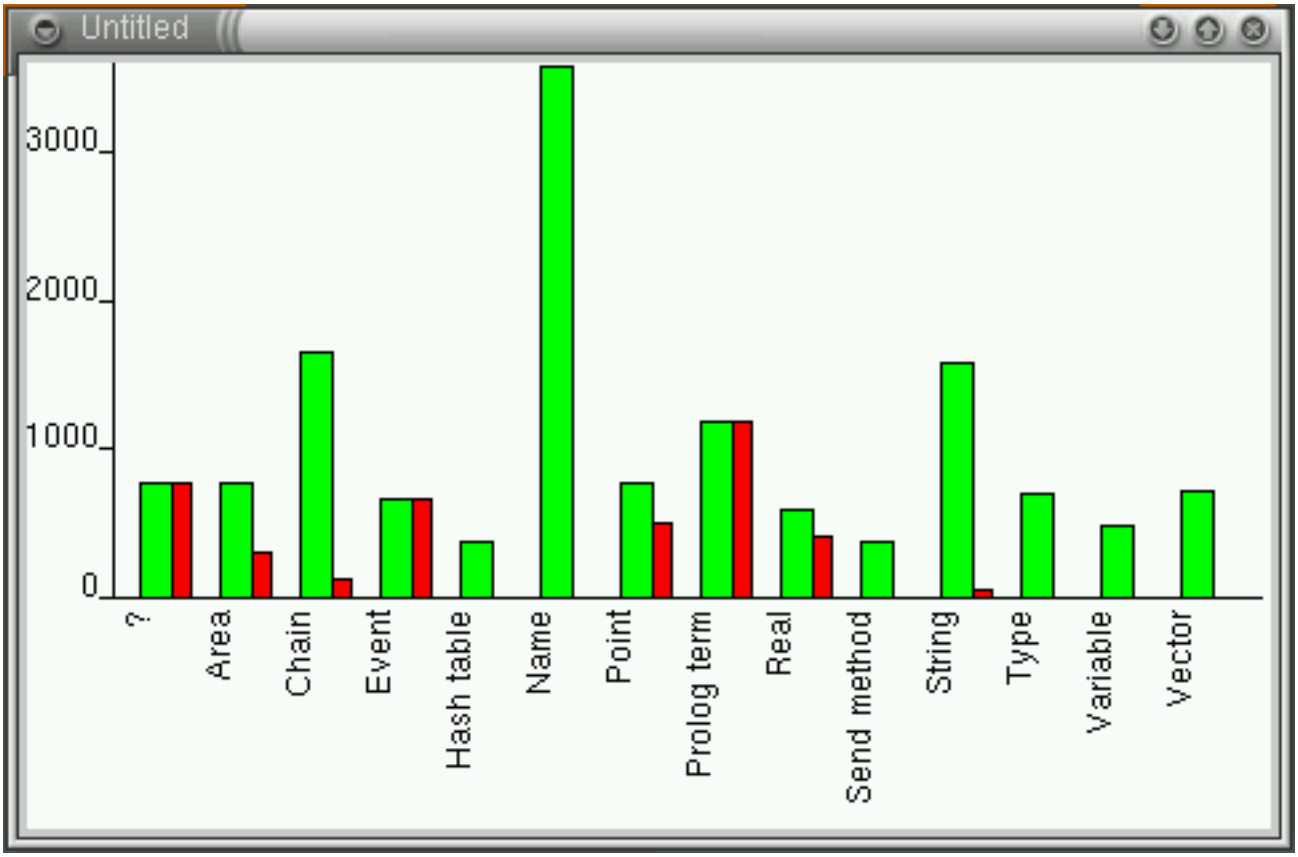


Figure 11.7: Classes of XPCE with > 250 instances created

are the scale arguments for the value-axis, while *scale_length* denotes the length of the axis. The *nbars* argument determines the length of the axis on which the bars are footed.

bar_chart → append: bar|bar_stack
Append a single `bar`, `bar_stack` or `bar_group` to the chart. Bars and bar-stacks are named and can be addressed using their name.

bar_chart → delete: member:bar|bar_stack
Remove the given bar. The `member:` construct makes the type-conversion system translate a bar-name into a bar. If the bar is somewhere in the middle, the remaining bars are compacted again.

bar_chart → clear
Removes all bars from the chart.

bar_chart ⇆ value: name
real Modifies or requests the value of the named bar. Fails if no bar with this name is on the chart.

bar_chart → event: event
Processes a single-click outside the bars to clear the selection.

bar_chart → select: bar|bar_stack, [{toggle,set}]

bar_chart → selection: bar|bar_stack|chain*

bar_stack ← selection→ chain
Deal with the selection. Selection is visualised by selecting the labels, but communicated in terms of the bars themselves.

Class `bar`

Bars can either be displayed directly on a `bar_chart`, or as part of a stack or group. Stacked bars are used to indicate composition of a value, while grouped bars often indicate development or otherwise related values for the same object.

bar → initialise: name=name, value=real, colour=[colour],
orientation=[{horizontal,vertical}]
Create a `bar` from its *name* and *value*. The bar itself is a subclass of `box` and *colour* is used to fill the interior. The *orientation* needs only be specified if the bar is not attached to a `bar_chart`.

bar → value: real
Set the value of th bar. This updates the bar-size automatically.

bar → range: low=real, high=real
If the bar is editable (see also →`message` and →`drag_message`), these are the lowest and highest values that can be set by the user.

bar → message: code*
: If not `@nil`, the bar may be modified by dragging it. After releasing the mouse-button, the new ←`value` is forwarded over the `code`.

bar → drag_message: code*
: If not `@nil`, the bar may be modified by dragging it. While dragging, the new value is forwarded on every change over the `code`. It is allowed to specify both →`message` and →`drag_message`.

bar_stack → initialise: name, bar ...
: Create a pile of bars representing a value composed of multiple smaller values.

bar_group → initialise: name, bar ...
: Same as `bar_stack`, but places the bars next to each other instead of stacked.

Class `bar_button_group`

A subclass of `dialog_group` that can be used to associate one or more buttons or other controllers with a `bar` or `bar_stack`. This association is achieved by simply creating an instance of this class. Figure 11.8 shows both associated buttons and a stacked bar.

bar_button_group → initialise: bar|bar_stack, graphical ...
: Associate the given graphicals with given bar.

bar_button_group ← bar→ bar|bar_stack
: Return the bar or stack this group is connected too. This behaviour may be used to make to make the buttons less dependent on the bar they are attached too.

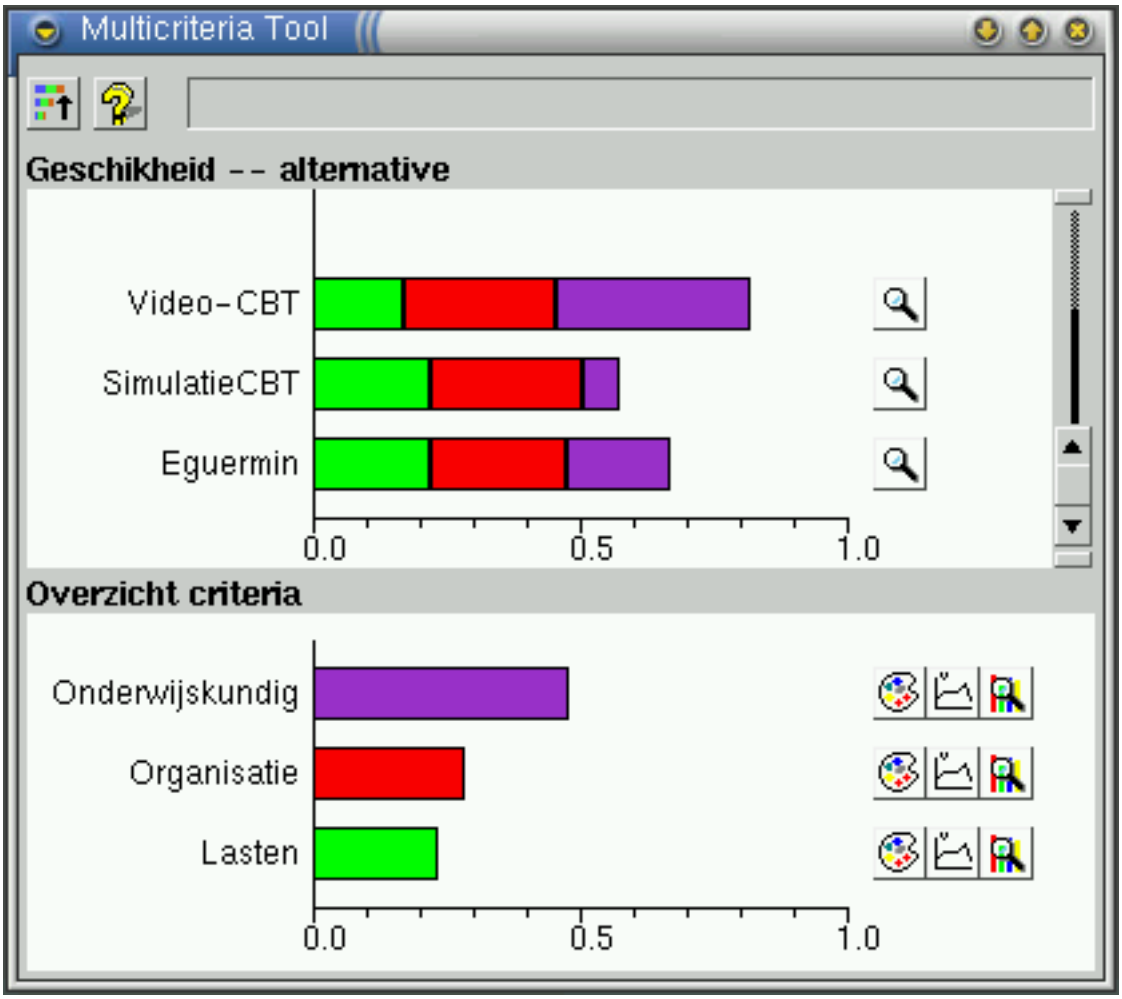


Figure 11.8: Stacked bars with associated buttons

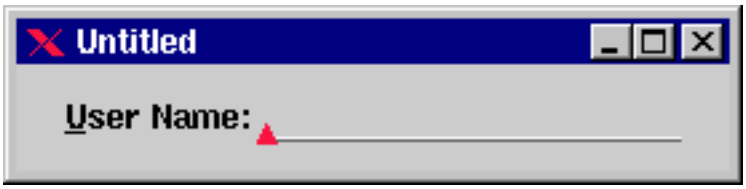
11.7 Multi-lingual applications

XPCE provides some support for building multi-lingual applications without explicitly mapping terms all the time. This section provides an overview of how multi-lingual operation was realised in a simulator for optics.

When writing a multi-lingual application, several different types of information needs to be translated. We will discuss each of them below.

- *Labels*
  Labels as they are used by the subclasses of `dialog_item`, menu items, etc. These can be mapped by redefining a number of methods that realise the default mapping between internal names and externally visible names:

  dialog_item ← label_name: Id → Label
  This method performs the mapping that ensures that the code `text_item(user_name, '')` renders as:

  This method may be redefined to return another name or image object, depending on the current language mapping.

  menu_item ← label_name: Id → Label
  Similar to '`dialog_item ← label_name`'.

  dialog_group ← label_name: Id → Label
  Similar to '`dialog_item ← label_name`', but is, in the current implementation, not allowed to return an `image`. This method needs to be redefined separately as `dialog_group` (a super-class of `tab`) is not in the same branch of the inheritance hierarchy as `dialog_item`.

  In the current implementation, window and frame labels are not covered by this schema.

- *Error messages*
  Although it is convenient to present error messages directly using the report mechanism described in section 10.7, this approach is not very well suited for multi-lingual applications. A better approach is to use `error` objects, as described in section 10.8.

  Using error objects is slightly more cumbersome as errors need to be declared separately, but they improve the embedding possibilities using error handling, and the mapping from an error identifier to a format string provides the indirection necessary in multi-lingual applications.

- *Other messages and help text*
  There is no special support for other textual information, help-texts, etc.

Below is a summary of the file `language.pl` as using in the optics simulator to reach at an English/Dutch application.

```
:- module(language,
          [ message/2,                  % Id, Message
            current_language/1,         % -Language
            set_language/1              % +Language
          ]).
:- use_module(pce).
:- use_module(configdb).
:- require([ concat_atom/2
           , is_list/1
           , memberchk/2
           ]).

:- dynamic
        current_language/1.

current_language(english).              % the default
%current_language(dutch).

set_language(Lan) :-
        retractall(current_language(_)),
        assert(current_language(Lan)),
        make_errors.

%       message(+Term, -Translation)
%       The heart of the translator. Map a term
%       (normally an atom, but if can be an arbitrary
%       Prolog term, into an image or atom. If no
%       translation is found, the default case and
%       underscore translation is performed.

message(Term, Translation) :-
        current_language(Lan),
        term(Term, Translations),
        (   is_list(Translations)
        ->  T =.. [Lan, Translation0],
            memberchk(T, Translations),
            (   is_list(Translation0)
            ->  concat_atom(Translation0, Translation)
            ;   Translation = Translation0
            )
        ;   Translation = Translations
        ), !.
message(Term, Translation) :-
        get(Term, label_name, Translation).


                /*******************************
                *    MAP DIALOG IDENTIFIERS    *
                *******************************/
```

```
50
51 :- pce_extend_class(dialog_item).
52
53 label_name(DI, Id:name, Label:'name|image') :<-
54         "Multi-lingual label service"::
55         message(Id, Label0),
56         (   atomic(Label0)
57         ->  get(DI, label_suffix, Suffix),
58             get(Label0, ensure_suffix, Suffix, Label)
59         ;   Label = Label0
60         ).
61
62 :- pce_end_class.
63
64 :- pce_extend_class(dialog_group).
65
66 label_name(_DI, Id:name, Label:name) :<-
67         "Multi-lingual label service"::
68         (   message(Id, Label),
69             atomic(Label)
70         ->  true
71         ;   get(Id, label_name, Label)
72         ).
73
74 :- pce_end_class.
75
76 :- pce_extend_class(menu_item).
77
78 default_label(_MI, Id:name, Label:'name|image') :<-
79         "Multilingual label service"::
80         message(Id, Label).
81
82 :- pce_end_class.
83
84                  /*******************************
85                  *        GENERIC LABELS        *
86                  *******************************/
87
88 %       term(+Term, -Translated)
89 %
90 %       Term translates a term. There are three examples
91 %       here. The first only contains the translation
92 %       for an English label name into a Dutch one. The
93 %       second replaces all labels named label' into an'
94 %       image. The last is for generating a more
95 %       elaborate message from an identifier.
96
97 term(settings,
98      [ dutch('Instellingen')
99      ]).
100 term(label,
101      image('label.lbl')).
102 term(start_named_test(Name),
```

```
103     [ english([’Click "OK" to start test "’, Name, ’"’]),
104       dutch([’Klik op "OK" om aan de toets "’, Name,
105              ’" te beginnen’])
106     ]).
107
108
109                 /*******************************
110                 *             ERRORS           *
111                 *******************************/
112
113 %       error(Id, Kind, Translations)
114 %
115 %       Specify and create the required error messages.
116 %       An object that detects there are too many
117 %       instruments directs this information to the user
118 %       by
119 %
120 %               ...
121 %               send(MySelf, error, max_instruments, 5),
122 %               ...
123
124 error(max_instruments, error,
125       [ dutch(’%IU kunt niet meer dan %d van deze \
126                instrumenten gebruiken’),
127         english(’%IYou can not use more than %d of \
128                  these instruments’)
129       ]).
130
131 make_errors :-
132         current_language(Lan),
133         T =.. [Lan, Message],
134         error(Id, Kind, Messages),
135         (   memberchk(T, Messages)
136         ->  true
137         ;   Message = Id
138         ),
139         new(_E, error(Id, Message, Kind, report)),
140         fail.
141 make_errors.
142
143 :- initialization make_errors.
```

11.8 Drag and drop interface

XPCE's drag-and-drop interface allows the user to drag-and-drop `graphical` objects and `dict_item` objects. Drag-and-drop is a common GUI technique to specify operations that require two objects in a specific order. Moving files to another directory by dragging them to the icon of the target directory is a common example.

It may also be used to specify operations on a single object, where the operation is represented by an icon. Dragging files to a trash-bin is the most commonly found example.

For the drag-and-drop interface to work, the programmer must connect a `drag_and_drop_gesture` to the object to be dragged.[1]. A *Drop-zone* defines the method →`drop` and (not strictly obligatory) →`preview_drop`. →`drop` is called to actually perform the associated operation, while →`preview_drop` may be used to indicate what will happen if the object is dropped now.

Below is a complete example that allows the user to drag objects for moving and copying on another window.

Class drop_picture defines a graphical window that imports graphical objects when they are dropped onto it. The feedback is a dotted rectangle indicating the area of the graphical to be imported. See 'graphical →`preview_drop`' for a description of the arguments.

```
:- pce_begin_class(drop_picture, picture).

preview_drop(P, Gr:graphical*, Pos:[point]) :->
        (   Gr == @nil                  % pointer leaves area
        ->  (   get(P, attribute, drop_outline, OL)
            ->  send(OL, free),
                send(P, delete_attribute, drop_outline)
            ;   true
            )
        ;   (   get(P, attribute, drop_outline, OL)
            ->  send(OL, position, Pos)
            ;   get(Gr?area, size, size(W, H)),
                new(OL, box(W, H)),
                send(OL, texture, dotted),
                send(P, display, OL, Pos),
                send(P, attribute, drop_outline, OL)
            )
        ).
```

The method →`drop`. If the graphical originates from the same picture just move it. Otherwise ←`clone` the graphical and display the clone.

```
drop(P, Gr:graphical, Pos:point) :->
        (   get(Gr, device, P)
        ->  send(Gr, position, Pos)
        ;   get(Gr, clone, Gr2),
            send(P, display, Gr2, Pos)
        ).

:- pce_end_class.
```

[1] Attach a `drag_and_drop_dict_item_gesture` to a `list_browser` to enable dragging the items in the dictionary

Class dragbox defines a simple subclass of class box that can be resized and dragged.

```
:- pce_begin_class(dragbox, box).
:- pce_autoload(drag_and_drop_gesture, library(dragdrop)).
:- pce_global(@dragbox_recogniser, make_dragbox_recogniser).
make_dragbox_recogniser(G) :-
        new(G, handler_group(resize_gesture(left),
                             drag_and_drop_gesture(left))).
event(B, Ev:event) :->
        (   send(B, send_super, event, Ev)
        ;   send(@dragbox_recogniser, event, Ev)
        ).
:- pce_end_class.
```

The toplevel predicate creates two drop_pictures in one frame (note that drag-and-drop-gestures work accross frames, but not accross multiple XPCE processes at the moment). It displays one dragbox in one of the windows. Dragging it inside a picture moves the box, dragging it to the other windows makes a copy of the box.

```
dragdropdemo :-
        new(F, frame('Drag and Drop Demo')),
        send(F, append, new(P1, drop_picture)),
        send(new(drop_picture), right, P1),
        send(P1, display, dragbox(100, 50), point(20,20)),
        send(F, open).
```

11.8.1 Related methods

drag_and_drop_gesture →
initialise: Button, Modifier, Warp, GetSource

Initialises a new `drag_and_drop_gesture`. *Button* is the name of the pointer-button the gesture should be connected to (left, middle or right). *Modifier* is a `modifier` description (see class `modifier`). *Warp* is for compatibility with older releases of this library. *GetSource* is a function used to fetch the object dragged from the graphical representing it. Suppose the graphical to which the gesture is attached represents a database record. In this case it is much more natural to pass the identifier for the database record to the →`drop` and →`preview_drop` methods than to pass the icon representing it. *GetSource* is a function that is evaluated with `@arg1` bound to the graphical when the gesture is activated. An example could be:

```
drag_and_drop_gesture(left,
                      get_source :=
                             @arg1?db_record)
```

graphical → drop: Object:⟨*Type*⟩ [, Pos:point]

This method may be defined on any graphical object that is a drop-zone. It will only be activated if the drag_and_drop_gesture can locate the method and make the necessary type transformations. Thus, if the type is specified as `file`, this method will only be activated if the dragged object can be converted to a `file` object. See also the discussion about the get_source argument above.

If the method accepts a point for the second argument, a point will be passed that represents the location of the pointer in the coordinate system of the drop-zone, subtracted by the distance between the top-left corner of the dragged graphical to the pointer at the moment the button was depressed. To get the position of the pointer itself, just ask for the position of `@event` relative to the drop-zone.

graphical → preview_drop: Object:⟨*Type*⟩* [, Pos:[point]]

Sent by the `drag_and_drop_gesture` to allow the drop-zone providing feedback. The arguments and semantics are the same as for →`drop`, but the first argument can be `@nil`, indicating that the mouse has left the drop-zone. Under this condition, the position argument is `@default`.

If a position argument is available, the `drag_and_drop_gesture` will be activated on each `drag` event. Otherwise it is only activated if the pointer enters the area of the drop-zone.

11.9 Playing WEB (HTTP) server

Web presentation has attractive features. It is well accepted, standardised (if you stick to the basics) and network-transparent. Many people think you need a web-server like Apache with some sort of server-scripting (CGI) to realise a server. This is not true. Any application capable of elementary TCP/IP communication can easily act as a web-server.

Using XPCE for this task may be attractive for a number of reasons.

- *Prototyping*
  As the XPCE/Prolog running on your desktop is the server you can use the full debugging capabilities of Prolog to debug your server application.
- *Including graphics*
  XPCE can generate GIF and JPEG images for your web-pages on the fly. You can include XPCE `graphical` objects directly in the output and have the server library handle the required transformations.
- *Remote presentation*
  XPCE can be used as groupware server, presenting state of the applications and allowing remote users to interact using their web-browser[2]
- *Report generation*
  Applications may to use HTML as framework for report generation. Though rather weak in its expressiveness, the advantage is the wide support on presentation and distribution applications.

We start with a small demo, illustrating frames and text.

```
:- module(my_httpd,
          [ go/1
          ]).
:- use_module(library(pce)).
:- use_module(library('http/httpd')).
:- use_module(library('http/html_write')).
:- use_module(library('draw/importpl')).
%       Create server at Port
go(Port) :-
        new(_, my_httpd(Port)).
:- pce_begin_class(my_httpd, httpd, "Demo Web server").
```

→`request` is sent after the super-class has received a complete request header. We get the 'path' and have a Prolog predicate generating the replies.

```
request(HTTPD, Request:sheet) :->
        "A request came in."::
        get(Request, path, Path),
        reply(Path, HTTPD).
:- discontiguous
        reply/2.
```

[2] Using the Unix/X11 version XPCE can manage windows on multiple desktops. For MS-Windows users this is not supported.

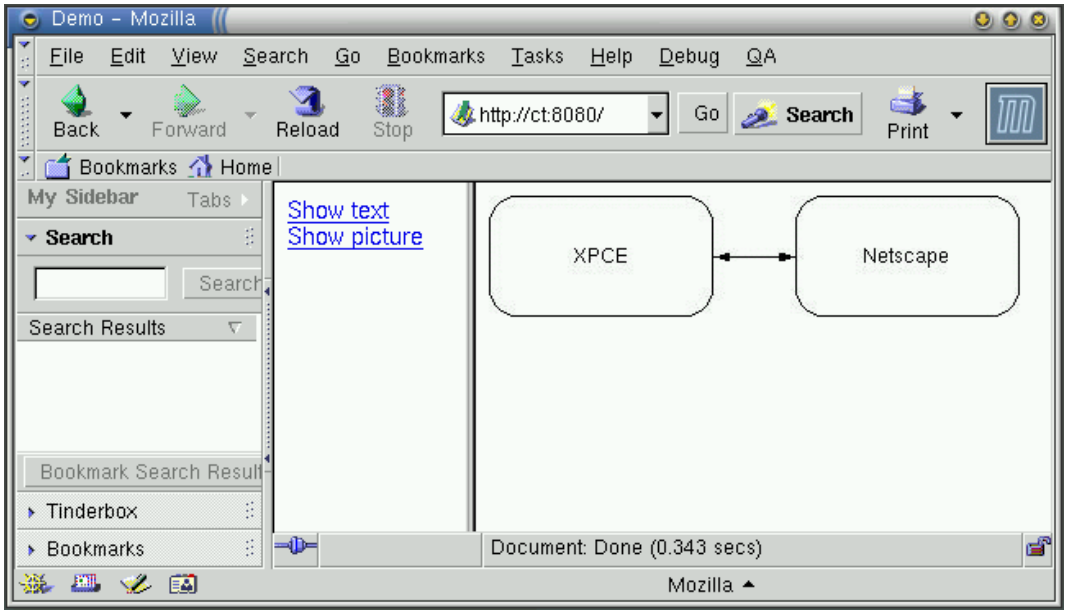


Figure 11.9: Mozilla showing XPCE generated figure

→`reply_html` takes ⟨*Module*⟩:⟨*DCGRuleSet*⟩ to formulate a reply. This uses the html_write library, converting a complex Prolog term into a formatted HTML document. The complex term can invoke additional DCG rulesets, providing nicely structured content-generation.

```
reply('/', HTTPD) :- !,
        send(HTTPD, reply_html, my_httpd:frames).

frames -->
        html(html([ head(title('Demo')),
                    frameset([cols('25%,75%')],
                             [ frame([ src('/index'),
                                       name(index)
                                     ]),
                               frame([ src('/blank'),
                                       name(body)
                                     ])
                             ])
                  ])).

reply('/blank', HTTPD) :-
        send(HTTPD, reply_html, my_httpd:blank).

blank -->
        page(title('Blank'),
             []).

reply('/index', HTTPD) :-
        send(HTTPD, reply_html, my_httpd:index).

index -->
        page(title('Index'),
             [ a([ href('/text'), target(body) ],
                 [ 'Show text' ]),
```

```
                br([]),
                a([ href('/picture'), target(body) ],
                  [ 'Show picture' ])
              ]).
reply('/text', HTTPD) :-
        send(HTTPD, reply_html, my_httpd:text).
text -->
        page(title('Text'),
             [ p(['Just showing a little text'])
             ]).
```

Reply a graphical object. The server translates the graphical to a GIF or JPEG bitmap and provides the proper HTTP reply header. You can also embed graphicals into the HTML structures used above.

The drawing itself is exported from the demo program PceDraw and turned into an XPCE graphical using the support library draw/importpl.

```
reply('/picture', HTTPD) :-
        make_picture(Gr),
        send(HTTPD, reply, Gr, 'image/gif').
make_picture(Dev) :-
        new(Dev, device),
        drawing(xpcenetscape, Drawing),
        realise_drawing(Dev, Drawing).
%       Drawing imported from PceDraw
drawing(xpcenetscape,
        [ compound(new(A, figure),
                   drawing([ display(box(137, 74)+radius(17),
                                     point(0, 0)),
                             display(text('XPCE', center, normal),
                                     point(52, 30))
                           ]),
                   point(163, 183)),
          compound(new(B, figure),
                   drawing([ display(box(137, 74)+radius(17),
                                     point(0, 0)),
                             display(text('Netscape', center, normal),
                                     point(42, 30))
                           ]),
                   point(350, 183)),
          connect(connection(A,
                             B,
                             handle(w, h/2, link, east),
                             handle(0, h/2, link, west)) +
                    arrows(both))
        ]).
:- pce_end_class(my_httpd).
```

11.9.1 Class `httpd`

The library library(`http/httpd`) defines the class `httpd`. This subclass of `socket` deals with most of the HTTP protocol details, breaking down HTTP requests and encapsulating responses with the proper headers. The class itself is an *abstract* class, a subclass needs to be created and some of the *virtual methods* needs to be refined to arrive at a useful application.

httpd → initialise: Port:[int]
Create a server and bind it to *Port*. If *Port* is omitted a free port is chosen. With a specified port, 8080 is a commonly used alternative to the standard 80 used by web-servers. If you have a web-server running on the same machine you may can generate a page on your website redirecting a page to this server. The URI of this server is `http://`⟨*host*⟩`/`⟨*Port*⟩.

httpd → accepted
This is sent after a connection has been accepted. The system implementation logs the new connection if debugging is enabled. You can refine or redefine this method, asking for the '`socket ← peer_name`' and sending →`free` to the socket if you want to restrict access.

httpd → request: Data:sheet
This is sent from →`input` after a complete request-header is received. →`input` decodes the header-fields, places them in *Data* and then calls →`request`. The attribute-names in the sheet are downcase versions of the case-insensitive request fields of the HTTP header. In addition, the following fields are defined:

Fields that are always present	
request	`GET`, `POST`, etc. I.e. the first word of the request-header. In most cases this will be `GET`.
path	The 'path' part of the request. This is normally used to decide on the response. If the path contains a ? (question mark) this and the remaining data are removed and decoded to the 'form' attribute.
form	If the request is a `GET` request with form-data, the form attribute contains another sheet holding the decoded form-data. Otherwise ←`form` holds `@nil`.
http_version	Version of the HTTP protocol used by the client. Normally `1.0` or `1.1`.
Other fields	
user	If authorisation data is present, this contains the user-name. If this field is present, the password field is present too.
password	Contains the decoded password supplied by the user.

After decoding the request, the user should compose a response and use →`reply` or →`reply_html` to return the response to the client.

httpd → reply: Data:string—source_sink—pixmap, Type:[name], Status:[name], Header:[sheet]

Send a reply. This method or →`reply_html` is normally activated at the end of the user's →`request` implementation. Data is one of:

- *A string or source_sink*
  If the reply is a `string`, `text_buffer`, `resource` or `file`, the data in this object will be returned. Unless otherwise specified →`reply` assumes the data has mime-type `text/plain`.
- *A pixmap*
  If the reply is a `pixmap` (or can be converted automatically, for example any `graphical`), this image is encoded as GIF or JPEG and sent with the corresponding image/gif or image/jpeg mime-type. For more information on image save-types, see '`image → save_in`'.

Type is the mimi-type returned and tells the browser what to do with the data. This should correspond with the content of *Data*. For example, you can return a PNG picture from a file using

```
send(HTTPD, reply, file('pict.png'), 'image/png'),
```

Status is used to tell the client in a formal way how the request was processed. The default is `200 OK`. See the methods below for returning other values.

Header is a `sheet` holding additional name-value pairs. If present, they are simply added to the end of the reply-header. For example if you want to prevent the browser caching the result you can use

```
send(HTTPD, reply, ...,
     sheet(attribute('Cache-Control', 'no-cache'))),
```

httpd → reply_html: Term:prolog, Status:[name], Header:[sheet]

Uses the library(`http/html_write`) library to translate *Term* into HTML text using DCG rules and then invokes →`reply` using the *Type* `text/html`. *Status* and *Header* are passed unmodified to →`reply`.

In addition to the principal methods above, a number of methods are defined for dealing with abnormal replies such as denying permission, etc.

httpd → forbidden: What:[name]

Replies with a `403 Forbidden` message. *What* may be provided to indicate what is forbidden. Default is the path from the current ←`request`.

httpd → authorization_required: Method:[{Basic}], Realm:[name]

Challenges the user to provide a name and password. The only method provided is `Basic`. *Realm* tells the user for which service permission is requested. On all subsequence contacts from this client to this server the →`request` data contains the `user` and `password` fields. The demo implementation of →`request` in `httpd` contains the following example code:

```
request(S, Header:sheet) :->
        "Process a request.  The argument is the header"::
        (   get(Header, path, '/no')
        ->  send(S, forbidden, '/no')
        ;   get(Header, path, '/maybe')
        ->  (   get(Header, value, user, jan),
                get(Header, value, password, test)
            ->  send(S, reply, 'You hacked me')
            ;   send(S, authorization_required)
            )
        ;   send(S, reply, 'Nice try')
        ).
```

httpd → not_found: What:[char_array]
Reply with a `404 Not Found` message, using the request-path as default for *What*.

httpd → moved: Where:char_array
Reply with a `301 Moved Permanently`. Normally the client will retry the request using the URL returned in *Where*.

httpd → server_error: What:[char_array]
Reply with a `500 Internal Server` using '*What* as additional information to the user. This is the default reply if →`request` fails or raised an exception.

11.9.2 The "http/html_write" library

Producing output for the web in the form of an HTML document is a requirement for many Prolog programs. Just using `format/2` is satisfactory as it leads to poorly readable programs generating poor HTML. This library is based on using DCG rules.

The library(`pl/html_write`) is intended to structure the generation of HTML from a program. It is an extensible library, providing a *DCG* framework for generation of legal HTML under (Prolog) program control. It is especially useful for the generation of structured pages (tables) from Prolog data structures.

The normal way to use this library is through the DCG `html/1`. This grammar-rule provides the central translation from a structured term with embedded calls to additional translation rules to a list of atoms that can then be printed using `print_html/[1,2]`.

html(:*Spec***)**
The DCG rule `html/1` is the main predicate of this library. It translates the specification for an HTML page into a list of atoms that can be written to a stream using `print_html/[1,2]`. The expansion rules of this predicate may be extended by defining the multifile DCG html_write:`expand/1`. *Spec* is either a single specification or a list of single specifications. Using nested lists is not allowed to avoid ambiguity caused by the atom `[]`

- *Atomic data*
  Atomic data is quoted using the `html_quoted/1` DCG.

- Fmt - Args
  Fmt and *Args* are used as format-specification and argument list to `sformat/3`. The result is quoted and added to the output list.
- \List
  Escape sequence to add atoms directly to the output list. This can be used to embed external HTML code.
- \Term
  Invoke the grammar rule *Term* in the calling module. This is the common mechanism to realise abstraction and modularisation in generating HTML.
- Module*:*Term
  Invoke the grammar rule ⟨*Module*⟩:⟨*Term*⟩. This is similar to *Term* but allows for invoking grammar rules in external packages.
- *&(Entity)*
  Emit &⟨*Entity*⟩`;`.
- `Tag`*(Content)*
  Emit HTML element *Tag* using *Content* and no attributes. *Content* is handled to `html/1`. See section 11.9.2 for details on the automatically generated layout.
- `Tag`*(Attributes, Content)*
  Emit HTML element *Tag* using *Attributes* and *Content*. *Attributes* is either a single attribute of a list of attributes. Each attributes is of the format `Name`(*Value*) or `Name`(*Value*).

page(*:HeadContent, :BodyContent***)**

The DCG rule `page/2` generated a complete page, including the SGML `DOCTYPE` declaration. *HeadContent* are elements to be placed in the `head` element and *BodyContent* are elements to be placed in the `body` element.

To achieve common style (background, page header and footer), it is possible to define DCG rules `head/1` and/or `body/1`. The `page/1` rule checks for the definition of these DCG rules in the module it is called from as well as in the `user` module. If no definition is found, it creates a head with only the *HeadContent* (note that the `title` is obligatory) and a `body` with `bgcolor` set to `white` and the provided *BodyContent*.

Note that further customisation is easily achieved using `html/1` directly as `page/2` is (besides handling the hooks) defined as:

```
page(Head, Body) -->
        html([ \['<!DOCTYPE HTML PUBLIC "-
//IETF//DTD HTML 4.0//EN">\n'],
               html([ head(Head),
                      body(bgcolor(white), Body)
                    ])
             ]).
```

page(*:Contents***)**

This version of the `page/[1,2]` only gives you the SGML `DOCTYPE` and the `HTML` element. *Contents* is used to generate both the head and body of the page.

html_begin(+*Begin*)

Just open the given element. *Begin* is either an atom or a compound term, In the latter case the arguments are used as arguments to the begin-tag. Some examples:

```
html_begin(table)
html_begin(table(border(2), align(center)))
```

This predicate provides an alternative to using the *Command* syntax in the `html/1` specification. The following two fragments are the same. The preferred solution depends on your preferences as well as whether the specification is generated or entered by the programmer.

```
table(Rows) -->
        html(table([border(1), align(center), width('80%')],
                   [ \table_header,
                     \table_rows(Rows)
                   ])).

% or

table(Rows) -->
        html_begin(table(border(1), align(center), width('80%'))),
        table_header,
        table_rows,
        html_end(table).
```

html_end(*E***)**

nd an element. See `html_begin/1` for details.

Emitting HTML documents

The `html/1` grammar rules translates a specification into a list of atoms and layout instructions. Currently the layout instructions are terms of the format `nl`(*N*), requesting at least *N* newlines. Multiple consequtive `nl`(*1*) terms are combined to an atom containing the maximum of the requested number of newline characters.

To simplify handing the data to a client or storing it into a file, the following predicates are available from this library:

print_html(+*List*)

Print the token list to the Prolog current output stream.

print_html(+*Stream, +List*)

Print the token list to the specified output stream

html_print_length(+*List, -Length*)

When calling `html_print/[1,2]` on *List*, *Length* characters will be produced. Knowing the length is needed to provide the `Content-length` field of an HTTP reply-header.

Adding rules for `html/1`

In some cases it is practical to extend the translations imposed by `html/1`. When using XPCE for example, it is comfortable to be able defining default translation to HTML for objects. We also used this technique to define translation rules for the output of the SWI-Prolog library(`sgml`) package.

The `html/1` rule first calls the multifile ruleset html_write:`expand/1`. The other predicates contain commonly rules for defining new rules.

html_write:expand(+*Spec*)
Hook to add additional translationrules for `html/1`.

html_quoted(+*Atom*)
Emit the text in *Atom*, inserting entity-references for the SGML special characters <&>.

html_quoted_attribute(+*Atom*)
Emit the text in *Atom* suitable for use as an SGML attribute, inserting entity-references for the SGML special characters <&>'".

Generating layout

Though not strictly necessary, the library attempts to generate reasonable layout in SGML output. It does this only by inserting newlines before and after tags. It does this on the basis of the multifile predicate html_write:`layout/3`

html_write:layout(+*Tag, -Open, -Close*)
Specify the layout conventions for the element *Tag*, which is a lowercase atom. *Open* is a term *Pre-Post*. It defines that the element should have at least *Pre* newline characters before and *Post* after the tag. The *Close* specification is similar, but in addition allows for the atom `-`, requesting the output generator to omit the close-tag altogether or `empty`, telling the library that the element has declared empty content. In this case the close-tag is not emitted either, but in addition `html/1` interprets *Arg* in `Tag`(*Arg*) as a list of attributes rather than the content.

A tag that does not appear in this table is emitted without additional layout. See also `print_html/[1,2]`. Please consult the library source for examples.

Examples

In the following example we will generate a table of Prolog predicates we find from the SWI-Prolog help system based on a keyword. The primary database is defined by the predicate `predicate/5` We will make hyperlinks for the predicates pointing to their documentation.

```
html_apropos(Kwd) :-
        findall(Pred, apropos_predicate(Kwd, Pred), Matches),
        phrase(apropos_page(Kwd, Matches), Tokens),
        print_html(Tokens).

%       emit page with title, header and table of matches
```

```
apropos_page(Kwd, Matches) -->
        page([ title([’Predicates for ’, Kwd])
             ],
             [ h2(align(center),
                  [’Predicates for ’, Kwd]),
               table([ align(center),
                       border(1),
                       width(’80%’)
                     ],
                     [ tr([ th(’Predicate’),
                            th(’Summary’)
                          ])
                     | \apropos_rows(Matches)
                     ])
             ]).

%       emit the rows for the body of the table.

apropos_rows([]) -->
        [].
apropos_rows([pred(Name, Arity, Summary)|T]) -->
        html([ tr([ td(\predref(Name/Arity)),
                    td(em(Summary))
                  ])
             ]),
        apropos_rows(T).

%       predref(Name/Arity)
%
%       Emit Name/Arity as a hyperlink to
%
%               /cgi-bin/plman?name=Name&arity=Arity
%
%       we must do form-encoding for the name as it may con-
tain illegal
%       characters.  www_form_encode/2 is defined in library(url).

predref(Name/Arity) -->
        { www_form_encode(Name, Encoded),
          sformat(Href, ’/cgi-bin/plman?name=~w&arity=~w’,
                  [Encoded, Arity])
        },
        html(a(href(Href), [Name, /, Arity])).

%       Find predicates from a keyword. ’$apro-
pos_match’ is an internal
```

```
%       undocumented predicate.

apropos_predicate(Pattern, pred(Name, Arity, Summary)) :-
        predicate(Name, Arity, Summary, _, _),
        (   '$apropos_match'(Pattern, Name)
        ->  true
        ;   '$apropos_match'(Pattern, Summary)
        ).
```

Final remarks

This library is the result of various attempts to reach at a more satisfactory and Prolog-minded way to produce HTML text from a program. We have been using Prolog for the generation of web pages in a number of projects. Just using `format/2` never was a real option, generating error-prone HTML from clumsy syntax. We started with a layour on top of format, keeping track of the current nesting and thus always capable of properly closing the environment.

DCG based translation however naturally exploits Prologs term-rewriting primitives. If generation fails for whatever reason it is easy to produce an alternative document (for example holding an error message).

The approach presented in this library has been used in combination with library(`http/httpd`) in three projects: viewing RDF in a browser, selecting fragments from an analysed document and presenting parts of the XPCE documentation using a browser. It has proven to be able to deal with generating pages quickly and comfortably.

In a future version we will probably define a `goal_expansion/2` to do compile-time optimisation of the library. Quotation of known text and invokation of sub-rules using the *RuleSet* and ⟨*Module*⟩:⟨*RuleSet*⟩ operators are costly operations in the analysis that can be done at compile-time.

11.10 Document rendering primitives

Dynamic HTML has enabled the presentation of user interfaces as documents the user can interact with. None of the traditional GUI components can deal with the mixture of properly formatted text with (interactive) graphics. XPCE provides support using a set of primitive classes that realise a box-model based on TEX. This basic model is enhanced using a Prolog library providing HTML-based primitives. First we will introduce the basics.

- *Class* `parbox`
  This is the central class of the document-rendering support. It is a subclass of `device`, placing `hbox` objects as words within a paragraph. Parbox devices have a ⇆`width`, realise alignment and adjustment and can place text around *floating* objects.

- *Class* `hbox`
  This is an abstract super-class for `tbox` and `grbox` that allow for text and graphics within a `parbox`.

- *Class* `tbox`
  Represent text in a `parbox`. Normally, a `tbox` represents a word.

- *Class* `grbox`
  Embeds any `graphical` object in a `parbox`. The `grbox` negotiates with the `parbox` on the placement and then places the graphical on the `parbox` using normal behaviour of `device`.

- *Class* `lbox`
  This is another subclass of `device` that realises LATEX-like list-environment. The class deals with placement of labels and text for the items. Both label and item are arbitrary graphical objects. Items are normally instances of `parbox`.

- *Class* `rubber`
  This is a data object containing information on the *stretchability* of boxes. Rubber object are used to distribute space horizontally as well as to determine the location of line-breaks. A space in a `parbox` is realised using a `hbox` whose natrual width is the width of a space in the current `font` that can shrink a little and expand a bit easier.

Before discussing the library we show a small example using the primitives directly.

```
1  parbox :-
2          send(new(W, window), open),
3          send(W, display, new(P, parbox(W?width)), point(10,10)),
4          send(W, resize_message, message(P, width, @arg2?width-20)),
5
6          send(P, alignment, justify),
7          send_list(P,
8                    [ append(grbox(box(40,40), left)),
9                      cdata('This is the central class of the '),
10                     cdata('document-rendering support.  It is '),
11                     cdata('a subclass of '),
12                     cdata('device', style(underline := @on)),
13                     cdata(', placing '),
```

```
14                    cdata('hbox', style(underline := @on)),
15                    cdata(' objects as words within a paragraph. '),
16                    cdata('Parbox devices have a <->width, realise '),
17                    cdata('alignment and adjustment and can place text '),
18                    cdata('around '),
19                    cdata('floating ', style(font := italic)),
20                    cdata(' objects.')
21                  ]).
```

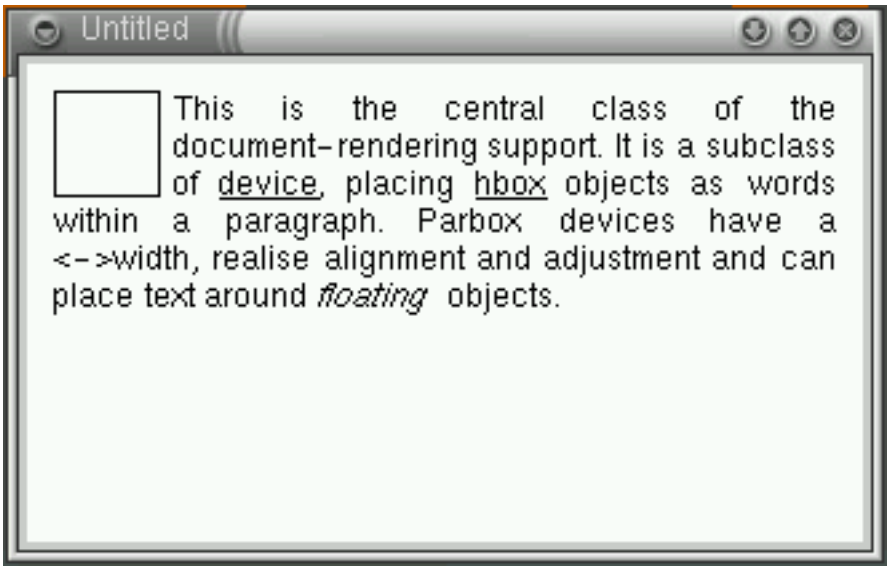


Figure 11.10: A parbox floating text around a box

In line 4, we forward changes to the width of the window to the `parbox` to force reformatting the text if the width of the window changes. Line 6 asks for a straight right-margin. Line 8 appends a `box` that is left aligned, the text floating around it. The remaining lines append text. The method '`parbox` → `cdata`' breaks the argument into words and adds instances of `tbox` for each word, separated by instances of `hbox` with appropriate `rubber` for each sequence of white-space, doing much the same as an HTML browser processing `CDATA` text.

Defining a document from Prolog text using these primitives is not attractive. This is the motivation for using a library.

11.10.1 The rendering library

The directory `\bnfmeta{pcehome}/prolog/lib/doc` contains the document-rendering library, providing HTML-oriented primitives for document rendering. The translation process can be extended by defining predicates for two *multifile* predicates.

The central library is library(`doc/emit`). This library defines the predicate `emit/3`:

emit(*+ListOfTerm, +PBox, +Mode***)**

This predicate takes a list of processing instructions and applies these on *PBox*, an

instance of class pbox, a subclass of parbox supporting this library. *Mode* provides an instance of doc_mode, a data-object holding style-information.

ListOfTerm consists of the following instructions:

- ⟨*Atomic*⟩
  Atomic data is added as text to the output. If 'doc_mode ← space_mode' equals preserve, spaces are preserved. Otherwise space is canonised, smashing multiple consecutive banks into a single (rubber) space.
- *Action*
  Execute an *Action*. Actions are like TEX commands, which is why we use the backslash. The built-in actions and the definition of new actions is discussed in section 11.10.3.
- @Object
  XPCE objects are simply appended to the pbox. This can be used to append self-created objects or use one of the predefined layout objects defined in section 11.10.2.

Before handed to the above, emit/3 calls the multifile predicate doc:emit/3 passing the whole list as you can see from the definition below. This allows the user-hook to process sequences of instructions.

```
emit(Term, PB, Mode) :-
        doc:emit(Term, PB, Mode), !.
```

11.10.2 Predefined objects

The file library(doc/objects) defines a number of globally named instances of the box classes. Appending these objects is like executing an action. These objects are:

@br	Nul-dimension hbox with rubber to force a line-break.
@nbsp	Non-breaking space.
@hfill	Rubber hbox for alignment and centering.
@space_rubber	Rubber used for hbox objects representing a space. This rubber allows for a line-break.
@h⟨*n*⟩_above	Space above HTML section-headers.
@h⟨*n*⟩_below	Space below HTML section-headers.

11.10.3 Class and method reference

This section provides a partial reference to the classes and methods defining the document-rendering library. For full information, please use the ClassBrowser and check the source-code.

pbox → show: Content:prolog, Mode:[doc_mode]
Calls emit/3 using *Content* and *Mode*. If mode is omitted, a default mode object is created.

pbox → event: Event:event
Handles clicking a button or footnote and showing *balloons* from buttons after trying to pass the event to one of the embedded graphicals.

pbox ← anchor: Name → tuple(Box, Index)
Return the *Box* and index thereof in the '`parbox ← content`' vector that starts the named anchor (see section 11.10.3.

doc_mode → initialise
Creates a default document-rendering mode. This mode has the following properties:

vfont	new(vfont)	The *virtual* font for text
link_colour	dark_green	Text-colour while rendering buttons
parsep	hbox(0,8)	Skip 8 pixels between paragraphs
parindent	hbox(0,0)	Do not indent paragraphs
space_mode	canonical	Collapse spaces
alignment	justify	Make a right-margin
base_url	”	URL for HTML hyper-links

doc_mode → set_font: Att:name, Value:any
Set an attribute of ←`vfont` and update ←`style` and ←`space` to reflect the new font-settings.

doc_mode → colour: Colour
Set the colour of ←`style`.

doc_mode → underline: Bool
Set underline mode in ←`style`.

Class `vfont`

The class `vfont` realises *virtual* fonts, Microsoft also calls these *logical* fonts. XPCE `font` objects are read-only. When dealing with incremental font manipulation it is necessary to have a font that can be manipulated. Therefore, `vfont` defines a number of slots to represent the font attributes regardless of the fonts existence. At any time the user can request the best matching font using '`vfont ← font`'. The mapping between virtual font attributes and physical fonts can be programmed by adding clauses to the multifile predicate vfont:`font_map/7`. This class is defined in library(`doc/vfont`) where you find further programming details.

Rendering actions

The action subsystem processes actions (\Action) from `emit/3`, providing a hook for adding new actions. Before doing anything else, the hook doc:`action/3` is called:

doc:action(*+Action, +PBox, +Mode***)**
Execute *Action*. The actions are passed from `emit/3` after stripping the backslash. If this hook succeeds the action is considered handled.

The built-in actions are:

ignorespaces
Tells `emit/3` eat all input until the first action or non-blank character.

space(*SpaceMode***)**
Tells `emit/3` to `preserve` white-space or render it `canonical`. Default is `canonical`.

pre(*Text***)**
Add verbatim text.

par
Start a new paragraph. This is the action-sequence `parskip`, followed by `parsep`.

parskip
Inserts ←`parsep` from the current mode, surrounded by two line-breaks (`@br`).

parindent
Insert the ←`parindent` from the current mode.

group(*Group***)**
Use `emit/3` on *Group* on a ←`clone` of the `doc_mode` object. This is essentially the same as a TEX group, scoping changes to the mode such as font-changes, etc.

setfont(*Attribute, Value***)**
Set an attribute of the font. Fonts are maintained using the Prolog defined class `vfont` (*virtual* font) that allows for independent changes to font attributes. When text needs to be rendered, a close real font is mounted. Defined attributes are: `family`, `encoding`, `slant`, `weight`, `fixed` and `size`. See `\bnfmeta{pcehome}/prolog/lib/doc/vfont.pl` for details.

ul
Switch on underlining. Normally this should be inside a group to limit the scope.

colour(*Colour***)**
Set the text-colour.

list(*Options, Content***)**
Produce a list-environment. The option `class`(*Class*) defines the subclass of `lbox` used to instantiate, default `bullet_list`. The *Content* is passed to `emit/3`, using the created list-object as 2nd argument.

When using a `bullet_list` or `enum_list` *Content* must translated into a sequence of `li` commands. Using a `definition_list`, a sequence of `dt` and `dd` commands is expected.

li(*Content***)**
If we are processing a `bullet_list` or `enum_list`, start a new item using '←`make_item`', then emit *Content* to the new item.

dt(*ItemTitle***)**
If we are processing a `definition_list`, create a new label from *ItemTitle*.

dd(*ItemContent***)**
Create a `pbox` for the item body and emit *ItemContent* to it.

title(*Title***)**
Get the ←`frame` of the `pbox` and send it the given title using '`frame` → `label`'.

body(*Options***)**
Apply options to the window as a whole. Defines options are `bgcolour`(*Colour*), `background`(*Image*) and `text`(*Colour*).

button(*Message, Content, Balloon***)**
Add an active area (hyper-link) to the document. When depressed, *Message* is executed. When hoovering, *Balloon* is reported as status (see section 10.7). *Content* is emitted inside a group after setting the default colour to '`doc_mode` ← `link_colour`' and underlining to @on.

anchor(*Label, Content***)**
Label some content. This has no visual implications, but the the anchor can be located using '`pbox` ← `anchor`'.

parbox(*Content, Options***)**
Add a sub-parbox. Defined options are:

width(*Width***)**
Define the width of the sub-box.

rubber(*Rubber***)**
Define the shrink- and stretchability of the sub-box.

align(*Alignment***)**
Define text-adjustment (`left`,`center`,`right`,`justify`) within the box.

valign(*VAlign***)**
Define the vertical alignment of the box (`top`, `center`, `bottom`.

auto_crop(*Bool***)**
If @on, tell the `pbox` its ←`area` is determined by the content rather than the specified width. Text may be formatted left-to-write without wrapping by defining a wide parbox and using this option.

table(*Options, Content***)**
Create a tabular layout using the class `doc_table`, a `device` holding a `table`. See also section 11.5. The options and commands are modelled after HTML-3. Table-related commands are `tr`, `td`, `col`, `thead` and `tbody`. Defined options are:

align(*Align***)**
Graphical alignment, allowing placement as `left` or `right` floating object or `centered` placement.

width(*Width***)**
Give the table a specified width.

bgcolor(*Colour***)**
Set the default background colour for the rows, columns and cells.

cellpadding(*IntOrSize***)**
Specify the space inside a cell around its content.

cellspacing(*IntOrSize***)**
Specify the space between cells.

tr
Open the next table-row.

tr(*Options, Content***)**
Open a new row and emit *Content* therein. *Options* are applied to the row. See class `table_row` for details.

td(*Options, Content***)**
Add a table-cell and emit *Content* into it. *Options* are applied to the new cell. See class `table_cell` for details.

col(*Options***)**
Handle an HTML-3 `col` element, specifying parameters for the next column. Defined *Options* are `span`(*Span*) to apply these settings to the next *Span* columns and `width`(*Spec*), where *Spec* is an integer (pixels) or a term `*`(*N*), defining the weight for shrinking and stretching relative to other columns using this notation. The following defines the second column to be twice as wide as the first:

```
[ \col(*(1)),
  \col(*(2))
]
```

tbody(*Options***)**
Start a row-group. See '`table_row` → `end_group`'. *Options* is currently ignored.

thead(*Options, Content***)**
Handle a table-head. It expects a number of rows in *Content*. While processing *Content* it sets the default cell alignment to `center` and font to `bold`.

footnote(*Content***)**
Add a footnote-mark. Pressing the mark shows a popup-window holding the text of the footnote.

preformatted(*Text***)**
Adds *text* in a `tbox` to the parbox without checking the content. The current style is applied to *Text*

11.10.4 Using the “doc/emit” library

In section 11.10.1 and section 11.10.3 we have seen the definition of the basic rendering library infrastructure. It uses concepts from TEX and HTML-3, introducing primitives for grouping, attribute settings, lists, tables and whitespace-handling.

The `emit/3` predicate as described above is not intended for direct use though. It is hard to imagine a good syntax for defining significant amounts of formatted text in a Prolog textfile. In some cases it is feasible to define a suitable set of new commands and use `emit/3` directly from Prolog. In most cases you’ll want to use tokens from an external source using an external markup language.

One option to arrive at a token-list is using the XML/SGML parser included in SWI-Prolog. It can be used either with a domain-specific DTD, in which case you need to define the translations by hand or using an HTML DTD, in which case the library library(`doc/html`) defines the required translations.

We will illustrate this in a small example. First the code to show HTML inside a window is below. In line 1 we load the always-needed document rendering infra-structure and register the `doc` search-path to reflect the `\bnfmeta{pcehome}/prolog/lib/doc` directory. Next we import `emit/3` and load the HTML-rendering extensions to doc:`emit/3` and doc:`action/3`.

```
1  :- use_module(library('doc/load')).
2  :- use_module(doc(emit)).
3  :- use_module(doc(html)).
4  :- use_module(library(sgml)).
5
6  show_html(File) :-
7          send(new(P, picture), open),
8          send(P, display, new(PB, pbox), point(10,10)),
9          send(P, resize_message, message(PB, width, @arg2?width - 20)),
10
11         load_html_file(File, Tokens),
12         send(PB, show, Tokens).
```

Here is the HTML code loaded and the visual result.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">

<html>
  <head>
    <title>Document rendering</title>
  </head>
<body>
<h1>SWI-Prolog SGML-DOM</h1>

<p>
SWI-Prolog 4.0 provides a library for loading XML, SGML and
HTML files and convert them to a complex Prolog term.  This
term has the format
```

```
<p>
<pre>
element(Name,
        ListOfNameValue
        ListOfContent)
</pre>

<p>
Where <var/ListOfNameValue/ reflects the attribute-list of the
element and <var/ListOfContent/ is mixed list of atoms representing
<em/CDATA/ and <b>element/3</b> terms representing nested elements.
</body>
</html>
```

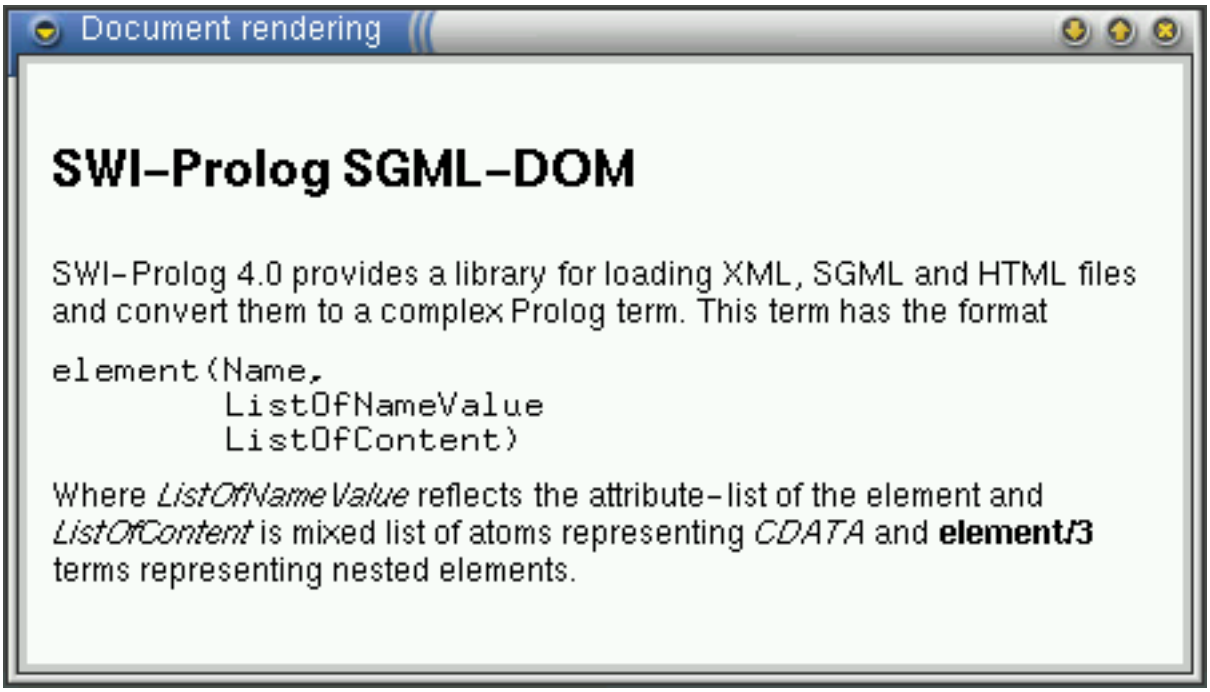


Figure 11.11: Rendering HTML code

In general you do not want to render plain HTML using XPCE/Prolog as it is far less flexible than real browsers dealing with errornous HTML, the implementation of HTML is incomplete and it supports Java nor Javascript.

It has proved to be worthwile using the extensibility of SGML and this layout to render domain-specific documents, often using HTML elements for the basic layout.

11.11 Library “broadcast” for all your deliveries

to realise GUI applications consisting of stand-alone components that use the Prolog database for storing the application data. Figure 11.12 illustrates the flow of information using this design

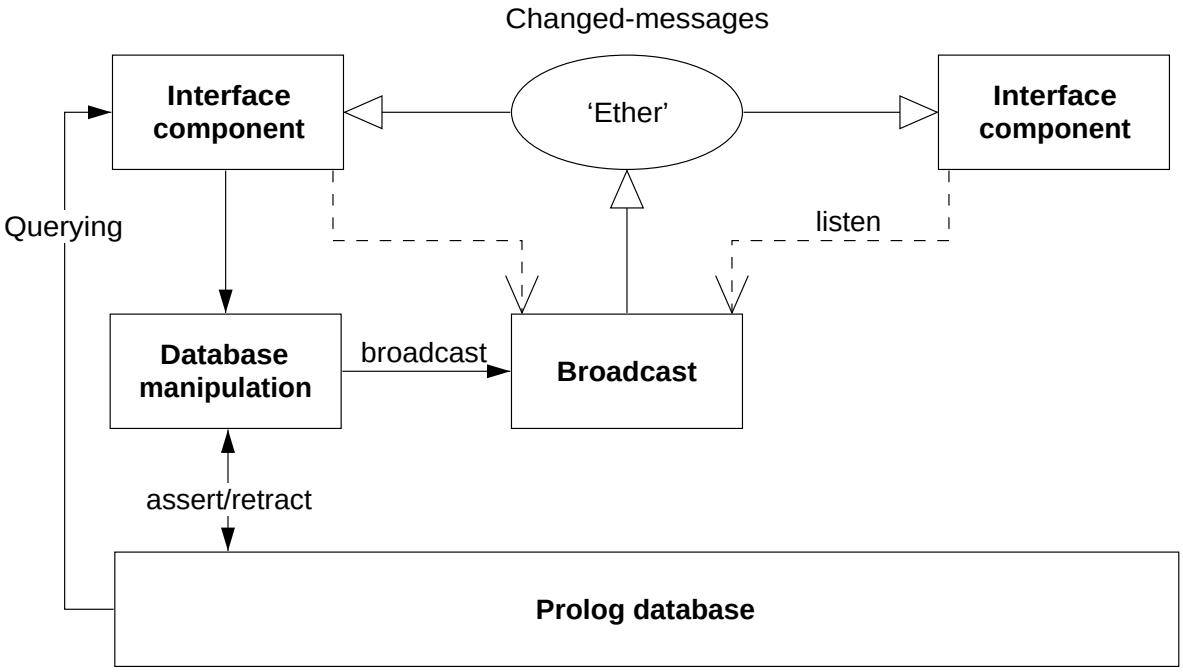


Figure 11.12: Information-flow using broadcasting service

The broadcasting service provides two services. Using the ‘shout’ service, an unknown number of agents may listen to the message and act. The broadcaster is not (directly) aware of the implications. Using the ‘request’ service, listening agents are asked for an answer one-by-one and the broadcaster is allowed to reject answers using normal Prolog failure.

Shouting is often used to inform about changes made to a common database. Other messages can be “save yourself” or “show this”.

Requesting is used to get information while the broadcaster is not aware who might be able to answer the question. For example “who is showing X?”.

broadcast(+*Term***)**

Broadcast *Term*. There are no limitations to *Term*, though being a global service, it is good practice to use a descriptive and unique principal functor. All associated goals are started and regardless of their success or failure, `broadcast/1` always succeeds. Exceptions are passed.

broadcast_request(+*Term***)**

Unlike `broadcast/1`, this predicate stops if an associated goal succeeds. Backtracking causes it to try other listeners. A broadcast request is used to fetch information without knowing the identity of the agent providing it. C.f. “Is there someone who knows the age of John?” could be asked using

```
        ...,
        broadcast_request(age_of('John', Age)),
```

If there is an agent (*listener*) that registered an 'age-of' service and knows about the age of 'John' this question will be answered.

listen(+*Template, :Goal***)**

Register a *listen* channel. Whenever a term unifying *Template* is broadcasted, call *Goal*. The following example traps all broadcasted messages as a variable unifies to any message. It is commonly used to debug usage of the library.

```
?- listen(Term, write_ln(Term)).
?- broadcast(hello(world)).
hello(world)

Yes
```

listen(+*Listener, +Template, :Goal***)**

Declare *Listener* as the owner of the channel. Unlike a channel opened using `listen/2`, channels that have an owner can terminate the channel. This is commonly used if an object is listening to broadcast messages. In the example below we define a 'name-item' displaying the name of an identifier represented by the predicate `name_of/2`.

```
1  :- pce_begin_class(name_item, text_item).
2
3  variable(id,    any,    get, "Id visualised").
4
5  initialise(NI, Id:any) :->
6          name_of(Id, Name),
7          send_super(NI, initialise, name, Name,
8                     message(NI, set_name, @arg1)),
9          send(NI, slot, id, Id),
10         listen(NI, name_of(Id, Name),
11                send(NI, selection, Name)).
12
13 unlink(NI) :->
14         unlisten(NI),
15         send_super(NI, unlink).
16
17 set_name(NI, Name:name) :->
18         get(NI, id, Id),
19         retractall(name_of(Id, _)),
20         assert(name_of(Id, Name)),
21         broadcast(name_of(Id, Name)).
22
23 :- pce_end_class.
```

unlisten(+*Listener*)
Deregister all entries created with `listen/3` whose *Listener* unify.

unlisten(+*Listener, +Template*)
Deregister all entries created with `listen/3` whose *Listener* and *Template* unify.

unlisten(+*Listener, +Template, :Goal*)
Deregister all entries created with `listen/3` whose *Listener*, *Template* and *Goal* unify.

listening(?*Listener, ?Template, ?Goal*)
Examine the current listeners. This predicate is useful for debugging purposes.

Development and debugging tools 12

This section describes various tools and techniques to help finding bugs in XPCE/Prolog code. Most of the tracing is done in the hosting Prolog language. XPCE provides additional tools for tracing the execution of certain methods, breaking on the implementation of a Prolog-defined method, finding and inspecting objects.

12.1 Object-base consistency

Unlike Prolog, XPCE is not *secure*: if a Prolog environment traps a fatal error there is almost always a bug in the Prolog system. Except for violating system limits there is no Prolog program that can make the Prolog environment crash. For XPCE this is different. Consider the following example:

```
1 ?- new(@p, picture),
     send(@p, display, new(B, box(100,100))),
     get(B, area, Area),
     free(Area).

Area = @ 803438, B = @803419/box
```

After these calls, the ←area attribute of the box has been destroyed, but the box is not aware of this fact. The utility predicate `checkpce/0` scans the XPCE object-base for various inconsistencies and will report that the box contains a slot referring to a freed object.

```
2 ?- checkpce.

[WARNING: Freed object in slot area of @803419/box: @803438/area]
[PCE: Checked 13173 objects]
```

XPCE uses heuristics trying to avoid that such problems actually crash the system (in the example above execution continues normally).

We advice using `checkpce/0` regularly during program development to verify your application is not violating object consistency. Please see section 10.4 and section 10.11 for techniques to avoid 'dangling' object references.

12.2 Tracing methods

It is often useful to inspect the execution of a particular method. Suppose we want to monitor geometry-changes of graphical objects. The utility predicate `tracepce/1` (see also section D) may be used:

```
1 ?- tracepce((graphical->geometry)).
Trace method: graphical ->geometry
```

As `->` is a standard Prolog operator with priority over 1000, `tracepce/1` requires two brackets. Get-methods may be specified in a similar way. `<-` is not an operator. It is suggested to define `<-` as an infix operator in the XPCE initialisation file, so this operator is available during program development:

```
:- op(100, xfx, <-).
```

Instance variables may be specified as ⟨*class*⟩-⟨*variable*⟩. A trace-point on an instance variable makes both reading and writing the variable visible.

The predicate `notracepce/1` disables a tracepoint.

12.3 Breaking (spy) on methods

Prolog-defined methods are all implemented using the same predicate, which makes it hard to use `spy/1` for starting the debugger on a method. One way around is to stick a call to `trace/0` into the method body and recompile the file. For those among us who dislike this idea there is the possibility to use `spypce/1`.

spypce(+*Spec***)**

Spec specifies a send- or get-method like `tracepce/1`. If the method is defined in Prolog XPCE calls `trace/0` just before starting the implementation of the method.

12.4 Visual hierarchy tool

The “Visual Hierarchy” tool provides an overview of the consists-of relations between the various UI components in a tool. It is part of the online manual tools and may be started from the “Tool” entry in the main dialog. Figure 12.1 shows this tool examining the structure of the “PceDraw” demo program.

This tool is very useful to examine the structure of existing applications (for example the various demo programs). It may also be used to find object-references and to launch the inspector (section 12.5) on a particular object.

There are three ways to expand the tree of the visual hierarchy tool. The first is to expand the tree from the initially displayed root object. The second is to type the reference in the “Visual” text_item and press RETURN. The most comfortable way is to position the mouse in the target object and type META-SHIFT-CONTROL-V.[1]

12.5 Inspector tool

The inspector provides a visual representation of all attributes of an object. It is a visual form of the debugging predicate `show_slots/1` which dumps the class and slot value of the

[1] To remember this sequence: **V** for Visual and all defined modifiers to avoid a conflict with application defined key-bindings.

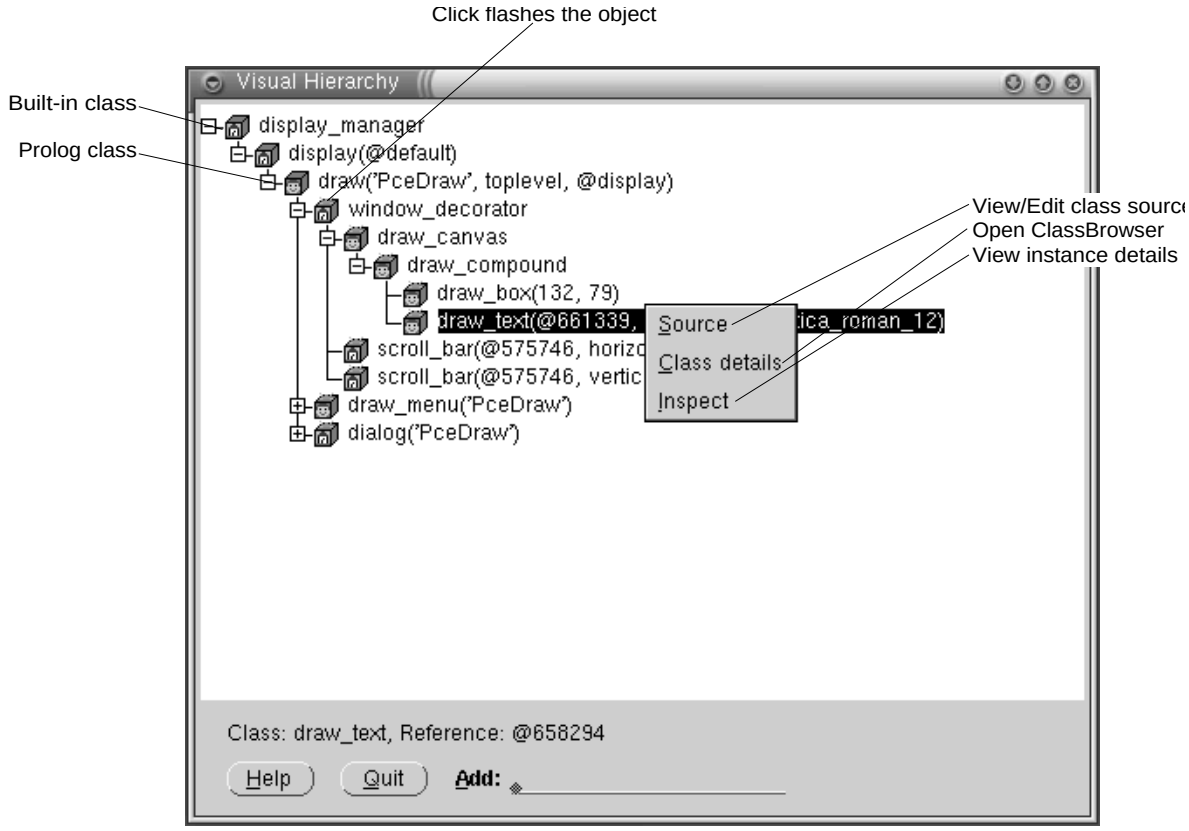


Figure 12.1: Visual Hierarchy Tool showing “PceDraw”

argument reference in the Prolog window. The inspector is started from the “Tools” entry of the manual tools.

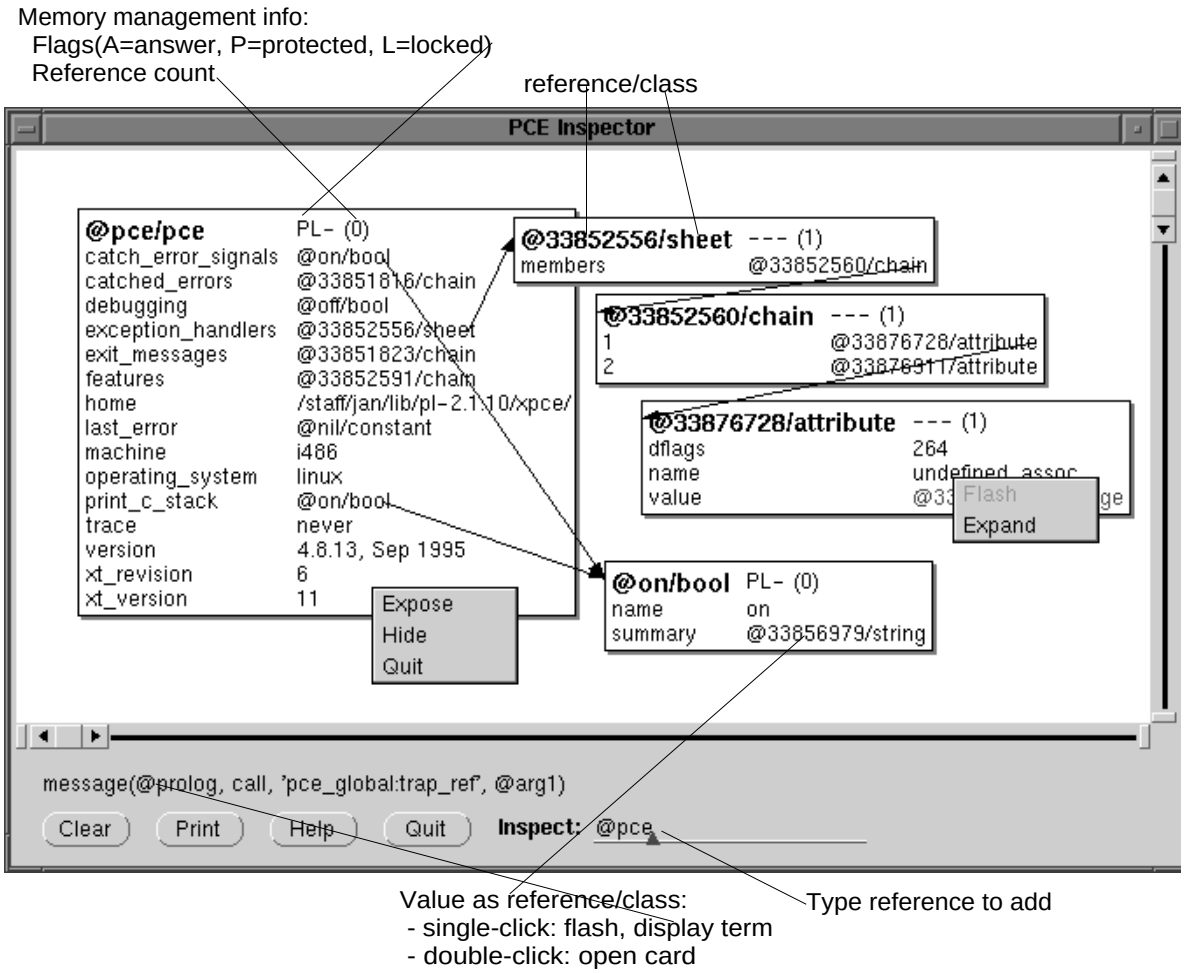


Figure 12.2: Inspector Tool

Bibliography

[Anjewierden, 1992] A. Anjewierden. *XPCE/Lisp: XPCE Common Lisp Interface*. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1992. E-mail: anjo@swi.psy.uva.nl.

[Anjewierden *et al.*, 1990] A. Anjewierden, J. Wielemaker, and C. Toussaint. Shelley — computer aided knowledge engineering. In B. Wielinga, J. Boose, B. Gaines, G. Schreiber, and M. van Someren, editors, *Current trends in knowledge acquisition*, pages 41 – 59, Amsterdam, May 1990. IOS Press.

[Chambers *et al.*, 1989] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF, a dynamic-typed object-oriented language based on prototypes. *Sigplan Notices*, 24(10):49–70, Oct 1989.

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

[Wielemaker & Anjewierden, 1989] J. Wielemaker and A. Anjewierden. Separating User Interface and Functionality Using a Frame Based Data Model. In *Proceedings Second Annual Symposium on User Interface Software and Technology*, pages 25–33, Williamsburg, Virginia, November 1989. ACM Press.

[Wielemaker & Anjewierden, 1993] J. Wielemaker and A. Anjewierden. *XPCE-4 Reference Manual*. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1993. Paper version of online manual.

[Wielemaker & Anjewierden, 1994] J. Wielemaker and A. Anjewierden. *A C++ interface for XPCE*. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1994. E-mail: jan@swi.psy.uva.nl.

[Wielemaker, 1992] J. Wielemaker. *PceDraw: An example of using XPCE-4*. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1992. E-mail: jan@swi.psy.uva.nl.

[Wielemaker, 1994] J. Wielemaker. *XPCE/Prolog Course Notes*. SWI, University of Amsterdam, Roetersstraat 15, 1018

WB Amsterdam, The Netherlands, 1994. E-mail: jan@swi.psy.uva.nl.

[Wielemaker, 1996] J. Wielemaker. *SWI-Prolog 2.5: Reference Manual*. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1996. E-mail: jan@swi.psy.uva.nl.

The dialog editor

A

The dialog editor is a GUI based tool for the definition of dialog windows (windows with controls). It supports the following phases of the definition of a GUI:

- Specifying the required controls
  Prototypes of controls are dragged to the *target* dialog window. They are moved to—roughly—the right location.
- Refining the controls
  The controls may be refined: specifying labels, sizes, fonts, items in menus, etc.
- Fixing the layout and size of the window
  The layout specification for the dialog window is established. The Dialog Editor guesses the layout intentions of the user and translates these into XPCE's dialog-window symbolic layout statements.
- Specifying behaviour
  Both internal behaviour and the link to the application may be established using graphics. The dialog may be tested, while graphical animation illustrates how user-actions are processed and transferred to the application.
- Generation of code
  A Prolog description of the dialog window is realised by dragging the dialog from the list of dialog-windows to a PceEmacs window running in Prolog mode.
- Linking the dialog in the application
  The generated code is a clause of the predicate `dialog/2`. The first argument of this clause *identifies* the dialog, while the second arguments describes the structure and behaviour of the dialog. The body is empty. The library predicate `make_dialog/2` is used to create a dialog window from the description of `dialog/2`.

A.1 Guided tour

We will now illustrate the functionality of the Dialog Editor by defining a simple prompt dialog that asks for a name and has an ok and cancel button to confirm or cancel the operation. The result is shown in figure A.1

A.1.1 Creating the target dialog window

First, start the manual tools using `manpce/0` or `user_help/0`. Then, start the dialog editor using the option Tools/Dialog Editor. The main window of the dialog editor is now opened on

Figure A.1: Ask-name dialog generated by the Dialog Editor

the screen. Use the option File/New Dialog and enter the name ‘ask_name’. This will add ask_name to the Dialog Windows browser and open an window with the title ‘Ask Name’. See figure A.2. This window is called the ‘target window’.

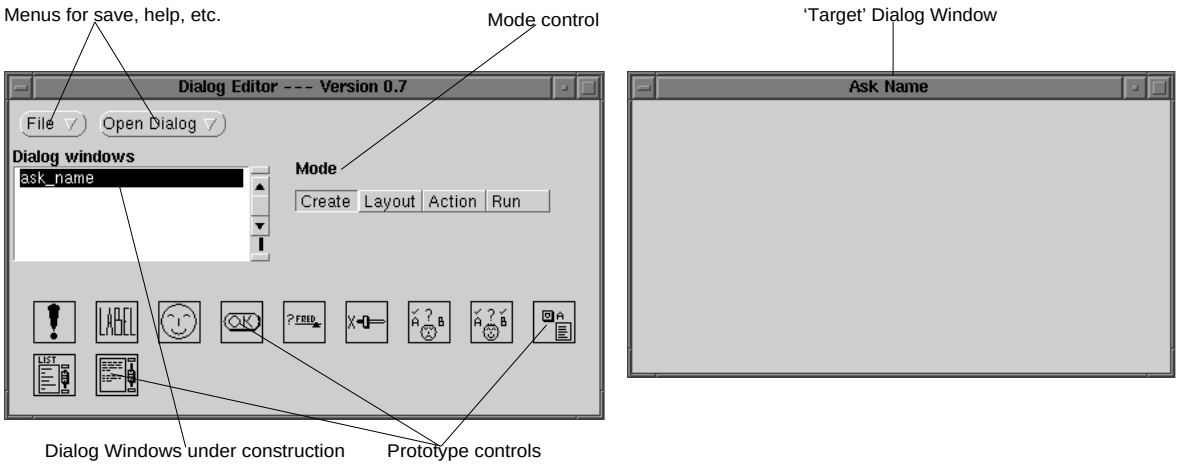


Figure A.2: The Dialog Editor with the ask-name target

A.1.2 Adding controls to the new window

Next, the controls are dragged to the dialog window. The control marked specifies a text-entry-field. Drag this icon using the left-mouse button to the target dialog. If the mouse is above the target dialog, a dotted box will indicate the outline of the new item when it is dropped. Drop it in about the right location. Now drag two instances of to the target dialog and place them again at about the right location. Items can be moved by dragging them with the left button. They can also be **copied** to other target dialog windows by dragging them there and they can be **deleted** by dragging them to the window holding the prototypes.

Now, double-click, using the left button, the text-entry field. A dialog with attributes will appear. The caret is positioned at the Name field. Clear the name field (Control-U is the fastest way) and enter `name`, followed by RETURN. The system will automatically fill the Label field with Name (capitalising the name of the control). If the label should be anything else than the capitalised name, type the correct label now. The other fields are self-explanatory, except for the Type field. This specifies the type of object edited by the text-entry field. See

'`text_item` ⇆ `type`' for details. Pressing Help creates a window containing the online manual cards of all displayed attributes.

Double click on both buttons to fix the name/label. Assign the ok button to be the default button. The result is shown in figure A.3.

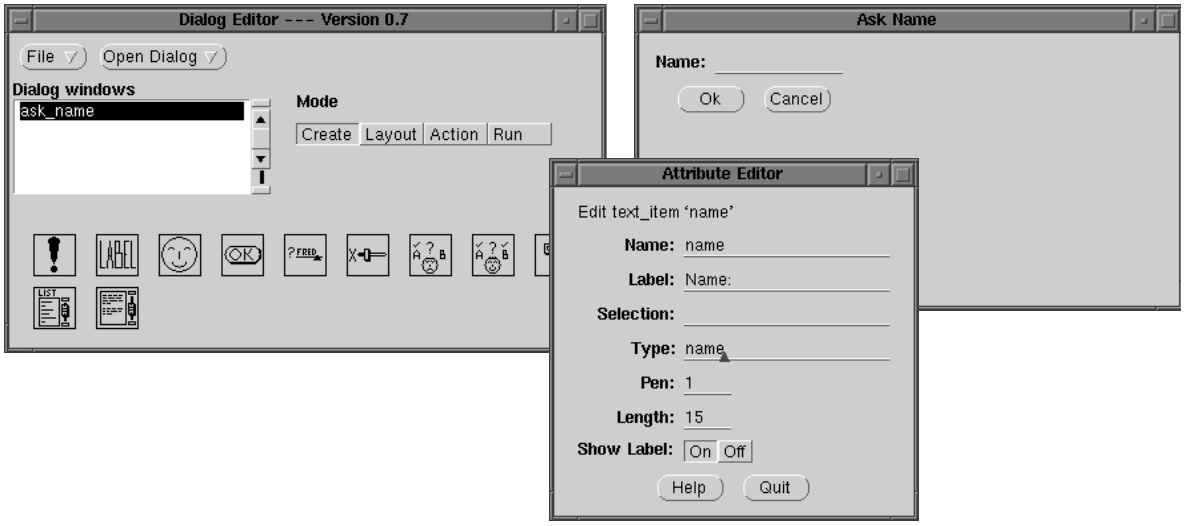


Figure A.3: The Dialog Editor after specifying attributes

A.1.3 Defining the layout

Next, the Mode menu of the main Dialog Editor window is used to select Layout Mode. The button Layout guesses the symbolic layout description of the dialog and places the items. If you are not satisfied with the result, press Undo. Next, you can help the layout mechanism by positioning the items closer to the intended position and try again, or place the items by hand. In the latter case, the generated code will express the layout using pixel-locations rather then a symbolic description and the result may look bad if the end-user runs XPCE with a different look-and-feel. The Fit button adjusts the size of the dialog window to its content.

A.1.4 Specifying the behaviour

The next step is to specify the *behaviour* of the dialog window. Select the Action mode and press the Behaviour Model button to open the behaviour window. Now drag all items from the target dialog window to the behaviour window.

Each control is now represented by a behavioural component. Each such component defines a number of *ports*. The Dialog Editor distinguishes between three types of ports:

- send-port
  A send-port is the representation of a send-method defined on the controller represented by the behavioural component.

- get-port
  A get-port is the representation of a get-method defined on the controller.

- event-port
  An event-port is the representation of an instance variable defined on the controller that can hold an executable (`code`) object (see section 10.2). Controls use these variables to define the *callback* actions of the dialog item.

The window (background) represents the *target* dialog as a whole. Ports may be added to both behavioural components and the background window using the popup menu associated with the right mouse-button.

While hovering over the various parts of the behavioural model, the feedback window at the bottom describes the meaning of the current item. The popup menus defined on all components of the model provide context-sensitive access to the XPCE online manual as well as the online documentation of the Dialog Editor.

In general, the action(s) associated with a control are specified by connecting its event-ports to a send-port in the diagram. The line connecting both ports has a fat dot in the middle. Get-ports may be linked to this fat dot to specify *arguments* for the send-operation. If a get-method needs to be performed on the value of a get-port to define the argument, Extend the get-port, define a new get-port on the extension and link the result to the argument dot.

For our Ask Name dialog, we need to make a *modal* dialog, see section 4.4. Such a dialog returns by invoking the →`return` method on the dialog window. The popup of the background is used to define a send-port named `return`, representing →`return` to the dialog. Position the pointer above the new item to validate your action.

Now, as both methods will make the dialog return immediately, link the `message` event-port of both buttons to this `return` send-port. Link the `selection` get-port of the text field to the argument dot of the link from the Ok button. This specifies that the dialog will return with the '`text_item ← selection`' if the Ok button is pressed. Add the constant `@nil` to the background using the popup menu on the background. This finishes the specification of our dialog window. The resulting behaviour model is shown in figure A.4.

A.1.5 Generating source code

To generate source code, start PceEmacs using the Edit command from the background menu. This will open PceEmacs on the file `ask_name.pl`. Any other PceEmacs window editing a Prolog source file will do as well. Drag the `ask_name` entry from the main window of the Dialog Editor to the PceEmacs window. The window will indicate it is ready to accept the Prolog source code. Now 'drop' the code. The source code will be inserted at the caret location. See figure A.5.

A.1.6 Linking the source code

The generated source is a *description* of the dialog window. This description requires an interpreter to create the dialog window and use it in an application. This interpreter is implemented by `make_dialog/2`:

make_dialog(*?Reference, :Identifier***)**
: Create a dialog window from a description generated by the Dialog Editor. The predicate `make_dialog/2` searches for a predicate `dialog/2` and calls this using the given dialog *Identifier* to obtain a description of the dialog window. See `dialog/2` for the syntax of the description.

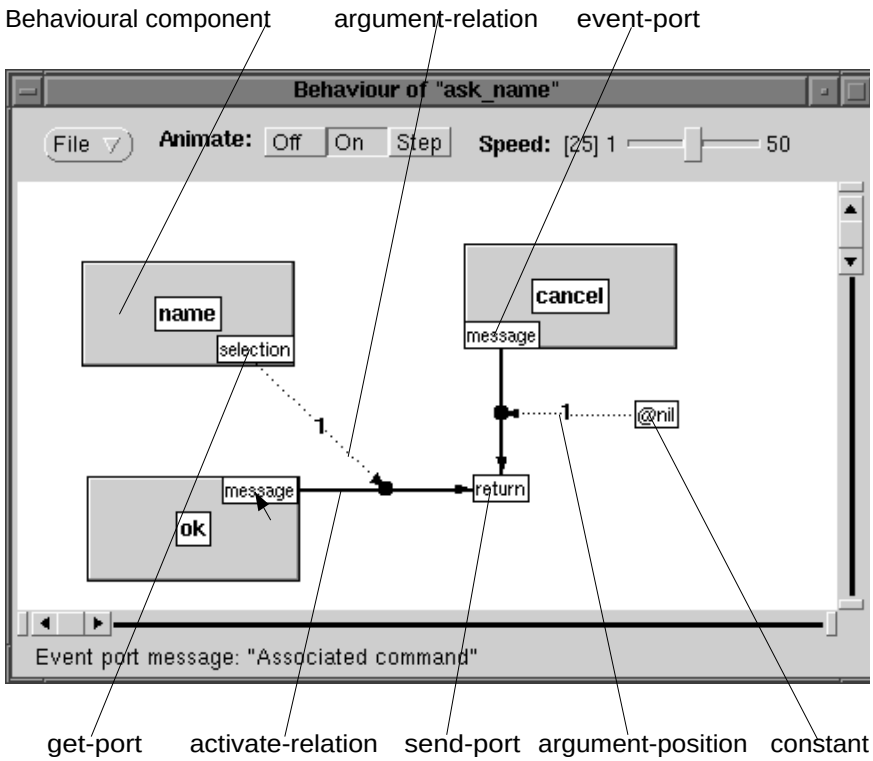


Figure A.4: Behaviour model of Ask Name

dialog(*?Identifier, ?Description***)**

Clause as generated by the dialog editor. The description is a list of ⟨*Name*⟩ `:=` ⟨*Value*⟩ pairs. It contains the following elements:

- object
  Points to the Prolog variable defining the main object reference (the first argument of `make_dialog/2`).
- parts
  A list of ⟨*Var*⟩ `:=` ⟨*NewTerm*⟩. `make_dialog/2` will simply call `new/2` on these terms to create the parts of the dialog window.
- modifications
  List of ⟨*Var*⟩ `:=` ⟨*ListOfModifications*⟩ that have to be applied to the parts to modify them from the default to the target configuration. The ⟨*ListOfModifications*⟩ is a list of ⟨*Attribute*⟩ `:=` ⟨*Value*⟩.
- layout
  List of `below(`⟨*Part1*⟩`,` ⟨*Part2*⟩`)` and `right(`⟨*Part1*⟩`,` ⟨*Part2*⟩`)`, describing the symbolic layout of the dialog window.
- behaviour
  List of ⟨*Part*⟩ `:=` ⟨*ListOfBehaviour*⟩, describing the behaviour of the control element. ⟨*ListOfBehaviour*⟩ is a list of ⟨*Attribute*⟩ `:=` ⟨*Message*⟩, describing the `code` objects associated with the controls.

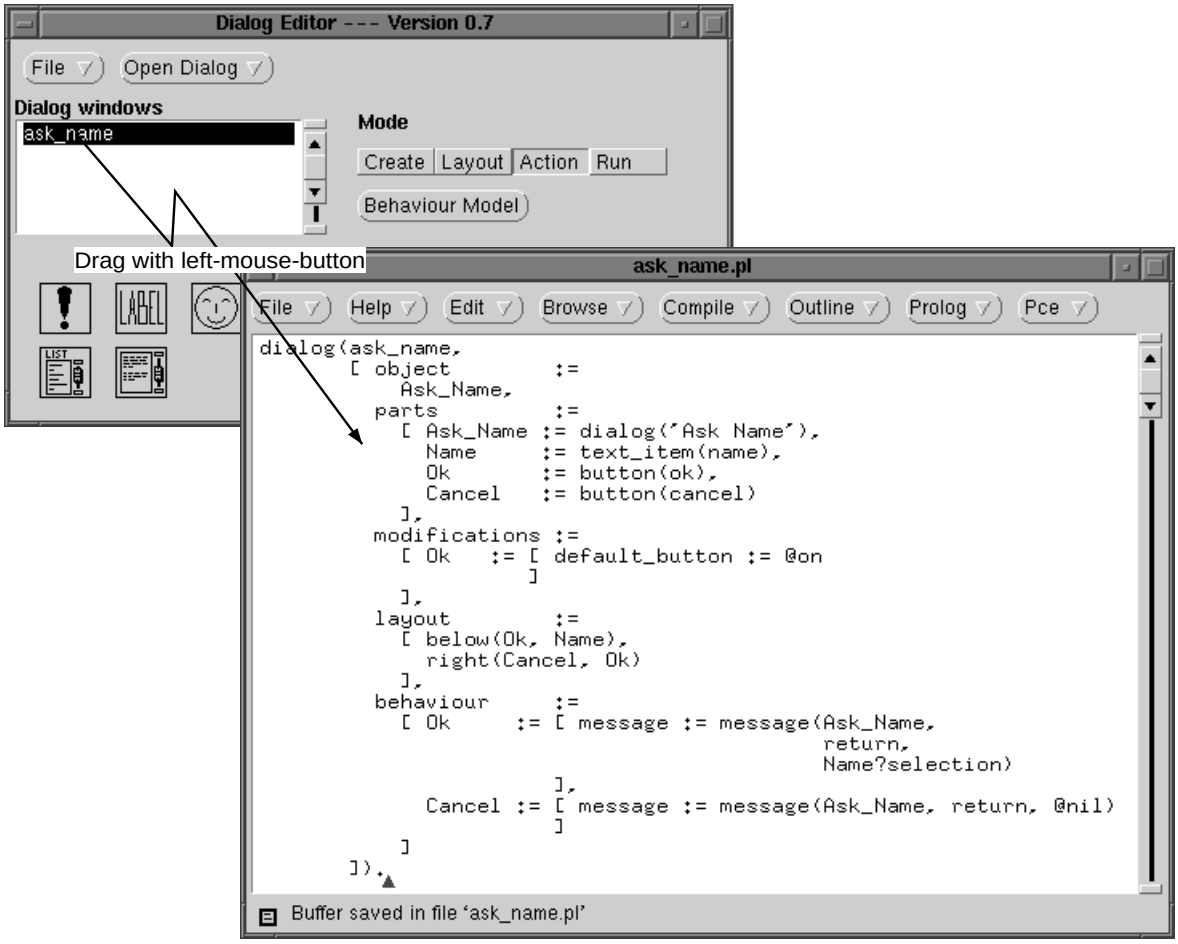


Figure A.5: Creating source-code

The wrapper program to make the dialog-description useful from an application is given below. First `make_dialog/2` is used to create the dialog. Next the dialog is opened in the center of the display and the system waits for the '`frame → return`' message to be send.

```
ask_name(Name) :-
        make_dialog(D, ask_name),
        get(D, confirm_centered, RawName),
        send(D, destroy),
        Name = RawName.
```

A.1.7 Summary

We have now completed the first guided tour through the Dialog Editor, introducing the main concepts and the process of creating a dialog window using this tool. After creating a target dialog window, controls are added to the dialog using drag-and-drop. Their attributes are edited by double-clicking the new controls and filling the subsequently shown dialog window. Next, the items are dragged roughly to the right location, the editor is placed in layout mode and the layout button is used to let the Dialog Editor guess the symbolic layout description. Next the behaviour is defined using the behaviour model editor. Event-ports (control-attributes defining the callback of a control) are linked to send-ports (send-method ports) and arguments are linked to this activation relation. Finally the dialog window is dropped in a PceEmacs window running in Prolog mode (the default when editing a file with extension `.pl` or `.pro`). Finally, a small wrapper must be defined that creates the dialog window from the description using `make_dialog/2` and opens the dialog in the proper way.

A.2 Miscellaneous topics

This section discusses various topics that were omitted from the Guided Tour to keep it simple.

A.2.1 Specifying callback to prolog

Using the background popup of the behaviour editor, the object `@prolog` (see section 6) can be added to the model. Select Add Send Port to add a new predicate to the `@prolog` interface. Then type the name of the predicate. Now link the event-port of a control to the predicate and link the arguments.

If the predicate is not defined, select 'Edit' on menu of `@prolog` to start PceEmacs on the source file. Now drag the predicate to the PceEmacs window. This will insert the head of the predicate at the caret. See figure A.6

A.2.2 Advanced example of behaviour

Figure A.7 is the screen dump of an application and its behaviour model of a tool that shows all files in a directory and clicking on a file shows the file's contents in the editor to the left. It demonstrates various aspects of advanced features for specifying behaviour.

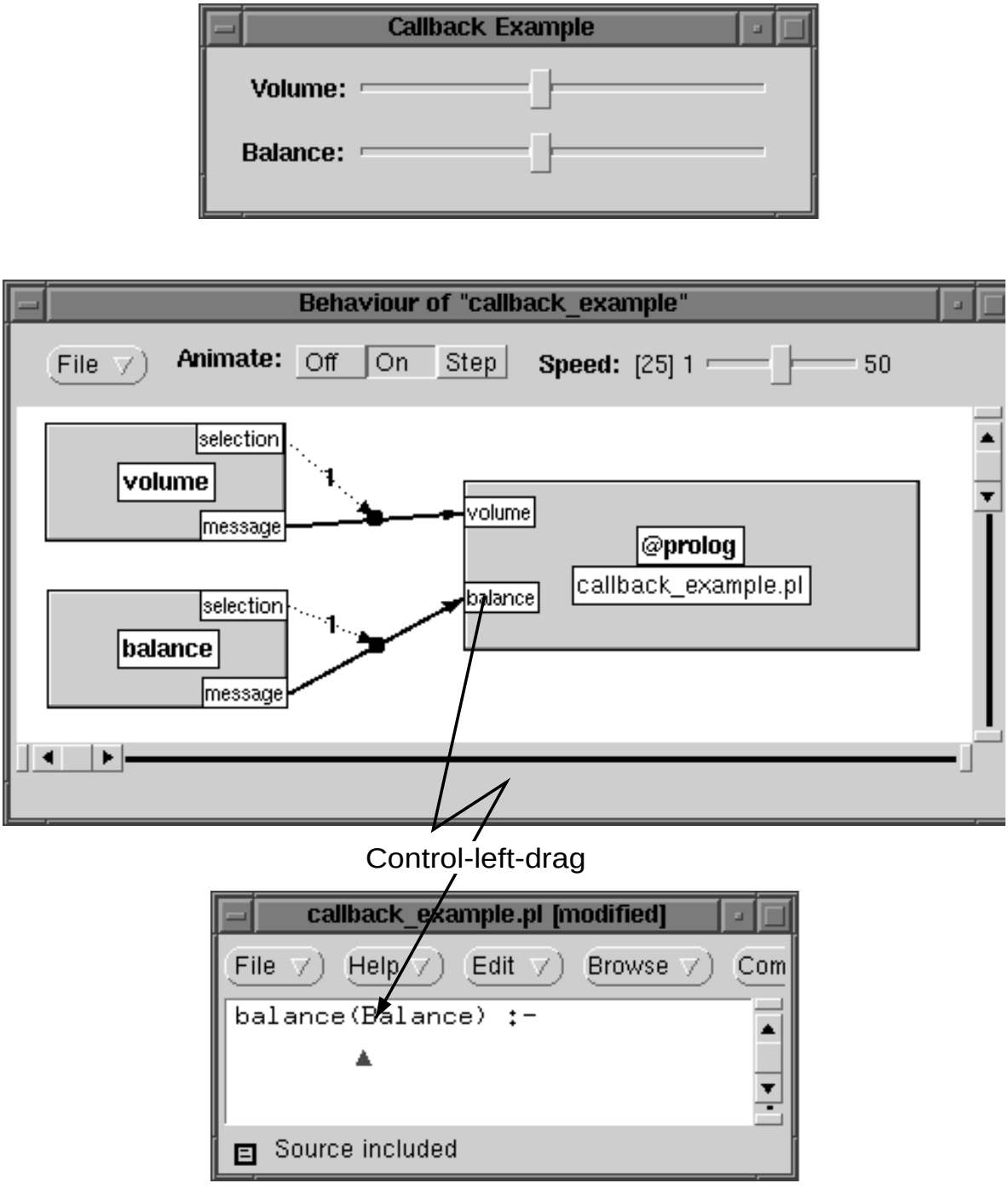


Figure A.6: Specifying Call-back to Prolog

- The text-field directory
  Represented in the model by (1). It is a normal `text_item`, but the '`text_item` ⇆ `type`' field is set to 'directory'. This implies the ←`selection` of the text-item will return a `directory` object.

- Showing the files of the directory
  If the text-item is modified, a list of files in the directory should be shown in the 'file_list', a `list_browser`. The method '`list_browser` → `members`' may be used to fill the browser with a collection of items. This method expects a `chain` object. The get method '`directory` ← `files`' provides a chain holding the names of all files in the directory.

  Thus, the event-port 'message' of the directory field must invoke the send-port 'members' of the file-list. The argument should be the result of applying ←`files` on the ←`selection` of the text-entry field. To specify this, the get-port 'selection' is *expanded* using the popup menu of this port. This operation adds (2) to the diagram. The system infers this expansion is an instance of class `directory` and shows the most useful get-ports in its get-port menu. The 'files' get-port is added to (2) and linked to the activation relation between '`text_item` → `message`' and '`list_browser` → `members`'.

  To **test** this part, put the Dialog editor in 'run' mode, type the name of a directory and ENTER to activate the event-port. If anything goes wrong, the Simulate option of the various popup menus in the diagram may be used to test small parts of the model. The *Documentation* option of these menus may be used to view the relevant documentation from the online manual tools.

- Specifying the initial directory as a parameter
  Initialisation of the dialog is expressed by adding one or more *init-ports* to the diagram using the background menu. In this particular case, we would like to be able to pass a directory to start as a parameter. Hence, a *parameter-port* is added with the name 'dir'. First, the →`selection` is set using the parameter and then the item is →`executed` to activate its →`message`.

  Code generation will append the directory parameter to the identifier of the `dialog/2` clause. In this case, this clause will start as:

```
dialog(viewer(Dir),
       [ ...
       ]).
```

  An instance is opened using the following calls:

```
...,
make_dialog(D, viewer(StartDir)),
send(D, open),
...
```

- Showing a file
  The first step is to link the event-port 'select_message' of the file_list to the file_contents (3) `editor` object's send-port 'load'. This method expects a file. Due to the type conversion rules of class `file`, the name of a file suffices, but the names from file_list are relative to the directory object (2). First, the ←`selection` of the file_list is expanded, resulting in (5), a `dict_item` object. The '`dict_item` ← `key`' contains the name of the file.

  The method '`directory` ← `file`' can be used to create a `file` object from a name, that specifies an absolute path. A get-port 'file' is added to (2) and this get-port is linked to the activation relation. This get operation requires the filename argument from (5).

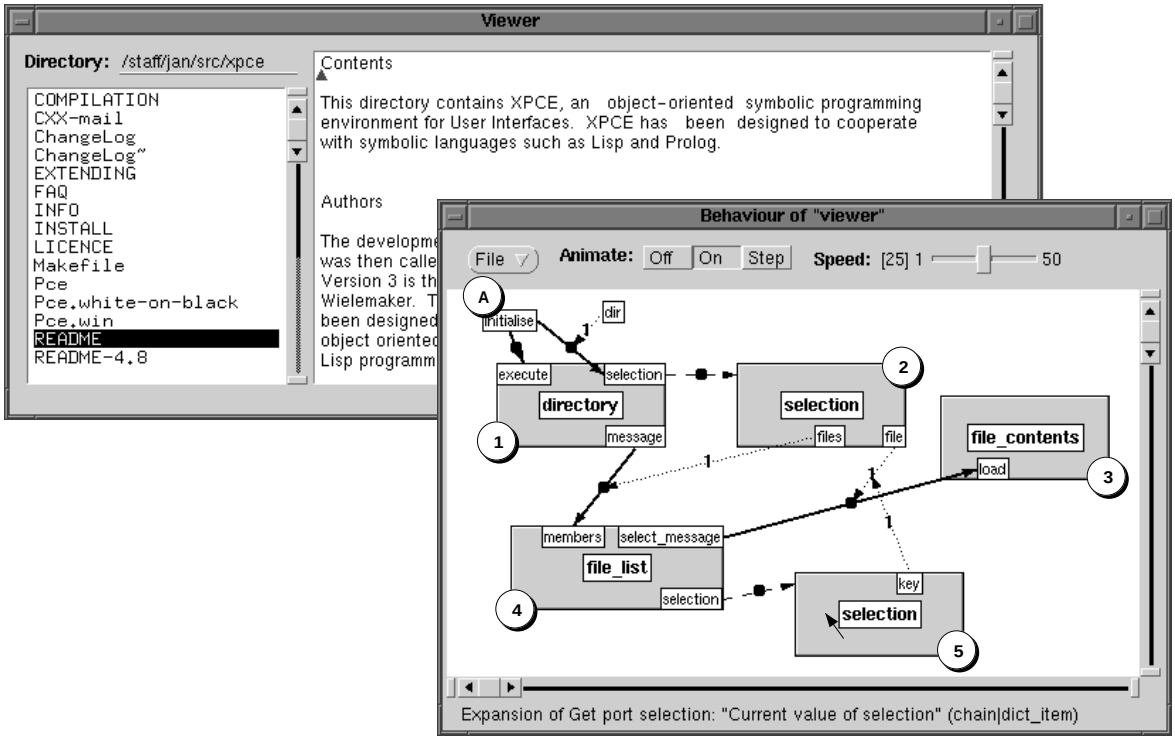


Figure A.7: A file viewer

A.2.3 Specifying conditional actions

Figure A.8 shows an example of a conditional activation relation. A conditional relation is created making a connection from the fat dot in the middle of an activation relation to a send-port. Success or failure of the send-port will be interpreted as a condition on the activation relation.

A.2.4 Load and save formats

The Dialog Editor provides two load/save formats. The Load, Save, Save As and Save All save and load the status of the dialog editor as an XPCE object using

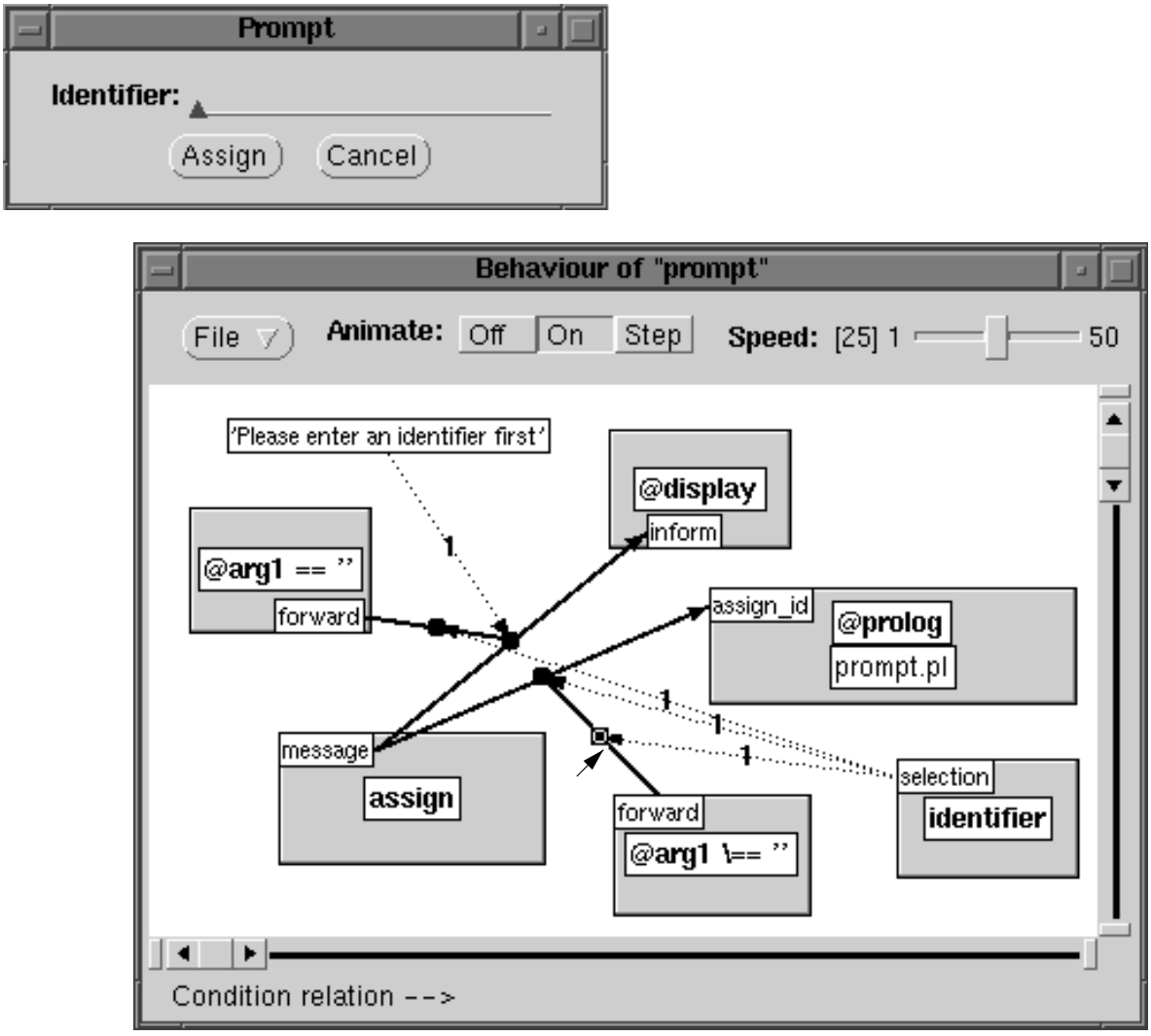


Figure A.8: Conditional activation

'`object → save_in_file`'. This format loads quickly, but is rather vulnerable to future changes in the Dialog Editor or any of the graphical classes.

Alternatively, the Dialog Editor can restore itself from the identifier of a `dialog/2` clause generated by itself or (slightly) modified. In this case, the layout of the behaviour model will be lost.

We advice to use Save/Load during application development. If, during the maintenance phase of your product it is necessary to modify a dialog, either simply edit the `dialog/2` clause, or load both the application and the dialog editor and use Reload From Id to restart the Dialog Editor.

A.3 Status and problems

The current version of the Dialog Editor is experimental. It can be used for serious application development as the output format is extensible, so future extensions to the Dialog Editor will not break already generated dialog windows. The main problems identified are:

- Defining new controls
  It is desirable to be able to create new (compound) controls using the dialog editor and save these in a library. At the moment new controls can only be created by programming them as a user-defined class. Connecting these user-defined controls to the Dialog Editor is not difficult, but no supported interface has been defined.

- Layout detection
  The layout detection often makes mistakes, partly because it does not know about various important layout concepts.

- Integration with user-defined classes
  It is desirable to integrate the dialog editor in a neat way with user-defined classes. Notably, the editor should support activating and defining methods on a user-defined refinement of the containing frame.

A.4 Summary and Conclusions

Though the dialog editor has attracted quite some attention when it was developed, it remains a difficult product. Using WYSIWYG style of interface building appears attractive, but looses generalisations that can be made in a programming language. If you have been in a country of which you don't speak the language you understand that pointing is a rather crippled way to express your needs. Especially XPCE/Prolog is strong in meta-representation and symbolic layout and the combination can easily be exploited to automate most of the simple control generation.

A good WYSIWYG should provide a smooth transition between the beginners choice for WYSIWYG and the expert choice of using language. It was one of the aims of this project to achieve this transaction but modern XPCE/Prolog applications are generally programmed in classes and the dialog editor presented here is build around direct relations between objects.

Notes on XPCE for MS-Windows

B

The same binary version of XPCE runs on both Windows 95 and NT. Its functionality is very close to the Unix/X11 version, making applications source-code compatible between the two platforms. .

XPCE does not build on top of the hosting window-systems GUI library. Instead, the primitive windowing and graphics facilities of the host are used to implement 'XPCE's Virtual Window System'. All of XPCE's graphical functionality is build on top of this 'Virtual Window System'. This approach guarantees full portability of applications between the platforms.

The look-and-feel of XPCE may be tailored using the defaults file located in ⟨*pcehome*⟩`/Defaults`.

B.1 Currently unsupported features in the Win32 version

- *Class socket*
  No support of file-based addressing (Unix domain sockets). Inet-domain sockets are provided (interfacing to WinSock).

B.2 Interprocess communication, extensions and interaction

- *DDE*
  Not (yet) supported by XPCE. SWI-Prolog supports it though, making DDE a feasible interprocess communication approach.
- *WinSock*
  Provides standard TCP/IP communication, both server- and client-side.
- *Named Pipes*
  Not (yet) supported.
- *OLE*
  Not considered yet. We however are considering CORBA, which provides an open standard for object-oriented, network-transparent interprocess communications. CORBA and OLE are integrated.
- *Drag-And-Drop*
  XPCE can accept dropped files from other applications. Inside the application, drag-and-drop is fully compatible to the Unix version. See library(`dragdrop`).
- *Cut/Paste*
  Supported for exchanging text, and pictures using the Windows MetaFile format.

- *DLL*
  Not supported by XPCE. SWI-Prolog provides it though, making DLL available to XPCE/Prolog applications.

B.3 Accessing Windows Graphics Resources

XPCE on Win32 defines the same cursor, colour and font-names as the Unix/X11 version to guarantee portability. It is desirable to have access to all the native Windows graphical resources. This allows the application to maintain better look-and-feel compatibility to other Win32 applications. Therefore the classes colour, cursor and font provide access to related Window resources.

It is NOT advised to use these objects in your application code directly as this will stop the application to run on the Unix/X11 version of XPCE. We advice using these objects in the XPCE defaults file (⟨*pcehome*⟩/Defaults) only, or use conditional code using 'pce ← window_system'.

B.4 Accessing Windows Colours

Colours may be created from their X11 names. The X11 name-table is in ⟨*pcehome*⟩/lib/rgb.txt. In the Windows API, all colours described as RGB (Red, Green, Blue) tuples. This is no problem as XPCE also provides RGB colours. Note however that Win32 intensity is ranged 0..255, where the XPCE intensity is ranged 0..65535. This is true on all XPCE's platforms.

To provide access to the window-system colours as they can be obtained using the Win32 API function GetSysColor(), XPCE binds these colours to named colour objects. These colour objects are normally used in the XPCE resource file (⟨*pcehome*⟩/Defaults) to colour XPCE's controller objects according to the user's preferences.

If the name of the Windows API colours are COLOR_SOMETHING, the XPCE name is win_something. The full list is in table B.1.

B.5 Accessing Windows Fonts

The normal screen, helvetica, roman and times font families available in the Unix/X11 version are available using the same names. The system will try to use an as close as possible equivalent Windows TrueType font for these.

The Windows 'stock' fonts as available from the GetStockObject() API are available under the special 'family' "win". They are in table B.2

Note that these fonts do not have a specified point-size. Their point-size depends on the Windows installation. The get-method ←points will return the ←height of the font.

Other Windows fonts may be accessed using a similar method as in Unix/X11: provide a fourth argument describing the font using the hosts conventions. For the Win32 API, this is a textual description of the Windows API structure LOGFONT passed to CreateFontIndirect(). The description is a ':' (colon) separated list of attributes of the structure. The attributes need not be specified in the order of the structure-layout. Omited attributes are set to their default.

win_activeborder	Active window border.
win_activecaption	Active window caption.
win_appworkspace	Background colour of (MDI) applications.
win_background	Desktop.
win_btnface	Face shading on push buttons.
win_btnshadow	Edge shading on push buttons.
win_btntext	Text on push buttons.
win_captiontext	Text in caption and size box
win_graytext	Grayed (disabled) text.
win_highlight	Item(s) selected in a control.
win_highlighttext	Text of item(s) selected in a control.
win_inactiveborder	Inactive window border.
win_inactivecaption	Inactive window caption.
win_inactivecaptiontext	Color of text in an inactive caption.
win_menu	Menu background.
win_menutext	Text in menus.
win_scrollbar	Scroll bar gray area.
win_shadow	Color of automatic window shadows.
win_window	Window background.
win_windowframe	Window frame.
win_windowtext	Text in windows.

Table B.1: Windows colour name mapping

font(win, ansi_fixed)	Default ANSI encoded fixed font
font(win, ansi_var)	Default ANSI encoded variable font
font(win, device_default)	Default device font
font(win, oem_fixed)	Computers ‘native’ fixed font (PC)
font(win, system)	Variable pitched system font
font(win, system_fixed)	Fixed system font

Table B.2: Windows font name mapping

`charset`	ansi
`height`	⟨*points*⟩ × $font.scale$
`weigth`	bold if ⟨*style*⟩ is bold, normal otherwise
`italic`	TRUE if ⟨*style*⟩ is italic or oblique
`pitch`	fixed if ⟨*family*⟩ is screen
`family`	swiss if ⟨*family*⟩ is helvetica, roman if ⟨*family*⟩ is times, modern if ⟨*family*⟩ is screen dontcare otherwise.
`face`	⟨*family*⟩

Table B.3: Windows font defaults

Attributes come in four types: numeric, boolean, enumerated and string. In general, an attribute is specified as:

⟨*name*⟩ (⟨*value*⟩)

⟨*name*⟩ is matches case-insensitive against the name of the structure field without the leading 'lf' string. For numeric types, the argument is interpreted as a decimal number (spaces are not allowed). For a boolean argument, the (value) part is omitted. By default the boolean attributes are FALSE. Including the attribute name in the specification sets the field to TRUE. Enumerated fields are specified using their symbolic name. Name-matching is case-insensitive. Common parts of the API identifier to make the symbol unique (for example _CHARSET in ANSI_CHARSET) are removed. String arguments simply take the value between the brackets. Spaces are included in the output, case is not changed and there is no escape for the closing-brace.

The default settings are in table B.3, the attributes are in table B.4.

The following example binds the Windows 'WingDings' symbol-font:

```
1 ?- new(F, font(wingdings, roman, 20, 'charset(symbol)')).
```

The following example uses this font to create an image from such a character:

```
:- send(@display, font_alias, wingdings,
        font(wingdings, roman, 20, 'charset(symbol)')).

wingding_image(Index, Image) :-
        new(Image, image(@nil, 32, 32)),
        new(T, text(string('%c', Index), center, wingdings)),
        send(T, center, point(16, 16)),
        send(Image, draw_in, T),
        send(T, done).

test :-
        wingding_image(60, Floppy),
        send(label(test, Floppy), open).
```

`height`(int)	point-size of the requested font
`width`(int)	average width of the characters
`escapement`(int)	angle in 1/10 degrees of the baseline
`orientation`(int)	angle for each character
`weigth`(int)	0..1000 scale for thickness
`italic`	request italic look
`underline`	underline all characters
`strikeout`	use strikeout-fonts
`charset`(enum)	character encoding {`ansi, oem, symbol`}
`outprecision`(enum)	accurate aspects {`character, default, string, stroke`}
`clipprecision`(enum)	how the characters clip {`character, default, stroke`}
`quality`(enum)	Quality of output {`default, draft, proof`}
`pitch`(enum)	Spacing attributes {`default, fixed, variable`}
`family`(enum)	Style of the characters {`decorative, dontcare, modern, roman, script, swiss`}
`face`(string)	Use specific font database

Table B.4: Windows font attributes

B.6 Accessing Windows Cursors

The Win32 version of XPCE supports all the X11 cursors. It also supports the definition of cursors from images. Note that such cursors are generally limited to 32×32 pixels on Windows (formally there is no limit in X11, but many (colour) servers exhibit strange behaviour when given cursors larger than this size). The window cursor names are in table B.5. Use the File/Demo/Cursors entry from the PCE Manual to inspect all available cursors.

The distributed `Defaults` file assigns `win_arrow` as the default cursor under Windows. `win_ibeam` is the default editor cursor and `win_wait` is the default wait cursor.

win_arrow	Default Windows arrow
win_ibeam	Like xterm
win_wait	hour-class (good replacement of ‘watch’)
win_cross	like crosshair
win_uparrow	long up arrow (no good X11 replacement)
win_size	A bit like ‘fleur’
win_icon	(see icon and icon_cross)
win_sizenwse	NorthWest - SouthEast arrow (no X11 replacement)
win_sizenesw	NorthEast - SouthWest arrow
win_sizewe	West - East arrow
win_sizens	North - South arrow
win_sizeall	as win_size
win_no	Stop-sign
win_appstarting	Arrow with hour-class

Table B.5: Windows cursor name mapping

C

XPCE/Prolog architecture

In this appendix we present an overview of XPCE's primitives and the interaction to the XPCE/Prolog environment.

C.1 What is "Object-Oriented"?

XPCE is an object-oriented system. This implies that the basic entity in XPCE's world is an object, an entity with state capable of performing actions. Such an action is activated by sending the object a *message*.

So far, most object oriented systems agree. Starting from these notions however one can find object oriented environments that take widely different approaches for representing objects, actions on objects and sending messages.

Rather than specifying operations on each individual object most OO environments define some way of sharing the operation definitions (called *methods*). There are two ways to share methods. One is to create objects as a copy of other objects and then modify them (by attaching and deleting slots and methods) to fit the particular need. If a series of similar objects is needed, one first creates an object that satisfies the common functionality and then creates multiple copies of this object. This approach is followed by SELF [Chambers *et al.*, 1989]. The other —more traditional— approach is to define a *class*. A class is an entity in the object oriented environment that defines the constituents of the persistent state and the methods for each of its *instantiations*.

XPCE takes the latter approach, but adds some notions of the object-copying approach because GUI's often contain unique objects and because object modification is more dynamic and therefore more suitable for rapid prototyping.

C.2 XPCE's objects

More concretely, a XPCE object is a set of *values* of *instance variables* bundled into a single entity which is referred to by its *object reference*. An object is an instantiation of a *class*. A class holds the key to decoding the information of its instances:[1] the instance variables. The class also serves as a placeholder for storing the methods understood by its instances. Figure C.1 illustrates this.

[1] We will mix the terms *instance* and *object* freely in this document. They are considered synonyms.

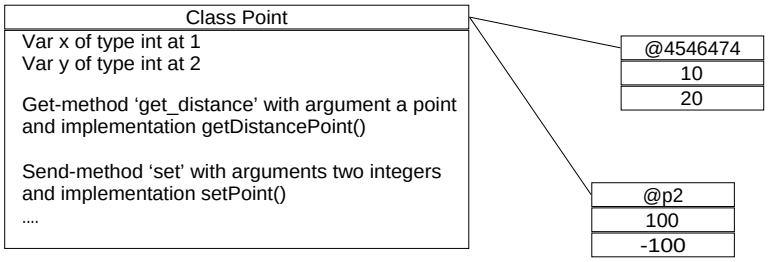


Figure C.1: Classes and Objects in XPCE

C.2.1 Classes

As explained above, a XPCE class describes the storage-layout and the methods of its instances. In XPCE a class is a normal object. It is an instance of class *class*.[2] As in most OO systems XPCE classes may inherit from a *super-class*. XPCE classes are organised in a single-inheritance hierarchy.[3] The root of this hierarchy is class *object*. Class object is the only class without a super-class. Figure C.2 gives the complete hierarchy of XPCE built-in classes.

C.3 Objects and integers

Except for integers, everything accessible to the user is represented as an object. By implementing classes, instance variables, methods, messages, conditions, constants, variables, etc. as objects everything in XPCE may be accessed through the basic predicates `new/2`, `send/[2-12]` and `get/[3-13]` from Prolog.

C.4 Delegation

XPCE does not offer multiple inheritance. Sharing functionality from multiple classes is generally dealt with using *delegation*. Delegation implies that messages not understood by a principal object are forwarded to an object that is associated to it.

For example, XPCE defines class `editor` to be a graphical object capable of editing text. Most applications require a `window` capable of editing text. This is implemented by XPCE's class `view`, which is not a subclass of both editor and window, but just of window. The window displays an instance of class `editor` and constrains the size of the editor to occupy the entire visible area of the window. Any message arriving on the view that is not defined on class `view` (or class `window`) will be forwarded to the associated editor object.

The dynamic nature of delegation makes this mechanism more flexible than multiple inheritance. For example, XPCE defines class `node`. This class defines the communication

[2] Class class is an instance of itself. In other systems (SmallTalk, [Goldberg & Robson, 1983]), classes are instances of a *meta-class*. Yet in other systems, classes have a completely different status (for example widgets in the X11 Intrinsics)

[3] Multiple inheritance introduces various technical and conceptual problems. XPCE uses delegation and templates to achieve similar results. This is explained in section C.4 and section 7.5.2.

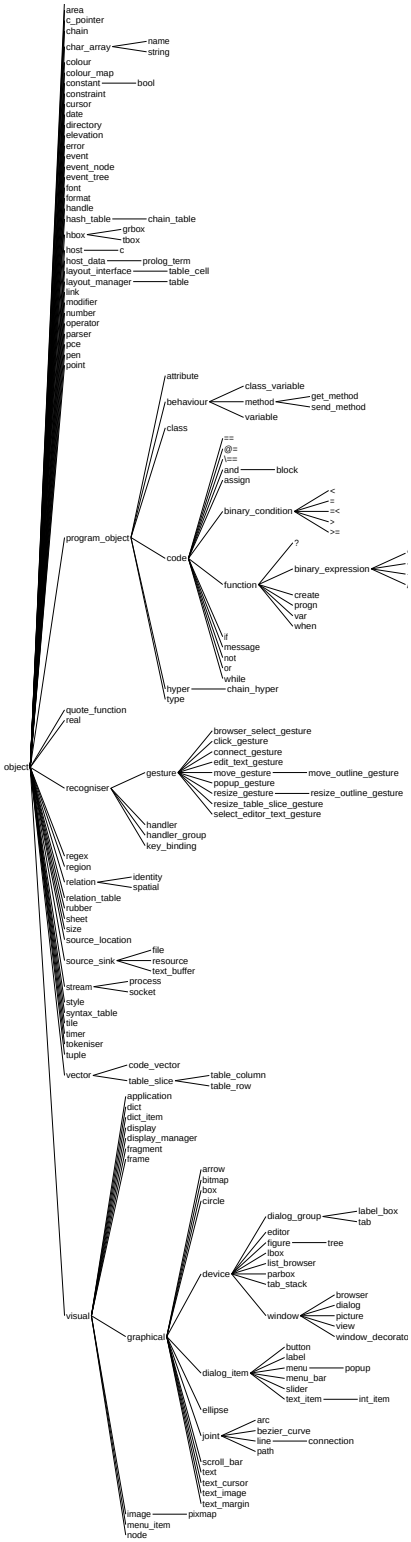


Figure C.2: XPCE's Class hierarchy

to a `tree` to automate the layout of hierarchies. A node can manipulate any graphical object. Using multiple inheritance would require a class *box_node*, *circle_node*, etc.

C.5 Prolog

As we have seen in section 2, activating XPCE is done by providing the user with access to XPCE's message passing primitives. Near the end of section 2 we briefly explained how control is passed from XPCE to Prolog. The predefined object `@prolog` is (the only) instance of class *host*. Any message sent to this instance will be mapped on a Prolog goal and given to the Prolog system as a query: the *selector* of the method will be used as a predicate name. The arguments will be translated from XPCE data-types to their corresponding Prolog data-types according to the transformation rules described in section D.

The relation between XPCE and Prolog is described in detail in chapter 6. Examples can be found throughout this manual.

Figure C.3 shows the data- and control-flow between XPCE and Prolog. The lines with arrows indicate data-flow in the direction of the arrow. The dotted ellipse with arrows indicates the flow of control.

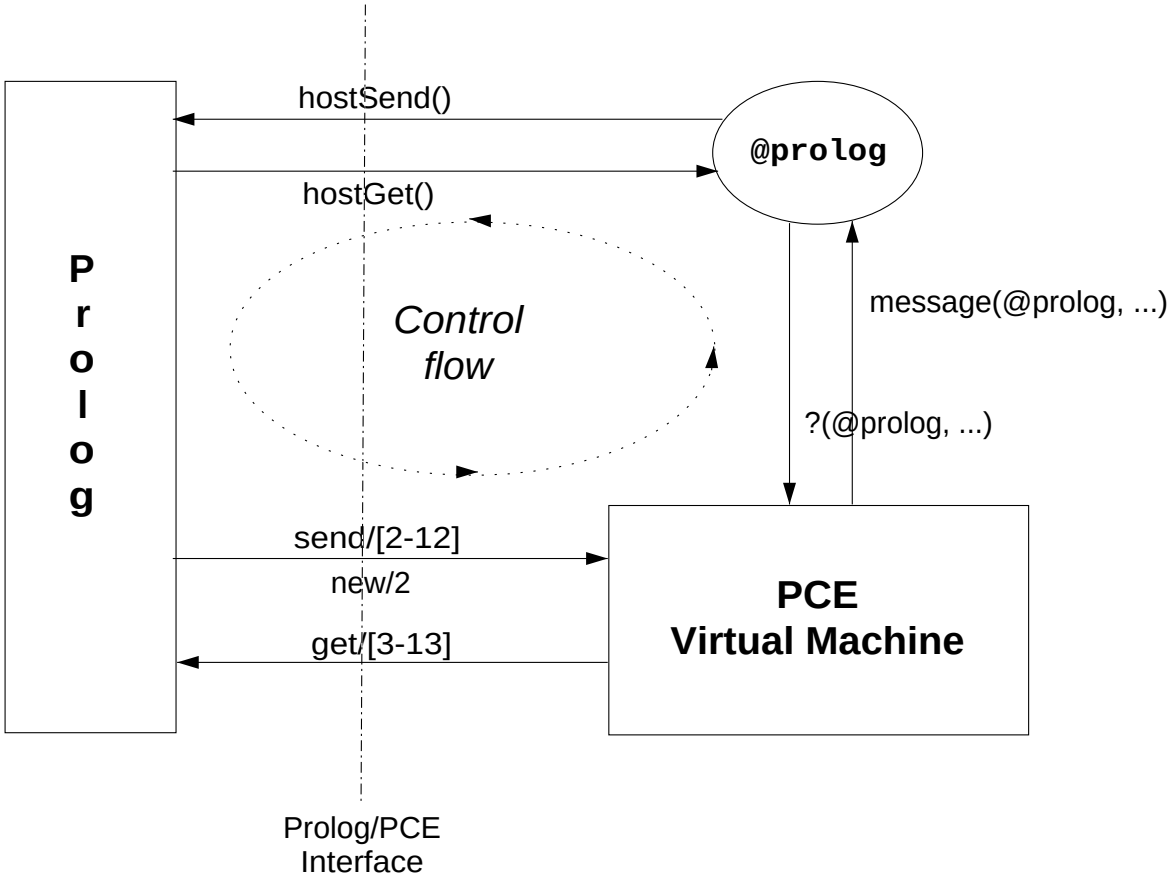


Figure C.3: Data and Control flow in XPCE/Prolog

C.6 Executable objects

Executable code (statements, control-structures, variables, etc.) can be expressed as first-class objects. Such expressions can be associated with controls to specify their actions, to method objects to specify their implementation and as arguments to method invocation to specify details of the operation to be performed.

Executable objects are used in many of the examples in this manual. Section 10.2 provides an overview of them.

C.7 Summary

This section explained the basic object-oriented notions used in XPCE. XPCE's data is organised in *objects* and integers. An object represents a state. An object is an instance of a class. A class describes the constituents of the state represented in its instances and the methods (actions) understood by its instances.

A class is a normal object, as are all the other constituents of XPCE's programming world: methods, instance variables, messages, expressions, etc. This uniform representation allows for inspecting and changing XPCE using the four basic interface predicates from Prolog.

The basic interface predicates pass control from Prolog to XPCE. As control is to be passed from XPCE to Prolog (for example if the user presses a button), a message is send to `@prolog`, the only instance of class host. This object will create a goal from the message and pass this goal to the Prolog environment.

Interface predicate definition D

This appendix provides a description of the Prolog predicates to communicate with PCE. Most of these predicates have been introduced informally in the previous sections.

D.1 Basic predicates

This section describes the basic interface predicates. These predicates reside in the library module 'pce', which is loaded when Prolog is started with PCE loaded on top of it.

new(*?Reference, +TermDescription***)**

Create a XPCE object from *TermDescription* and either unify an integer reference (e.g. @2535252) with *Reference* or give the new object the provided atomic reference (e.g. @my_diagram). The argument *TermDescription* is a complex term of the form Functor(...InitArg...). *Functor* denotes the class from which to create an object and *InitArg* are the initialisation arguments for the object creation. Each *InitArg* is translated to a XPCE data object using the following rules:

- *atom*
  Atoms are translated into XPCE name objects. This is a loss-less transformation.
- *integer*
  Prolog integers are translated into XPCE int data-types. The range of XPCE int is half that of Prolog (i.e. $\pm 2^30$ on a 32-bit machine).
- Class*(+InitArg...)*
  Creates an instance of class *Class* using *InitArg*.
- new*(+TermDescription)*
  Same as plain *InitArg*, but an atom is translated to an instance of the named class. E.g. the term new(chain) is translated to an empty chain object rather then the atom chain.
- new*(?Reference, +TermDescription)*
  Same as new/2, handling *Reference* the same as the predicate new/2.
- prolog*(Term)*
  Pass *Term* as an unmodified Prolog term.

Below we illustrate the use of embedded new/2 terms in *InitArg* to get access to the reference of in-line created objects. The examples are functionally equivalent.

```
1 ?- new(@icon_viewer, dialog('Icon Viewer 1')),
```

```
        new(P, picture),
        send(P, below, @icon_viewer),
        new(TI, text_item(name, ''),
                  and(message(P, display, @arg1),
                      message(@arg1, recogniser,
                              new(move_gesture)))),
        send(TI, type, bitmap),
        send(@icon_viewer, append, TI),
        send(@icon_viewer, open).

2 ?- D = @icon_viewer,
     new(D, dialog('Icon Viewer 1')),
     send(new(P, picture), below, D),
     send(D, append,
          new(TI, text_item(name, ''),
                    and(message(P, display, @arg1),
                        message(@arg1, recogniser,
                                new(move_gesture))))),
     send(TI, type, bitmap),
     send(D, open).
```

Using `new/2` with a variable reference argument is equivalent to invoking '`Class ← instance`: InitArgs ...'. The arguments needed to instantiate a class are defined by the →`initialise` method of this class. See also section 3.3.1.

send(*+Receiver, +Selector(+Argument...)***)**

send(*+Receiver, +Selector, +Argument...***)**

Invoke a send-method on the *Receiver*. *Receiver* is processed as the *InitArgs* described with `new/2`. This implies that a complex term is translated into an object before the method is invoked. An atom is translated into an XPCE name object. *Selector* is a Prolog atom which is translated into a XPCE name object. The *Argument*s are processed as the *InitArgs* described with `new/2`.

The predicate `send/[2-12]` fails with an error message if one of the arguments cannot be translated or there is a type-error or an argument-error. The method itself may also produce error messages. This predicate only succeeds if the requested method was executed successfully.

Trailing arguments that can handle `@default` (indicated by square brackets in the type declaration) may be omitted.

If the method accepts many arguments of which most are default, using the named argument convention may be preferred. For example:

```
...,
send(Graphical, graphics_state, colour := red),
...,
```

The first form using `Selector`(*Argument...*) is the principal form. The second is translated by the XPCE/Prolog macro-layer and available for compatibility and style-preference.

get(+*Receiver, +Selector(+Argument...), -Result***)**

get(+*Receiver, +Selector, +Argument..., -Result***)**

Invoke a get-method on *Receiver*. *Receiver*, *Selector* and *Argument...* are processed as with `send/[2-12]`. If the method fails, this predicate fails too. Otherwise the XPCE result of invoking the method is unified with *Result*.

If the return value is a XPCE integer, real object or name object, it is unified with a Prolog integer, float or atom. Otherwise if the Prolog return argument is a variable or a term @/1 it is unified with the object reference. Otherwise the Prolog argument should be a compound term. Its functor will be compared with the class-name of the XPCE return value. The arguments will be unified in the same manner with the term-description arguments as declared with the class. Examples:

```
1 ?- get(@pce, user, User).
User = fred
2 ?- get(@display, size, Size).
Size = @474573
3 ?- get(@display, size, size(W, H)).
W = 1152, H = 900
```

It is not advised to use the latter construct for other objects than elementary objects such as point, area, size, string, etc..

free(+*Reference***)**

Send →`free` to *Reference* if it is a valid reference. Defined as

```
free(Ref) :- object(Ref), !, send(Ref, free).
free(_).
```

This definition implies `free/1` only fails if the object may not be freed (see '`object → protect`').

send_class(+*Reference, +Class, +Selector(+Arg...)***)**

get_class(+*Reference, +Class, +Selector(+Arg...), -Result***)**

send_super(+*Reference, +Selector(+Arg...)***)**

get_super(+*Reference, +Selector(+Arg...), -Result***)**

send_super(+*Reference, +Selector, +Arg...*)

get_super(+*Reference, +Selector, +Arg..., -Result*)

The predicates `send_class/3` and `get_class/4` invoke methods on a super-class of the class *Reference* belongs to. In most cases methods access the *immediate* super-class and this is the function of `send_super/[2-12]` and `get_super/[3-13]`.

The `*_super` calls are macro-expanded to `send_class/3` or `get_class/4`. They **must** appear within a XPCE class definition. Though not enforced, using any of these predicates or macros outside the context of a method-definition should be considered illegal. See chapter 7 for further discussion on defining classes and methods.

object(+*Reference*)

Succeeds if *Reference* is a term of the form @/1 and the argument is a valid object reference. Fails silently otherwise. Note that the form `@Integer` is only save to test whether or not an object has already been freed as a side-effect of freeing another object. Consider the following example:

```
1 ?- new(P, point(100,100)).
P = @235636/point
2 ?- free(@235636).
3 ?- object(@235636).           ---> fail
4 ?- new(S, size(50,50)).
S = @235636/size
```

If →`free` is invoked on an object that has no references, its memory will be reclaimed immediately. As long as the memory has not been reused `object/1` is guaranteed to fail. If the memory is reused for storing a new object `object/1` will succeed, but point to another object than expected. Finally, the memory may be reused by a non-object data structure. In this case `object/1` only applies heuristics to detect whether the memory holds an object. See also section 12 and section 10.3.3

object(+*Reference, -TermDescription*)

Unify object description with the argument. Normally only used for debugging purposes. Equivalent to:

```
object(Ref, Term) :-
        object(Ref),
        get_object(Ref, self, Term).
```

:- pce_global(+*Reference, :Create*)

Define exception handler for undefined global (named) reference. When XPCE refers to a non-existing named reference an exception is raised. The standard handler for this exception will scan the `pce_global/2` database and execute the *Create* action. *Create* is either a term of the form new(+TermDescription) or another term. In the first case *TermDescription* is transformed into a XPCE object as the second argument of `new/2`. In the latter case, *Reference* is appended to the list of arguments of the term and the term is called as a Prolog goal:

```
:- pce_global(@succeed, new(and)).
:- pce_global(@event_receiver,
              new(@event?receiver)).
:- pce_global(@select_recogniser,
              make_select_recogniser).

make_select_recogniser(R) :-
        new(G, handler_group),
        send_list(G, append,
          [ click_gesture(left, '', single,
                  message(@event_receiver?device,
                          selection, @event_receiver))
          , click_gesture(left, s, single,
                  message(@event_receiver,
                          toggle_selected))
          ]).
```

See section 6 for more examples.

pce_open(+*Object*, +*Mode*, -*Stream***)**

The predicate `pce_open/3` opens an XPCE object as a Prolog stream. Using this stream, the normal Prolog I/O predicates for reading from, or writing to the object can be used.

This predicate works on any object that implements the *as_file methods. Currently this is only implemented for class text_buffer. See '`text_buffer ← read_as_file`', '`text_buffer ← size_as_file`', 'text_buffer →`truncate_as_file`' and '`text_buffer → write_as_file`'.

The stream handle is discarded using Prolog's `close/1` predicate. For example, to write to a view, one could use:

```
...
pce_open(View, append, Stream),
format(Stream, 'Hello World~n', []),
close(View),
...
```

See also '`text_buffer → format`'. Reading from a stream is used by the PceEmacs editor to verify the syntax of an entered clause.

pce_catch_error(+*ErrorIds*, +*Goal***)**

This predicates allows the application to handle errors occuring while *Goal* is called. *ErrorIds* is either an atom representing the id of XPCE error or a chain of such id's. If one of the given errors occurrs the goal will silently fail and '`@pce ← last_error`' holds the id of the trapped error. Any other error that occurs during the execution of *Goal* will be handled by XPCE's normal error handling mechanism. See section 10.8.

D.1.1 Portable declaration of required library predicates

Different Prolog implementations to which XPCE has been connected provide a different library structure and offers different means for accessing library predicates. For this reason, XPCE introduced the `require/1` directive. This directive is the preferred way to import library predicates. Below is a typical declaration of an XPCE/Prolog module:

```
:- module(mymodule, [myapp/0]).
:- use_module(library(pce)).
:- require([ member/2,
             send_list/3
           ]).
```

require(*:ListOfNameArity***)**

Defines that this module requires the named predicates. It is the task of the Prolog system to make sure the module can make calls to the named predicates and this predicate has the 'commonly accepted semantics'. This predicate is built-in for SICStus and SWI-Prolog. It is defined in the module library(pce) for ProWindows-3/Quintus. This is the reason why library(pce) should always be imported explicitely.

Note the command Pce/PceInsertRequireDirective in PceEmacs Prolog mode, which automatically determines the required require-directive for the current module-file.

auto_call(*:Goal***)**

Acts like `call/1`, but dynamically loads the predicate required by *Goal* if this predicate is not defined. On systems not having autoloading, the definition is:

```
auto_call(Goal) :-
        strip_module(Goal, Module, Predicate),
        functor(Predicate, Name, Arity),
        require(Module:[Name/Arity]),
        Goal.
```

D.2 Additional interface libraries

This section describes Some of the predicates available from the XPCE/Prolog library.

D.2.1 Library "pce_util"

The predicates in this section used to be XPCE principal predicates. Changes to XPCE, the interface and our current understanding about programming the XPCE/Prolog environment have made these predicates less important.

send_list(*+Receiver, +Selector [, +Argument]***)**

Invoke send-behaviour as `send/[2-12]`. Each of the arguments is either as accepted by `send/[2-12]` or a list of such arguments. The definition of `send_list/2` is below.

```
send_list([], _) :- !.
send_list(_, []) :- !.
send_list([Object|Objects], Selectors) :- !,
        send_list(Object, Selectors),
        send_list(Objects, Selectors).
send_list(Object, [Selector|Selectors]) :- !,
        send_list(Object, Selector),
        send_list(Object, Selectors).
send_list(Object, Selector) :-
        send(Object, Selector).
```

Note that, since `send/2` accepts `Selector`(*Arg...*) the following is now valid code:

```
        ...,
        send_list(Box,
                  [ colour(red),
                    fill_pattern(colour(greed))
                  ]),
```

get_object(*+Receiver, +Selector, +Argument..., -Result***)**
Equivalent to `get/[3-13]`, but instead of unifying a variable with a reference the variable is unified with the term-description. The arguments are unified as in `get/[3-13]`. Normally only used from the Prolog top level for debugging purposes.

chain_list(*?Chain, ?List***)**
Converts between a XPCE chain and a Prolog list. This may be useful to exploit Prolog's list-processing primitives. Note however that XPCE chains define various operations that may be exploited to avoid the translation. Suppose 'Pict' is a picture and 'Pos' is a point object. We want to determine the topmost graphical object overlapping with 'Pos'. The following two programs are identical:

```
topmost_graphical(Pict, Pos, Gr) :-
        get(Pict, graphicals, Grs0),
        chain_list(Grs0, Grs1),
        topmost(Grs1, Pos, @nil, Gr),
        Gr \== @nil.

topmost([], _, Gr, Gr).
topmost([H|T], Pos, _, Gr) :-
        send(H, overlap, Pos), !,
        topmost(T, Pos, H, Gr).
topmost([_|T], Pos, Gr0, Gr) :-
        topmost(T, Pos, Gr0, Gr).
```

Or, using XPCE's list processing:

```
topmost_graphical(Dev, Pos, Gr) :-
        get(Dev, graphicals, Grs),
        get(Grs, find_all,
            message(@arg1, overlap, Pos), O),
        get(O, tail, Gr),
        send(O, done).
```

The second implementation is not only shorter, it also requires far less data conversions between Prolog and XPCE and is therefore much faster.

get_chain(+*Receiver, +Selector, -List***)**
Utility predicate implemented as:

```
get_chain(Receiver, Selector, List) :-
        get(Receiver, Selector, Chain),
        chain_list(Chain, List).
```

See comments with `chain_list/2`.

D.2.2 Library "pce_debug"

The predicates in this section provide shorthands for common commands for debugging XPCE programs. See section 12 for more information on debugging XPCE/Prolog programs.

tracepce(+*Class* <- | -> | - *Selector***)**
Find send- (->), get- (<-) method or variable (-) and cause invocations thereof to be printed on the console.

Syntax note: (->) is a standard Prolog operator with $priority > 1000$. Therefore many Prolog systems require additional brackets:

```
1 ?- tracepce((graphical ->selected)).
```

In SWI-Prolog this is not necessary. To be able to trace get-methods with this predicate (<-) must be declared as an infix operator.

notracepce(+*Class* <- | -> | - *Selector***)**
Disables trace-point set with `tracepce/1`.

checkpce
Collect all global (named-) objects and run '`object` $\rightarrow$ `_check`' on them. This performs various consistency checks on the objects and prints diagnostic messages if something is wrong. '`object` $\rightarrow$ `_check`' checks all objects it can (recursively) find through slot-references, chains, vectors and hash-tables and deals properly with loops in the data-structure.

show_slots(+*Reference***)**
Prints the values of all instance variables of *Reference*:

```
1 ?- new(@move_gesture, move_gesture).
2 ?- show_slots(@move_gesture).
@move_gesture/move_gesture
        active                  @on/bool
        button                  middle
        modifier                @810918/modifier
        condition               @nil/constant
        status                  inactive
        cursor                  @default/constant
        offset                  @548249/point
```

A graphical tool for inspecting instance variables is described in section 12.5.

D.2.3 Accessing the XPCE manual

manpce

Start the XPCE online manual tools. This opens a small GUI console at the top-left of the screen, providing access to the manual, demo programs and tools described in this manual. See chapter 3.

manpce(+*Spec*)

As `manpce/0`, but immediately opens the the manual from *Spec*. *Spec* is either a class-name, opening the ClassBrowser, or a term Class `<-|->|-` Selector (see `tracepce/1`) to open the manual-card of the specified behaviour. Examples:

```
1 ?- manpce(box).
2 ?- manpce((view->caret)).
```

Memory management

E

This chapter describes the memory- and object-management aspects of PCE.

E.1 Lifetime of an object

Object lifetime management is a difficult issue in PCE/Prolog as PCE cannot be aware of all references to PCE objects stored in Prolog. Another complicating factor is that non-incremental garbage collection as performed by most Lisp systems is undesirable because they harm the interactive response of the system. For these reasons PCE performs *incremental* garbage collection. It distinguishes a number of prototypical 'life-cycles' for objects. Heuristics tell the system which of these is applicable and when the object may be deleted.

PCE distinguishes between *global*-, *top level*-, *support*-, *argument*- and *answer*- objects. *Global* objects are created and exist for the entire PCE session: `@prolog`, class objects, etc. *Top-level* objects are the principal objects of the application. They should exist even if no other PCE object refers to them. An example of a top level object is a frame or hash_table representing a database in the application. *Support* objects only complete the definition of other objects. If this 'other' object is removed, the support object may be removed as well. An example is the area attribute of a graphical. *Argument* objects are objects created to serve as an argument to a message. For example a graphical may be moved to a position described by a point object. The point may be deleted when the message is completed. Finally, *answer* objects are the result of some computation. For example '`area` ← `size`' returns a size object. This object may be deleted when the code that requested the value is done with it.

PCE maintains the following information on objects to support garbage collection. This information may be requested using the PCE inspector (see section 12.5).

Protect Flag
: This flag may be set using '`object` → `protect`'. When set, the object can not be freed by any means. This flag is set for most global and reusable objects: `@prolog`, `@pce`, `@display`, names, classes, etc.

Lock Flag
: This flag indicates that the object may not be removed by the garbage collector. Locked objects can only be freed by sending an explicit '`object` → `free`' message' or using the predicate `free/1`. It is used to avoid that 'top level' objects such as frames are deleted by the garbage collector. It is also used to indicate that Prolog wants to be responsible for destruction of the object rather than PCE's garbage collector. The lock flag is automatically set on any object that has a named reference. If Prolog wants

to store integer object references in the Prolog database locking is often necessary to protect the object for the PCE garbage collector. See also section 6.

Answer Flag
: This flag indicates that the object has been created as an answer of some computation or as a result of the Prolog predicate `new/2`. The *answer* status is cleared if the object is used to fill a slot of another object[1] or '`object → done`' is invoked on the object.

Reference Count
: PCE maintains the number of other objects referring to this object. When the reference count drops to zero and none of the protect, lock or answer flags are set PCE assumes the object is garbage and removes the object from the object base.

E.2 Practical considerations

The principal predicates `new/2`, `send/[2-12]` and `get/[3-13]` will destroy all argument- and answer- objects created during their execution except for the object created by `new/2` and the object returned by `get/[3-13]`.

An object created by `new/2` with an integer (anonymous) object reference must either be attached to another object, locked against PCE's garbage collector or destroyed using '`object → done`' if it is created during the *initialisation* of the application or in a loop that is passed many times. Such objects will be automatically reclaimed if (1) the object is created while handling a user-event after handling the event is finished or (2) the object is created in the implementation of a method on a user-defined class and the method terminates execution.

If is is not known whether or not the result of `get/[3-13]` is a computed object the user should invoke '`object → done`' to this result when the result is no longer needed. This will free the result if it was a computed (and no longer referenced) object and has no effect otherwise. If the result of the get operation is known to be an integer, no such message should be send.

E.3 Memory usage of objects

Currently an object consists of an object-header and an array of instance variables. The object-header includes various flags, a reference count and a pointer to the class. The size of an object header is 12 bytes. Each instance variable consumes an additional 4 bytes. For example a point object has 'x' and 'y' instance variables and thus consumes $12 + 2 * 4 = 20$ bytes.

The method '`class ← instance_size`' returns the size of an instance of this class in bytes. Note that the costs of supporting objects is not considered in this value. For example a box object has instance size:

```
1 ?- get(class(box), instance_size, S).
```

[1] PCE assumes the object has become a *support* object. This is generally not correct for code objects. Class `code` therefore has '`class ← un_answer: @off`', which implies that objects that fill a slot of a code object will not loose their 'answer' status.

```
S = 72
```

But a box has an ←area instance variable consuming an additional 28 bytes.

Commonly encountered problems

F

In this chapter we list a number of commonly encountered problems in using PCE/Prolog.

Cannot open display
: PCE tries to open the display from the address specified by the DISPLAY environment variable. It ignores the '–display' command line option. The display might also be specified explicitly using '`display` ⇆ `address`'. PCE will open the display as soon as it needs X-resource values or it needs graphical operations. This will fail if the specified address is not legal, there is no X-server at that address or the X-server denies the access. Examine the error message carefully. Make sure X-windows is running at the specified address. Make sure you have access to this server. See xauth (when running MIT_MAGIC_COOKIE) and xhost. If PCE still complains, validate the access rights by starting a normal X-application (e.g. xterm) in the same context. Always restart PCE after a fatal or system error as the system might be corrupted.

  This problem is not possible in the Win32 implementation.

Bad integer reference
: This is a PCE/Prolog interface warning. It implies the integer object reference given to `send/[2-12]`, etc. is not valid. The most common reason is that the object has already be freed, either explicitly or by PCE's incremental garbage collector. See section E.

Unknown class
: Attempt to create an instance of a non-existing class. Apart from the common mistakes like mistyped class-names, etc. this might be caused by 1) giving a list argument to a send- or get- operation (class '.') or 2) trying to pass a term through `send/[2-12]` or `get/[3-13]`. See section 6.1.

Illegal PCE object description
: This implies a non-translatable Prolog datum was passed to the interface. Normally this will be a non-ground[1] argument to `new/2`, `send/[2-12]` or `get/[3-13]`.

[1] A 'ground' term is a Prolog term that has no unbound variables.

Glossary

Attribute
: A `attribute` object is used to define additional properties of an object. The term *attribute* is also used as a synonym for *slot* and *instance-variable* referring to class defined properties.

Class
: A *class* is an object that acts as a description of other objects called *instances* of the *class*. Besides various house-keeping information, a PCE *class* describes the *instance-variables* and *methods* of its *instances*.

Class-Variable
: A *class-variable* defines a constant for all instances of the class. Class variables can be used to define default values for an *instance-variable*. Initial values for class-variables can be specified in the `Defaults` file. See section 8.

Code
: A *code* object is an object that represents a procedure. *Code* objects are used for implementation of methods and to associate actions with various events. For example a button object executes its associated code object when depressed. The most typical code object is a `message`.

Control
: A *control* is a standard *GUI* object normally placed in dialog windows. Examples are buttons, text-entry fields and menus.

Event
: An *event* is an object that represents an activity of the user: mouse-movements, mouse-buttons, keyboard activities.

Forwarding of argument
: When code objects are executed it is common to bind the *var* objects `@arg1`, `@arg2`, ... to pass context information for the executing code. For example, when a method object executes its code it will bind the arguments given to the method to `@arg1`, ...

Function
: A *function* is a subclass of class `code` which yields a value when executed. The most important functions are local variables (*var*), *obtainers* and mathetical operations. They may be used as arguments to code objects. They are executed when the code object is executed or when the function needs to be converted to a type that does not accept a function.

Get operation
: Virtual machine operation to request information from some object. Started by the Prolog predicate `get/[3-13]`, when an obtainer is executed or from PCE's built-in functionality.

GUI
: Abbreviation for Graphical User Interface.

Inheritance
: The sharing of definition from a super-class. When a PCE *class* is created from a *super-class* it is initially a copy of this *super-class*. After creation, instance variables and methods may be added and/or redefined.

Instance
: Synonym for *object*, often use to stress the fact that an object belongs to a particular class.

Instance-variable
: Placeholder for the local-state associated with an *object*. An *instance-variable* is associated with a class and has a name and a type. Each of the *instances* of the class defines a value for the instance variable. Instance variables are represented by class `variable`.

Message
: A *message* is an object representing a *send-operation*. The phrase "sending a message to X" is equivalent to "invoking a get- or send-operation on X".

Method
: A *method* maps a *selector* and a type vector onto an implementation which is either a C-function or a *code* object. PCE defines both get- and send-methods. If a *send-operation* is invoked on an object, PCE will find a method associated with the class of the object with a matching *selector*, check the argument types and invoke the implementation of the method.

Object-reference
: An *object-reference* is the identifier for a particular instance. In Prolog *object-references* are represented by `@Integer` or `@Atom`.

Object
: An *object* is en entity in PCE's world that is identified by an *object-reference* and has a local state. An object is an *instance* of a *class*. The *class* defines both the constituents of the local state as well as the operations (*methods*) understood by the object.

Obtainer
: An *obtainer* is a *function* which invokes a *get-operation* when evaluated. The class name is '?'.

Recogniser
: A *recogniser* object parses *events* for a graphical object.

Selector
: A *selector* is the name of a *send-operation* or *get-operation*.

Send Method
: Refinement of `method` that maps a *send-operation* onto its implementation. See also *Method*

Send operation
: Virtual machine operation which invokes of a *send-method* on some object. Started by the Prolog predicate `send/[2-12]`, when an *message* is executed or from PCE's built-in functionality.

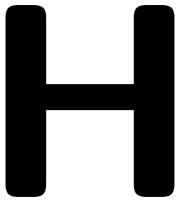
Slot
: Equivalent to *instance_variable*.

Super-class
: The *super-class* of a *class* serves as the initial definition of a *class*. See also *inheritance*.

Template-class
: User-defined subclass of class `template`. The refinements introduced from `template` can be imported in another user-defined class using the predicate `use_class_template/1`.

Var
: A *var* object is a *function*. The commonly used *vars* objects are: `@arg1`, ... (general argument forwarding), `@receiver` (receiver or a message), `@event` (currently processes event object).

Class summary descriptions



This appendix provides a complete overview of all built-in classes of XPCE. For each class, it presents the name, arguments needed to create an instance, place in the inheritance and delegation hierarchies as well as a summary description. For many classes we added a small illustrative example of typical usage of the class.

The summaries stress on describing what the class is commonly used for and what other classes are designed to cooperate with the class.

The classes are presented in alphabetical order. Some classes that are closely related and have symbol-names (`>`, `+`) are combined into one description, sometimes violating the alphabetical order.

object — :=

`:=`**(***name=name, value=any|function***)**

Instances of this class are used to specify named arguments, see section 2.4. Example:

```
...,
send(Editor, style,
     sensitive, style(underline := @on,
                      colour := dark_green)),
...,
```

object — program_object — code — `==` `\==`

`==`**(***left=any|function, right=any|function***)**
`\==`**(***left=any|function, right=any|function***)**

Conditional code object that succeeds if both arguments evaluate to the same object. Normally used to specify the conditions of `if` or `while`. The following example yields the names of all user-defined classes:

```
?- new(UDC, chain),
   send(@classes, for_all,
        if(@arg2?creator \== built_in,
           message(UDC, append, @arg1))).
```

object — program_object — code — function — ?

`?`**(***receiver=object|function, selector=name|function, argument=any|function ...***)**

Class ?, pronounced as ‘obtainer’, represents a ‘dormant’ get-operation. Obtainers are commonly used to ‘obtain’ arguments for other code objects. For example:

```
    ...,
    send(Dialog, append,
         new(TI, text_item(name))),
    send(Dialog, append,
         button(ok, message(Dialog, return, TI?selection))),
```

object — program_object — code — @=

@=(*reference=name, object=object|function***)**

Class @= assigns a symbolic reference name to the argument object. It is used to define global objects in the class-variable display.initialise. See the system defaults file ⟨*pcehome*⟩`/Defaults`. The following example from `Defaults` creates the objects `@_dialog_bg` and `@_win_pen` depending on whether or not the display is monochrome or colour.

```
display.initialise: \
        and(_dialog_bg @= when(@colour_display, \
                               grey80, white), \
            _win_pen   @= when(@colour_display, \
                               0, 1))
```

object < **program_object — code — and**
chain

and(*test=code ...***)**

Code object that executes its arguments one by one. It fails as soon as one of the arguments fails and succeeds otherwise. Commonly used to specify multiple actions for controllers. For example:

```
    ...,
    get(Dialog, frame, Frame),
    send(Dialog, append,
         new(Function, text_item(function))),
    send(Dialog, append,
         button(switch_to,
                and(message(Frame, switch_to,
                            Function?selection),
                    message(Function, clear)))),
    ...,
```

object — visual — application

application(*name***)**

An application object is a `visual` object used to combine multiple `frames`. See section 10.5 for a discussion on its usage.

object < **visual — graphical — joint — arc**
layout_interface

arc(*radius=[int], start=[real], size=[real]***)**

Graphical primitive describing a section from a circle. It may be used to create a pie-chart segment.

```
?- new(A, arc(100, 20, 50)),
   send(A, close, pie_slice).
```

object — area

area(*x=[int], y=[int], width=[int], height=[int]***)**
Combination of *X*, *Y*, *Width* and *Height* used by `graphical` to store the bounding box of the graphical. Also used to communicate with graphical objects and frames about their dimension.

```
...,
get(Box, area, area(X, Y, W, H)),
...,
```

object < **visual — graphical — arrow**
layout_interface

arrow(*length=[int], wing=[int], style=[open,closed], fill=[image|colour]****)**
Arrow-head. Normally only used implicitly to attach arrows to a `line`, `arc` or `path`, the subclasses of class `joint`. See '`joint` → `arrows`'. `arrow` can be used directory to create fancy arrows.

```
?- new(L, line(0, 0, 100, 50, second))
```

object — program_object — code — assign

assign(*variable=var, value=any|function, scope=[local,outer,global]***)**
Assign a value to an instance of class `var`, an XPCE variable. Used to realise variables in compound executable objects.

```
and(assign(new(C, var), @arg1?controller),
    message(C, ...),
    message(C, ...),
    ...)
```

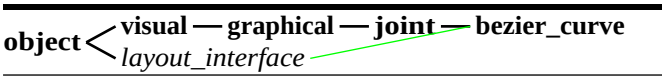
object — program_object — attribute

attribute(*name=any, value=any***)**
Attributes can be associated with any object to store data related to that object without the need to create a subclass. Normally attribute objects are used implicitly through the method '`object` ⇆ `attribute`'.

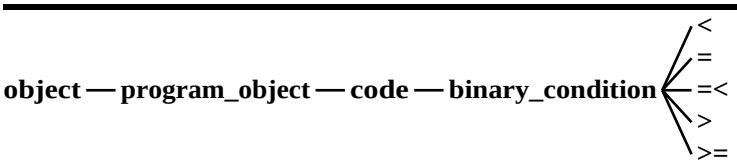
```
send(Frame, attribute, visualises, bicycle24)
```

object — program_object — behaviour

behaviour(*name=name, context=[class|object*]***)**
Super class of `method` and `variable`, representing the two types of objects that can realise behaviour in classes. Not useful for the application programmer.



bezier_curve(*start=point, end=point, control1=point, control2=[point]**)
Create a Bezier curve from *start* to *end* using one or two control-points (quadradic or cubic Bezier curve). Bezier curves are nice smooth curves departing and arriving in a specified direction. See also `path`.

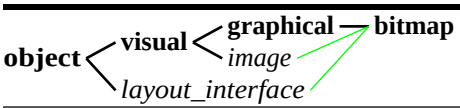


binary_condition(*left=expression, right=expression***)**
<(*left=expression, right=expression***)**
=(*left=expression, right=expression***)**
=<(*left=expression, right=expression***)**
>(*left=expression, right=expression***)**
>=(*left=expression, right=expression***)**
Arithmetic conditional code objects. These objects are normally used to specify the conditions of `if` or `while`. The following example creates a `chain` holding all graphicals on a device that either have ←`width` < 5 or ←`height` < 5.

```
...,
get(Device?graphicals, find_all,
    or(@arg1?width < 5,
       @arg1?height < 5),
    SmallGraphicals),
...,
```

binary_expression(*left=expression, right=expression***)**
(left=expression, right=expression***)**
+(*left=expression, right=expression***)**
-(*left=expression, right=[expression]***)**
/(*left=expression, right=expression***)**
Arithmetic functions, commonly used for computation of graphical dimensions or to specify spatial relations using class `spatial` or for simple functional computation from Prolog. For example:

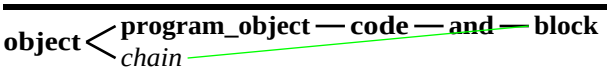
```
...
send(Box, height, Text?height + 10),
...
```



bitmap(*image=[image], transparent=[bool]***)**

A bitmap turns an `image` or `pixmap` into a graphical object that can be displayed on a `device`.

```
?- new(I, image('pce.bm')),
   new(B, bitmap(I)).
```



block(*var|code ...***)**

A `block` is similar to `and`, but provides formal parameters.

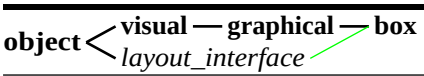
```
?- send(block(new(A, var),
              new(B, var),
              message(@pce, write_ln, A, B)),
        forward, hello, world).

hello world
```

bool()

Class `bool` defines two instances: `@on` and `@off`, representing 'true' and 'false'. The use cannot create instances of this class.

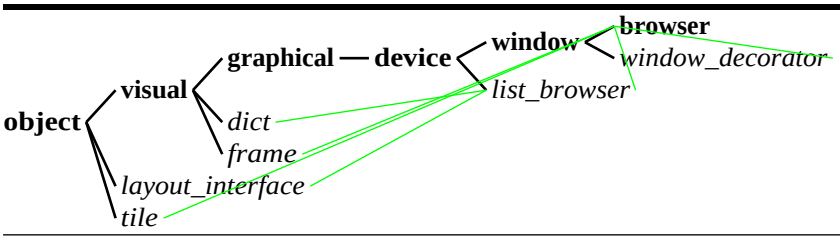
```
...,
send(Image, transparent, @on)
...
```



box(*width=[int], height=[int]***)**

Graphical representing a rectangle. Corners can be rounded, the interior can be filled, the texture and thickness of the line can be controlled and a shadow can be defined.

```
?- new(B, box(100, 50)),
   send(B, radius, 10).
```



browser(*label=[name], size=[size], display=[display]***)**

A `browser` is a window version of a `list_browser`. A browser visualises a list of

dict_item objects. The items are organised in a dict, providing fast access to browser items, even if there are many items in the browser. Individual items may be coloured, underlined, etc. using the style mechanism also available for editor. Columns can be realised using tab_stops on the text_image object that displays the actual text of the browser.

```
?- new(B, browser),
   send_list(B, append, [gnu, gnat]).
```

object — recogniser — gesture — browser_select_gesture

browser_select_gesture()
Internal class dealing with selection handling in class list_browser.

object < **visual — graphical — dialog_item — button**
layout_interface

button(*name=name, message=[code]*, label=[name]***)**
A button is a push-button controller. It has an associated message that is executed if the button is activated using the mouse. Inside a dialog, one button can be assigned as 'default' button.

```
?- new(B, button(hello,
                 message(@pce, write_ln, hello))).
```

object — host — c

c()
Class c is a subclass of class host, providing communication to C and C++ code. It is not used directly by the application programmer.

object — c_pointer

c_pointer(*[c_pointer]***)**
Class c_pointer encapsulates an anonymous C pointer (void *). It is used to register references to Prolog predicates with XPCE methods. See also chapter 7.

```
?- pce_predicate_reference(gnat:gnu(_,_), X).
X = @1190997/c_pointer
```

object — chain

chain(*member=any ...***)**
Class chain represents a single-linked list of arbitrary objects. Chains are commonly used inside XPCE to represent collections. Chains have methods to find elements, sort the chain, delete elements, etc. The predicate chain_list/2 converts between an XPCE chain and a Prolog list. It also provides methods to run code on all elements of the list, which is generally faster than translating the chain to a Prolog list and using Prolog iteration. In the example, 'device ← graphicals' returns a chain holding the graphicals displayed on the device. The example changes the font of all objects of class text to 'bold'.

```
...,
send(Device?graphicals, for_all,
     if(message(@arg1, instance_of, text),
        message(@arg1, font, bold))),
...
```

object — program_object — hyper — chain_hyper

chain_hyper(*from=object, to=object, forward=[name], backward=[name]***)**
Link two objects with a 'chain'. If either dies, the other will die with it. See also the library library(hyper) and section 10.11.

object — hash_table — chain_table

chain_table(*buckets=[int]***)**
Version of a hash_table that allows multiple values to be associated with the same key. The key can be any object. If the value for a key is requested, a chain of values associated with this key is returned.

object — char_array

char_array(*text=char_array***)**
Class char_array is a super-class of the classes string, representing modifiable text and name, representing read-only unique textual constants. Class char_array defines most of the analysis methods for its two subclasses. Almost the only usage of this class for application programmers is as type specifier for methods in user-defined classes that do not modify textual arguments.

```
insert_bold_text(Editor, Text:char_array) :->
        "Insert text with fragment of bold text"::
        get(Editor, caret, Start),
        send(Editor, insert, Text),
        get(Editor, caret, End),
        Len is End-Start,
        new(_, fragment(Editor, Start, Len, bold)).
```

object < **visual — graphical — circle**
layout_interface

circle(*diameter=[int]***)**
Equivalent to an ellipse with the same ←width and ←height. Not used frequently.

object — program_object — class

class(*name=name, super=[class]****)**
All XPCE classes are represented by an instance of class class. A class is a normal object and can thus be manipulated using send/[2-12], get/[3-13] and new/2. Classes are normally only created and modified through the user-defined class layer described in chapter 7. Get methods on classes are used to extract meta-information about its instances, as exploited by the online manual tools.

```
?- get(@pce, convert, box, class, ClassBox),
   get(ClassBox, super_class, X).
X = @graphical_class/class.
```

object — program_object — behaviour — class_variable

class_variable(*class=class, name=name, default=any, type=[type], summary=[string]****)**

A `class_variable` provides can be used to describe class properties as well as to provide access to the XPCE `Defaults` database. Typically, class-variables are defined similar to instance-variables in the XPCE/Prolog class definition:

```
:- pce_begin_class(title_text, text).

class_variable(font, font, huge, "Default font for titles").

...
```

object — recogniser — gesture — click_gesture

click_gesture(*button=[button_name], modifier=[modifier], multiple=[single,double,triple], message=[code]*, preview=[code]*, cancel=[code]**)**

Class `click_gesture` is a `recogniser` that parses button-events to a click. If the click is detected, it will execute the associated message. This class is normally used to make graphical objects sensitive to clicks.

```
...,
send(Bitmap, recogniser,
     click_gesture(left, double, message(Bitmap, open))),
...
```

object — program_object — code

code()

Class `code` is a super-class for all 'executable' objects. An important sub-class is class `function`, representing executable objects that yield a value. The method '`code` → `forward`: any ...' pushes the `var` objects `@arg1`, ... and then executes the code object. Code objects are often associated with controllers to describe the action the controller should perform. They also serve the role of lambda functions. See also section 10.2.

```
?- send(message(@prolog, format, 'Hello ~w.', @arg1),
        forward, world).
Hello world.
```

object — vector — code_vector

code_vector()

A `code_vector` is a subclass of class `vector` that can represent functions as well as normal objects. It is used for packing multiple arguments passed to a variable-argument method. Do not use this class directly. See section 7.5.2.

object — colour

colour(*name=[name], red=[0..65535], green=[0..65535], blue=[0..65535], model=[rgb,hsv]***)**

A `colour` represents an 'RGB' triple.[1] Colours are used as attributes to graphicals, windows, `styles` and `pixmaps`

object — colour_map

colour_map(*name=[name]*, colours=[vector]***)**

Manipulate the colourmap. Colourmaps are normally left untouched, but using a 256 entries colour palette in MS-Windows they can be used to improve full-colour image rendering. See also section 10.10.1.

object — recogniser — gesture — connect_gesture

connect_gesture(*button=[button_name], modifier=[modifier], link=[link]***)**

A `connect_gesture` allows the user to connect two graphicals by dragging from the first to the second. This requires two graphicals with `handless` attached, a `link` that is compatible with the handles and a `connect_gesture` associated width the graphical at which the connection should start. The demo program PceDraw as well as the XPCE Dialog Editor described in chapter A exploit connections and connect_gestures.

object < **visual — graphical — joint — line — connection**
layout_interface

connection(*from=graphical, to=graphical, link=[link], handle_from=[name]*, handle_to=[name]***)**

A `connection` is a line between two graphical objects that automatically updates on geometry, device and displayed-status changes to either of the connected graphicals. Both of the graphicals must have one or more `handles` associated with them. The connection can be attached to a specific handle, or to any handle of the proper '`handle` ← `kind`'. In the latter case, the system will automatically choose the 'best-looking' handle.

object — constant

constant(*name=name, summary=string***)**

A `constant` is a unique handle. XPCE predefines the following constants: `@nil`, `@default`, and from the subclass `bool`, `@on` and `@off`. The use can define additional constants and give them their own unique meaning. The most obvious usage is to indicate a slot that can hold arbitrary data including `@nil` and `@default` is in a special state.

```
?- new(@uninitialised,
       constant(uninitialised,
                'Not yet initialised slot')).
```

object — constraint

constraint(*from=object, to=object, relation=relation, propagate=[forwards,backwards]***)**

A constraint is a relation between 2 objects that has to be maintained if either of the

[1] Colour screens create their colour by mixing the 'primary' colours 'red', 'green' and 'blue'. With an 'RBG' triple, we refer to a triple of three numeric values representing the intensities of the three primary colours

objects is changed. The 'constraint ← relation' is a description of the relation maintained by the constrained. The system defines the relations identity (both objects have an attribute that has same value) and spatial (general purpose geometry-relation between two (graphical) objects. It is possible to define new relation classes. Constraints are getting out of fashion as XPCE lacks a good mechanism to detect when an object has been changed and therefore evaluates the relation far too often. User-defined classes, possibly combined with hyper objects form an attractive alternative. The following keeps a text centered in a box.

```
...,
new(_, constraint(Box, Text, identity(center))),
...
```

object — program_object — code — function — create

create(*class=name|class, argument=any|function ...***)**

Function that creates an instance of a class. It is often required if a code fragment executed by an 'iterator' method such as 'chain → for_all' has to create objects. The following code generates dict_items from all send_methods of the specified class and displays them to a browser.

```
send_methods_of_class(ClassName) :-
        new(B, browser(ClassName)),
        get(@pce, convert, ClassName, class, Class),
        get(Class, send_methods, SendMethods),
        send(SendMethods, for_all,
             message(B, append,
                     create(dict_item,
                            @arg1?name,
                            @default,
                            @arg1))),
        send(B, open).
```

object — cursor

cursor(*name=name*, image=[image], mask=[image], hot_spot=[point], foreground=[colour], background=[colour]***)**

A cursor defines the shape that indicates the position of the pointer (also called mouse). The system provides a large set of predefined cursors from X11. The Win32 version adds the standard Windows cursors to this set. Cursors can also be created from an image. The demo program Cursors displays all defined cursors.

Cursors can be associated with graphicals and windows using the →cursor method. They are also associated to gestures, where they define the cursor that is visible while the gesture is active (i.e. while the mouse-button that activated the gesture is down).

Type-conversion converts names into cursor objects. Explicit creation of cursors is rarely used.

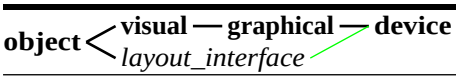
```
...,
send(Box, cursor, gobbler),
...
```

object — date

date(*seconds=[0..59], minutes=[0..59], hours=[0..23], day=[1..31], month=[1..12], year=[1970..2050]***)**

A `date` objects represents a point in time. The underlying representation is the POSIX file time-stamp: seconds elapsed since 00.00, Jan 1-st, 1970. This limits the applicability of this class to time-stamps of computer resources (files), agenda systems and other domains that do not require a granularity below 1 second or have to represent time-stamps in far history or future. Class `date` can parse various textual representations into date objects.

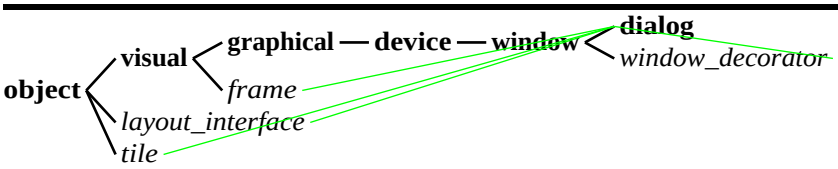
```
?- send(@pce, format, 'It is now "%s"\n',
        new(date)?string).
It is now "Tue Jan 30 14:07:05 1996"
```



device()

A graphical `device` is a compound graphical. It is the super-class of class `window`. It is a sub-class of `graphical`, which implies devices can be used to create a consist-of structure of graphical objects, giving structure to a diagram. Devices are commonly refined to establish user-defined graphics, see section 10.12. See also class `figure`.

```
make_icon(Icon, Image, Label) :-
        new(Icon, device),
        send(Icon, display,
             new(BM, bitmap(Image))),
        send(Icon, display,
             new(T, text(Label, center))),
        send(T, y, BM?bottom_side),
        send(T, center_x, BM?center_x).
```



dialog(*label=[name], size=[size], display=[display]***)**

A `dialog` is a `window` specialised for the layout and message handling required by `dialog_items`, the super-class of the XPCE controllers. In most cases, controller-windows are created by simply →`append`ing a number of controllers to a dialog window. The `frame`- and `dialog`-layout services take care of proper window sizes and layout of the controllers. Dialog windows are also involved in forwarding →`report` messages (see section 10.7) and keyboard accelerators, handling the default button.

```
:- pce_autoload(file_item, library(file_item)).

edit_file_dialog :-
        new(D, dialog('Edit File')),
        send(D, append,
             file_item(edit_file, '')),
        send(D, append,
             button(edit, message(@prolog, emacs, @arg1))),
        send(D, append,
             button(cancel, message(D, destroy))),
        send(D, open).
```

---

object < **visual — graphical — device — dialog_group**
layout_interface

dialog_group(*name=[name], kind=[box,group]***)**

A `dialog_group` is a collection of `dialog_items`. Dialog groups may be used to realise a (labeled) box around a group of controllers, or to combine multiple controllers into a compound one for technical or layout reasons. See also `tab`.

---

object < **visual — graphical — dialog_item**
layout_interface

dialog_item(*name=name***)**

Class `dialog_item` is a super-class of all XPCE controllers. It contains the code necessary to negotiate geometry with its neighbours and enclosing `dialog` window and provides default fonts for the label, etc. Class `graphical` defines similar methods to allow integration of raw graphical objects into dialog windows easily, but `graphical` uses the more expensive object-level attributes for storing the necessary status. Open the class-hierarchy below class `dialog_item` to find all available controllers.

---

object — visual — dict

dict(*member=dict_item ...***)**

A `dict` is an abbreviation of dictionary. Dicts map keywords to `dict_item` objects. Small dicts simply use a linear list (`chain`) of items. Large dicts will automatically built a `hash_table` for quick lookup on the first request that profits from the availability of a table. A dict provides the storage for a `list_browser`. See also class `browser`.

---

object — visual — dict_item

dict_item(*key=any, label=[char_array], object=[any]*, style=[name]***)**

Item in a `dict`. The *key* is used for lookup. *label* is the text displayed by the `browser` (`@default` uses the *key*). *Object* is an arbitrary object that can be associated to the dict. If a dict presents a set of XPCE objects, it is common practice to extract the key and or label from the object and store the object itself in the '`dict_item` ⇆ `object`' slot.

A `name` is translated to a dict_item using the name as *key*, default label and `@nil` object. '`dict_item` ⇆ `style`' can be used to give an item special attributes (colour, font, etc.).

object — directory

directory(*path=name***)**

A `directory` represents an node (folder) in the computer's file-system. Directories are most commonly used to enumerate the files and sub directories. Directory objects can also be used to create or delete directories from the file-system.

```
?- get(directory(.), files, Files).
```

object — visual — display

display(*address=[name]***)**

A `display` represents what X11 calls a screen, a desktop on which windows can be displayed with a mouse and keyboard attached to it. XPCE support multiple display instances under X11 and only the predefined default display `@display` under Win32. The display implements a number of global operations: getting the screen ←`size`, showing modal message boxes using →`inform` and →`confirm`, etc.

```
?- get(@display. size, size(W, H)).
W = 1024, H = 786
```

object — visual — display_manager

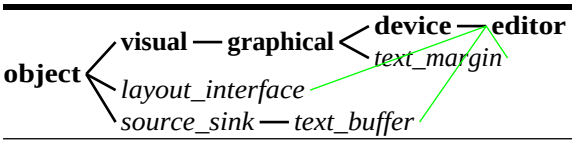
display_manager()

The object `@display_manager` is the only instance of this class. It represents the collection of available `display` objects and provides access to the system-wide event-dispatching services. It is the root of the consist-of hierarchy of `visual` objects as displayed by the *Visual Hierarchy* tool.

object — recogniser — gesture — edit_text_gesture

edit_text_gesture(*button=[button_name], modifier=[modifier]***)**

Used internally to handle selection inside a `text` object. See also the library library(`pce_editable_text`).



editor(*text=[text_buffer], width=[int], height=[int], margin=[int]***)**

An `editor` is a general-purpose text editor. It is a graphical. Class `view` provides a window-based version of the editor. XPCE's editors have commands and key-bindings that are based on GNU-Emacs. Editors are fully programmable. The associated `key_binding` object parses key-strokes into commands that are defined as methods on the editor.

An editor is a compound object and a subclass of `device`. The other components are a `text_image` to form the actual display, a `text_buffer` to provide the storage for the text, elementary operations on the text and undo, a `text_cursor` to indicate the location of the caret, and optionally a `text_margin` to visualise the presence of annotations.

A single `text_buffer` can be associated with multiple `editor` objects, providing shared editing.

Editors can handle sensitive regions, different fonts, colours and attributes using `fragment` objects. All text windows in XPCE's demo programs (PceEmacs, cards from the online help, application help, etc.) either use class view or class editor to display the text.

---

object — elevation

elevation(*name=[name|int]*, height=[int], colour=[hilited,reduced|colour|pixmap], relief=[colour|pixmap], shadow=[colour|pixmap], kind=[3d,shadow], background=[reduced|colour|pixmap]***)**

An `elevation` object describes an elevated region on the screen. Elevations come in two flavours: as a shadow for monochrome displays and using light and dark edges on colour displays. The elevation object itself just contains the colour definitions. The actual painting is left to the graphical object the elevation is attached to.

Most controllers handle elevations. The only general purpose graphical supporting an elevation is `figure`.

---

object < **visual — graphical — ellipse**
layout_interface

ellipse(*width=[int], height=[int]***)**

Elliptical shape. Class `ellipse` defines similar attributes as `box`: pen, texture, fill_pattern and shadow. See also `circle`.

---

object — error

error(*name=name, format=string, kind=[status,inform,warning,error,fatal,ignored], feedback=[report,throw,print]***)**

An `error` object represents a runtime message. Whenever an error is trapped or a message needs to be displayed, the system will invoke '`object → error`: id, context ...' to the object that trapped the error. If this method is not redefined, the system will report the error using the '`object → report`' mechanism described in section 10.7. Errors can be prevented from being reported using `pce_catch_error/2`. The *Error Browser* of the online manual shows all defined errors.

The development system will report errors that are considered 'programming errors' (undefined methods, type violations, invalid object references, etc.) to the terminal and start the tracer. See also section 12.

---

object — event

event(*id=event_id, origin=[window|frame], x=[int], y=[int], button_mask=[int], time=[int]***)**

An `event` represents an action from the application user: pressing a key, moving the mouse, pressing a mouse-button, or entering or leaving an area with the mouse. The main loop of XPCE will read window-system events from the computing environment (X11 or Win32). If the event concerns a repaint or similar system event, it will be handled appropriately. If it can be expressed as an XPCE event, an `event` object will be created and send to the window for which the event was reported by the system using the method '`event → post`'.

Graphical objects and windows can redefine their event handling using two mechanisms: by redefining the →event method or by associating a recogniser object using 'graphical → recogniser'.

Normally, XPCE will read and dispatch events when 'there is nothing else to do'. For processing events during computation, see 'graphical →synchronise' and 'display → dispatch'.

---

object — event_node

event_node(*value=name, parent=[event_node]****)**

An event_node is a node in the event 'is_a' hierarchy. See the demo program Events. Event-types are normally tested using 'event → is_a'.

```
event(Dev, Ev:event) :->
        "Forward all keyboard events to the text"::
        (   send(Ev, is_a, keyboard)
        ->  get(Dev, member, text, Text),
            send(Ev, post, Text)
        ;   send(Dev, send_super, event, Ev)
        ).
```

---

object — event_tree

event_tree(*root=[event_node]***)**

Event 'is_a' hierarchy. The only instance is @event_tree.

---

object < **visual — graphical — device — figure**
layout_interface

figure()

A figure is a refinement of a device. It is a compound graphical, but in addition can define a background, surrounding box with margin, possibly rounded corners and elevation and a clipping region. Finally, figures may be used not only to display all member graphicals, but also to show 'one of' the member graphicals only. See 'figure → status'. An example of the usage of figures are the 'object cards' of the *Inspector* tool.

---

object — source_sink — file

file(*path=[name], kind=[text,binary]***)**

An XPCE file object represents a file on the computers file-system. It can be used to specify a file, query a file for various attributes, read a file, etc. See also directory.

```
?- get(file('summary.doc', size, Size).
Size = 30762
```

---

object — font

font(*family=name, style=name, points=[int], x_name=[name]***)**

A font is a reusable object that describes the typeface of text. Section 10.9 documents the specification of physical and logical fonts.

```
...,
send(Text, font, bold),
...
```

object — format

format(*orientation=[horizontal,vertical], width=[1..], columns=[bool]***)**

A `format` describes the layout of graphicals on a `device`. It can specify 'tabular' and 'paragraph' style layout. A format itself just specifies the parameters, 'device →`format`' actually realises the format.

object — visual — fragment

fragment(*text=text_buffer, start=int, length=int, style=[name]***)**

A `fragment` defines a region of text in a `text_buffer` using a start-position and a length. Fragments are automatically updated if the contents of the text_buffer changes. A fragment can be assigned a logical 'category', called 'style'. The `editor` visualising the text_buffer maps the style-names of fragments into `style` objects using '`editor → style`'.

```
...,
send(Editor, style, title, style(font := huge)),
new(_, fragment(Editor, Start, Len, title)),
...
```

object — visual — frame

frame(*label=[name], kind=[toplevel,transient,popup], display=[display], application=[application]***)**

A `frame` is a collection of `tiled` windows. Frames handle the layout, resizing, etc. of its member windows. Any XPCE window is enclosed in a frame, though it is often not necessary to specify a frame explicitly. Applications are often implemented as subclasses of `frame`. Section 10.6 describes the layout of windows inside a frame.

```
...,
new(F, frame('My application')),
send(F, append, new(B, browser)),
send(new(P, picture), right, B),
...
send(F, open).
```

object — program_object — code — function

function()

A `function` is a `code` object that yields a value when executed. See section 10.2.2.

object — recogniser — gesture

gesture(*button=[button_name], modifier=[modifier]***)**

Class `gesture` is the super-class for the `recogniser` classes that deal with the sequence mouse-button-down ...dragging ...mouse-button-up. This super-class

validates the various conditions, handles the cursor and focus and activates the →initiate, →drag and →terminate methods that are redefined in its subclasses. This class is often sub-classed.

object — program_object — behaviour — method — get_method

get_method(*name=name, return=[type], types=[vector], implementation=function|c_pointer, summary=[string]*, source=[source_location]*, group=[name]**)**

Specification of get-behaviour that is associated with a class using 'class → get_method' or with an individual object using 'object →get_method'. Normally specified through the preprocessor layer defined in chapter 7.

object < **visual — graphical**, *layout_interface*

graphical(*x=[int], y=[int], width=[int], height=[int]***)**

The most generic graphical object. This class defines generic geometry management, display, update, event-handling, etc. This class can be sub-classed to defined specialised graphics. See section 10.12.

object < **hbox — grbox**, *visual — graphical*, *layout_interface*

grbox(*graphical=graphical, alignment=[top,center,bottom,left,right], rubber=[rubber]**)**

Embed a graphical in a parbox. Using left or right alignment, grbox can also be used to have text floating around graphical illustrations. See section 11.10.

object — handle

handle(*x=expression, y=expression, kind=[name], name=[name]***)**

A handle defines a typed and named position on a graphical used by connections to connect to. The positions are formulas expressed in the with and height of the graphical. The following definitions are encountered regularly:

```
:- pce_global(@north_handle,
              new(handle(w/2, 0, link, north))).
:- pce_global(@south_handle,
              new(handle(w/2, h, link, south))).
:- pce_global(@east_handle,
              new(handle(0, h/2, link, east))).
:- pce_global(@west_handle,
              new(handle(w, h/2, link, west))).
```

object — recogniser — handler

handler(*event=event_id, message=code*, restrict_to=[region]***)**

A handler is the most primitive recogniser, mapping an event-type to a message. Since the introduction of the more specialised gesture and key_binding as well as the possibility to refine the 'graphical → event' method, it is now rarely used.

```
...,
```

```
send(Graphical, recogniser,
     handler(area_enter,
             message(Graphical, report,
                     'Hi, I''m %s',
                     Graphical?name))),
...
```

object — recogniser — handler_group

handler_group(*member=recogniser ...***)**

A `handler_group` is a compound `recogniser` object. When asked to handle an event, it will try each of its members until one accepts the event, after which it will return success to its caller. The following defines a combined move- an resize-gesture. Note the order: resize gestures only activate close by the edges of the graphical, while move gestures do not have such a limitation.

```
:- pce_global(@move_resize_gesture,
      new(handler_group(new(resize_gesture),
                        new(move_gesture)))).
```

object — hash_table

hash_table(*buckets=[int]***)**

A `hash_table` is a fast association table between pairs of objects. For example, `@classes` is a hash_table mapping class-names into class objects. Names are often used as *keys*, but the implementation poses no limit on the type of the key.

```
?- new(@ht, hash_table),
   send(@ht, append, gnu, image('gnu.img')).

?- get(@ht, member, gnu, Image).
```

object — hbox

hbox(*width=[int], ascent=[int], descent=[int], rubber=[rubber]****)**

Superclass of `tbox` and `grbox` dealing with document-rendering. Instances of `hbox` itself can be used to define 'rubber'. See section 11.10 for details.

object — host

host(*name=name***)**

Class `host` represents the host-language, Prolog for this manual. It predefines a single instance called `@prolog`. Sending messages to `@prolog` calls predicates. See also section 6.

```
?- send(@prolog, write, hello).
hello
```

object — host_data

host_data(*alien:void ****)**

Support class for passing data of the host-language natively around in XPCE. The Prolog interface defines the subclass `prolog_term` and the interface-type `prolog`. Details are discussed in the interface definition in section 6.2.

object — program_object — hyper

hyper(*from=object, to=object, forward=[name], backward=[name]***)**

A `hyper` is a binary relation between two objects. The relation can be created, destroyed and inspected. It is automatically destroyed if either of the two connected objects is destroyed. The destruction can be trapped. Messages may be forwarded easily to all related objects. See also section 10.11.

object — relation — identity

identity(*selector1=name, selector2=[name]***)**

An `identity` is a `relation` that maintains the identify between an attribute on one object and an attribute on another object. Given a slider and a box, the following ensures the selection of the slider is the same as the width of the box, regardless of which of the two is changed. See also `constraint`.

```
new(_, constraint(Slider, Box,
                  identity(selection, width)))
```

object — program_object — code — if

if(*condition=code, then=[code]*, else=[code]****)**

Code object implementing a branch. All three arguments are statements. Both 'then' and 'else' are optional, and when omitted, simply succeed. Class `if` is most commonly used in combination with the iteration methods such as '`chain` → `for_all`':

```
...,
send(Device?graphicals, for_all,
     if(message(@arg1, instance_of, device),
        ...)),
...
```

object — visual — image

image(*name=[source_sink]*, width=[int], height=[int], kind=[bitmap,pixmap]***)**

An `image` is a two-dimensional array of pixels. Images come in two flavours: monochrome, where each pixel represents a boolean and colour, where each pixel represents a colour. XPCE can save and load both monochrome and colour images. Images are displayed on a graphical device using a `bitmap`. They are also used to specify `cursor` objects and the icon associated with a 'frame'. See section 10.10.

object < **visual — graphical — dialog_item — text_item — int_item**
layout_interface

int_item(*name=[name], default=[function|int], message=[code]*, low=[int], high=[int]***)**

Subclass of `text_item` for entering integer values. Has stepper buttons for incrementing and decrementing the value.

object < **visual — graphical — joint**
layout_interface

joint(*x=[int], y=[int], width=[int], height=[int], arrows=[none,first,second,both]***)**
Class `joint` is a super-class of the various line-types with a start- end end-point. It provides the code dealing with attached `arrow`-*heads* at either end. As well as common code to reason about the start and end. See also `line`, `path`, `arc` and `connection`.

object — recogniser — key_binding

key_binding(*name=[name]*, super=key_binding ...***)**
A `key_binding` object parses events into messages or methods on the object for which it is handling events. Key-bindings are used by the classes `text`, `text_item`, `editor` and `list_browser`. They can be used to defined keyboard-accelerators, though '`menu_item` ⇆ `accelerator`' is generally more suitable for this purpose.

object < **visual — graphical — dialog_item — label**
layout_interface

label(*name=[name], selection=[string|image], font=[font]***)**
A `label` is a controller used to display read-only text or `image`. Labels can handle →`report` messages. See section 10.7. The code below is the typical way to associate a label that will catch report messages for all windows of the `frame` in which the `dialog` is enclosed.

```
...,
send(Dialog, append, label(reporter)),
...
```

object < **visual — graphical — device — dialog_group — label_box**
layout_interface

label_box(*name=[name], message=[code]**)**
Subclass of `dialog_group` for the definition of compound controllers with a properly aligned label at their left-hand side.

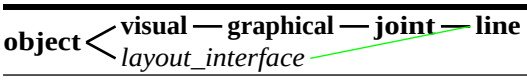
object < **layout_manager**
layout_interface

layout_manager()
layout_interface(*graphical***)**

A `layout_manager` may be attached to a graphical `device` (including a `window`) to manage the layout of graphicals displayed on the device, as well as painting the background of the device. See `table` for a typical example.

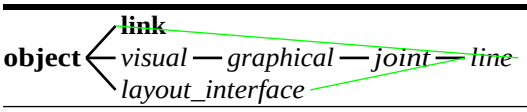
object < **visual — graphical — device — lbox**
layout_interface

lbox(*width=[0..]***)**
Class of the document-rendering system to render a list environment, a sequence of labels and text. See section 11.10 for details.



line(*start_x=[int], start_y=[int], end_x=[int], end_y=[int], arrows=[none,first,second,both]***)**

A `line` is a straight line-segment with optional arrows, thickness and texture. Class `path` implements a 'multi-line'.

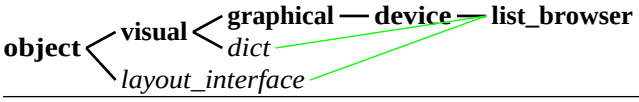


link(*handle_kind1=[name], handle_kind2=[name], line=[line], connection_class=[class]***)**

A `link` is a reusable specification for a `connection`. Links are used for defining connections and `connect_gesture` objects. A connection knows about the link used to instantiate it. The example defines the `handles`, `link` and `connect_gesture` and shapes that allows the user to create links with an error from 'out' ports to 'in' ports.

```
:- pce_global(@in_handle,
              new(handle(0, h/2, in, in))).
:- pce_global(@out_handle,
              new(handle(w, h/2, out, out))).
:- pce_global(@inout_link,
              new(link(out, in,
                       line(arrows := second)))).
:- pce_global(@link_in_out_gesture,
              new(connect_gesture(left, '',
                                  @inout_link))).

make_shape(S) :-
        new(S, box(50,50)),
        send_list(S, handle,
                  [@in_handle, @out_handle]),
        send(S, recogniser, @link_in_out_gesture).
```



list_browser(*dict=[dict], width=[int], height=[int]***)**

A `list_browser` is a graphical version of a `browser`, the visualisation of a list of items (`dict_item`) stored in a `dict`. The graphical version is sometimes displayed with other controllers on a `dialog` window. The example created a `list_browser` holding all current Prolog source files. Double-clicking a file will start *PceEmacs* on the file. Selecting a file and pressing Consult will (re)consult the file.

```
show_source_files :-
        new(D, dialog('Prolog Source Files')),
        send(D, append, new(B, list_browser)),
        forall(source_file(X), send(B, append, X)),
        send(B, open_message,
```

```
            message(@prolog, emacs, @arg1?key)),
        send(D, append,
            button(consult,
                message(@prolog, consult,
                        B?selection?key))),
        send(D, open).
```

---

object < **visual — graphical — dialog_item — menu**
layout_interface

---

menu(*name=[name], kind=[name], message=[code]****)**

Class `menu` realises various different styles of menus and is the super-class for `popup`. Basically, a menu presents multiple values, allows the user to choose one or more items ('menu →`multiple_selection`') and defines a 'look'. The '`menu → kind`' set the various attributes to often-used combinations. The other 'look-and-feel' attributes may be used to fine-tune the result afterwards.

Menu-items can have a textual or `image` label. Labels can be coloured and specify a different `font`.

```
...,
new(M, menu(gender, choice)),
send_list(M, append, [male, female]),
send(M, layout, horizontal),
...,
```

---

object < **visual — graphical — dialog_item — menu_bar**
layout_interface

---

menu_bar(*name=[name]***)**

A menu-bar is a row of pull-down menus. Many applications define a single menu-bar at the top of the frame presenting the various commands in the application.

```
:- pce_begin_class(my_application, frame).

initialise(F) :->
        send(F, send_super, initialise,
             'My Application'),
        send(F, append, new(MBD, dialog)),
        new(V, view),
        send(new(B, browser, left, V)),
        send(B, below, MBD),
        send(MBD, append, new(MB, menu_bar)),
        send(MB, append, new(F, popup(file))),
        send(MB, append, new(E, popup(edit))),
        send_list(F, append,
                  [ menu_item(load,
                              message(F, load)),
                    ...
```

object — visual — menu_item

menu_item(*value=any, message=[code]*, label=[name|image], end_group=[bool], condition=[code]*, accelerator=[name]***)**

Item of a `menu` or `popup`. For `popup` menus, the items are normally created explicitly as each item often defines a unique command. For `menus`, it is common practice to simply append the alternatives as `menu_item` will translate a `name` into a menu_item with this ←`value`, ←`message` `@default` and a ←`label` created by 'capitalising' the value.

object — program_object — code — message

message(*receiver=object|function, selector=name|function, argument=any|function ...***)**

A `message` is a dormant 'send-operation'. When executed using →`execute` or →`forward`, a message is sent to the *receiver*. Message are the most popular `code` objects. See section 10.2 and many examples in this chapter.

object — program_object — behaviour — method

method(*name=name, types=[vector], implementation=code|c_pointer, summary=[string]*, source=[source_location]*, group=[name]**)

Class `method` is the super-class of `send_method` and `get_method`. Instances of this class itself are useless.

object — modifier

modifier(*shift=[up,down], control=[up,down], meta=[up,down]***)**

A `modifier` is a reusable object that defines a condition on the status of the three 'modifier keys' shift, control and meta/alt. Modifiers are used by class `gesture` and its sub-classes. They are normally specified through their conversion method, which translates a `name` consisting of the letters `s`, `c` and `m` into a modifier that requires the shift, control and/or meta-key to be down an the other modifier keys to be up. The example specifies a 'shift-click' gesture.

```
...,
click_gesture(left, 's', single,
              message(...)),
...
```

object — recogniser — gesture — move_gesture

move_gesture(*button=[button_name], modifier=[modifier]***)**

If a `move_gesture` is attached to a `graphical`, the graphical can be moved by dragging it using the specified mouse-button. See also `move_outline_gesture`.

```
...,
send(Box, gesture, new(move_gesture)),
...
```

object — recogniser — gesture — move_gesture — move_outline_gesture

move_outline_gesture(*button=[button_name], modifier=[modifier]***)**

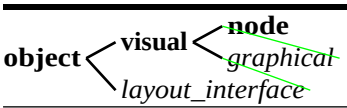
Similar to a `move_gesture`, but while the gesture is active, it is not the graphical itself

that is moved, but a dotted box indicating the outline of the graphical. If the button is released, the graphical is moved to the location of the outline. Should be used for complicated objects with many constraints or connections as a direct move_gesture would be too slow.

object — char_array — name

name(*text=char_array***)**

A `name` is a unique textual constant, similar to an atom in Prolog. Whenever an atom is handed to XPCE, the interface will automatically create a `name` for it. There is no limit to the number of characters that can be stored in a name, but some Prolog implementations may limit the number of characters in an atom. On these platforms, it is implementation-dependent what will happen to long names that are handed to the Prolog interface.



node(*image=graphical***)**

A *node* is a node in a `tree` of `graphical` objects.

```
...,
new(T, tree(new(Root, node(text(shapes))))),
send(Root, son, node(circle(50))),
send(Root, son, node(box(50, 50))),
...
```

object — program_object — code — not

not(*test=code***)**

Code object that inverses the success/failure of its argument statement. Often used for code objects that represent conditions.

```
primitives(Device, Primitives) :-
        get(Device?graphicals, find_all,
            not(message(@arg1, instance_of, device)),
            Primitives).
```

object — number

number(*value=int|number***)**

A `number` is the object version of an integer (int). If may be as a storage bin. To compute the widest graphical of a device:

```
widest_graphical(Device, Width) :-
        new(N, number(0)),
        send(Device, for_all,
             message(N, maximum, @arg1?width)),
        get(N, value, Width),
        send(N, done).
```

object

object(*unchecked ...***)**
Class `object` is the root of XPCE's class-inheritance hierarchy. It defines methods for general object-management, comparison, hypers, attributes, etc. It is possible to create instances of class `object`, but generally not very useful.

object — operator

operator(*name=name, priority=0..1200, kind=xf,yf,xfx,xfy,yfx,yfy,fy,fx***)**
Part of XPCE's object parser. Not (yet) available to the application programmer.

object < **program_object — code — or**
chain

or(*test=code ...***)**
Disjunctive `code` object. An or starts executing its argument statements left-to-right and terminates successfully as soon as one succeeds. The empty `or` fails immediately.

object < **visual — graphical — device — parbox**
layout_interface

parbox(*width=[int], alignment=[left,center,right,justify]***)**
Class to render text with mixed fonts and colours together with graphics. Class `parbox` is the heart of the document-rendering primitives described in section 11.10.

object < **parser**
tokeniser

parser(*tokeniser=tokeniser, operators=operator...***)**
Part of XPCE's object parser. Not (yet) available to the application programmer.

object < **visual — graphical — joint — path**
layout_interface

path(*kind=[poly,smooth], radius_or_interval=[int], points=[chain]***)**
A `path` is a multi-segment line. It comes in two flavours: `poly` as a number of straight connected line-segments and `smooth` as an interpolated line through a number of 'control-points'. Its line attributes can be defined and the interior can be filled. Paths are used both to define new graphicals, for example a triangle, or to defines curves.

```
draw_sine :-
        send(new(Pict, picture), open),
        send(Pict, display, new(P, path)),
        (   between(0, 360, X),
            Y is sin((X * 6.283185)/360) * 100,
            send(P, append, point(X, Y)),
        fail
        ;   true
        ).
```

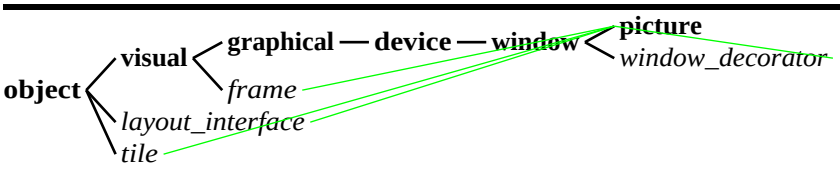
object — pce

pce()
Class `pce` defines a single instance called `@pce`. Actions that cannot sensibly be related to a particular object are often defined on class `pce`.

```
?- get(@pce, user, User).
User = jan
```

object — pen

pen(*thickness=[0..], [texture=texture_name], colour=[colour|pixmap]***)**
Reserved for future usage.



picture(*label=[name], size=[size], display=[display]***)**
A `picture` is a `window` with scrollbars, normally used for application graphics. If a graphical window without scrollbars is required, `window` should be considered.

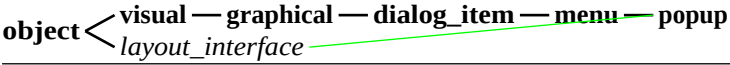
object — visual — image — pixmap

pixmap(*source=[image|file]*, foreground=[colour], background=[colour], width=[int], height=[int]***)**
A `pixmap` is a subclass of `image` that is reserved for colour images. All functionality of this class is in class `image`. The main reason for its existence is that some graphical operations *require* a colour image and the introduction of a class for it is the only way to allow this to be specified using XPCE's type system. The →`initialise` method is specialised for handling colour images.

object — point

point(*x=[int], y=[int]***)**
Position in a two-dimensional plane. Together with `size` and `area` used to communicate with graphicals about geometry.

```
...
get(Box, center, Point),
get(Point, mirror, Mirrored),
send(Box, center, Mirrored),
...
```



popup(*name=[name], message=[code]**)**
A popup menu is a menu that is shown after pressing a button on the object the menu is attached to. Popups are used in two different contexts, as pulldown menus attached to a `menu_bar` and as popup-menus associated with windows or individual graphical objects.

Popups are →`appended` to menu_bars. Various classes define the method →`popup` to associate popup menus. Finally, class `popup_gesture` provides a gesture that operates popup menus.

A popup consists of menu_items, each of which normally defines a message to be executed if the corresponding item is activated. Pull-right sub-menus are realised by appending a popup to a popup.

```
...,
new(P, popup(options)),
send(P, append,
     new(L, popup(layout, message(Tree, layout, @arg1)))),
send_list(L, append, [horizontal, vertical, list]),
send(P, append,
     menu_item(quit, message(Tree, destroy))),
...
```

object — recogniser — gesture — popup_gesture

popup_gesture(*popup=[popup|function], button=[button_name], modifier=[modifier]***)**

A popup_gesture parses events and activates a popup menu. Popup gestures are explicitly addressed by the application programmer to define compound gestures involving a popup:

```
:- pce_global(@graph_node_gesture,
              make_graph_node_gesture).

make_graph_node_gesture(G) :-
        new(P, popup),
        send_list(P, append, [...]),
        new(G, handler_group(connect_gesture(...),
                             move_gesture(middle),
                             popup_gesture(P))).
```

object — stream — process

process(*command=char_array, argument=char_array ...***)**

A process encapsulates a stream- or terminal program to get its input from a graphical program and redirect its output to the same graphical program. Various of the XPCE tools and demo programs exploit processes: The M-x shell, M-x grep and other shell commands of *PceEmacs*, the ispell program and the chess front-end. See also socket.

object — program_object — code — function — progn

progn(*statement=code|any ...***)**

Code object with semantics like the LISP progn function. A progn executes its statements. If all statements are successfully executed and the last argument is a function, execute the function and return the result of it or, if the last argument is not a function, simply return it. Used infrequently in the XPCE/Prolog context.

object — program_object

program_object()

The super-class of almost the entire ‘meta-word’ of XPCE: classes, behaviour,

attributes, types, etc. Class `program_object` defines the XPCE tracer. See `tracepce/1` and `breakpce/1`.

---

object < ~~**quote_function**~~ / *program_object — code — function*

quote_function(*function=function***)**

Most of XPCE is defined to evaluate function objects at the appropriate time without the user having to worry about this. Sometimes however, type-checking or execution of a statement will enforce the execution of a function where this is not desired. In this case class `quote_function` can help. As a direct sub-class of `object`, it will generally be passed unchanged, but type-conversion will translate extract the function itself if appropriate, while delegation allows the quote_function to be treated as a function.

In the example, *ChainOfChains* is a chain holding chains as its elements. The task is to sort each of the member chains, using the `function` `?(@arg1, compare, @arg2)` for sorting. If not enclosed in a `quote_function`, the message will try to evaluate the function. Now it passes the quote_function unchanged. The '`chain` → `sort`' method requires a `code` argument and therefore the function will be extracted from the quote_function.

```
...,
send(ChainOfChains, for_all,
     message(@arg1, sort,
         quote_function(?(@arg1, compare, @arg2)))),
...
```

---

object — real

real(*value=any***)**

A `real` is XPCE's notion of a floating-point number. Reals are represented using a C single-precision 'float'. Reals define the same operation as class `number`, its integer equivalent.

---

object — recogniser

recogniser()

Class `recogniser` is the super-class of all event-parsing classes. The sub-tree `gesture` handles mouse-button related events, `key_binding` handles typing and `handler` may be used for all events. The main purpose of this class itself is to provide a type for all its sub-classes.

---

object — regex

regex(*pattern=[char_array], case_sensitive=[bool]***)**

A `regex` is XPCE's encapsulation of the (GNU) Regular Expression library. Regular expression form a powerful mechanism for the analysis of text. Class `regex` can be used to search both `char_array` (`name` and `string`) text and text from a `text_buffer` as used by `editor`. It is possible to access the 'registers' of the regular expression.

```
?- new(S, string('Hello World')),
   new(R, regex('Hello\s +\(\w+\)')),
```

```
        send(R, match, S),
        get(R, register_value, S, 1, name, W).

    W = 'World'
```

object — region

region(*x=expression, y=expression, width=expression, height=expression***)**
A `region` defines a sub-region of a graphical. They are used to restrict `handler` objects to a sub-area of a graphical. Backward compatibility only.

object — relation

relation(*unchecked ...***)**
Class `relation` is the super-class of `identity` and `spatial`. Relations form the reusable part of `constraints`. Class `relation` may be sub-classed to define new relation-types.

object — recogniser — gesture — resize_gesture

resize_gesture(*button=[button_name], modifier=[modifier]***)**
A `resize_gesture` handles mouse-drag events to resize a graphical object on the corners or edges. See also `move_gesture` and `resize_outline_gesture`.

```
    ...,
    send(Box, recogniser, new(resize_gesture)),
    ...
```

object — recogniser — gesture — resize_gesture — resize_outline_gesture

resize_outline_gesture(*button=[button_name], modifier=[modifier]***)**
Outline version of the `resize_gesture`, often used to resize objects that are expensive to resize, such as `editor` or `list_browser`.

object — recogniser — gesture — resize_table_slice_gesture

resize_table_slice_gesture(*mode=[row,column], button=[button_name], modifier=[modifier]***)**
Gesture that can be used together with class `table` to allow the user dragging the boundaries between columns and rows in a table. See also section 11.5.

object — source_sink — resource

resource(*name=name, class=[name]***)**

A `resource` is data associated with the application. It is most commonly used to get access to image-data. Example:

```
    resource(splash, image, image('splash.gif')).

    show_splash_screen :-
            new(W, window),
            send(W, kind, popup),           % don't show border
            new(I, image(resource(splash))),
```

```
        get(I, size, size(W, H)),
        send(W, size, size(W, H)),
        send(W, display, bitmap(I)),
        send(W, open_centered),
        send(timer(2), delay),
        send(W, destroy).
```

object — rubber

rubber(*level=[1..], stretch=[0..], shrink=[0..], linebreak=[allow,force]**)

Defines how elastic objects in the document-rendering system such as `hbox` and its subclasses are. Used by `parbox` to realise layout. See section 11.10 for a full description of the rendering primitives.

object < **visual — graphical — scroll_bar**
layout_interface

scroll_bar(*object=object, orientation=horizontal,vertical, message=[code]**)

A `scroll_bar` is used to indicate and control the visible part of a large object viewed through a window. Though possible, scroll_bars are rarely used outside the context of the predefined scrollbars associated with `list_browser`, `editor` and `window`.

```
...,
send(Window, scrollbars, vertical),
...
```

object — program_object — behaviour — method — send_method

send_method(*name=name, types=[vector], implementation=code|c_pointer, summary=[string]*, source=[source_location]*, group=[name]**)

A `send_method` maps the name of a method *selector* onto an implementation and defines various attributes of the method, such as the required arguments, the source-location, etc. Send-methods are normally specified through user-defined classes pre-processor as described in chapter 7.

object — sheet

sheet(*member=attribute ...*)

A `sheet` is a dynamic set of attribute/value pairs. The introduction of object-level attributes implemented by 'object ⇆`attribute`' and user-defined classes have made sheets obsolete.

object — size

size(*width=[int], height=[int]*)

Combination of ←`width` and ←`height` used to communicate with graphical objects about dimension. See also `point` and `area`.

object < **visual — graphical — dialog_item — slider**
layout_interface

slider(*name, low=int|real, high=int|real, value=int|real|function, message=[code]**)

Controller for a numeric value inside a range that does not require exact values. Spec-

ifying volume or speed are good examples of the use of sliders. They can also be used to realise a percent-done gauge.

```
...,
new(Done, slider(done, 0, 100, 0)),
send(Done, show_label, @off),
send(Done, show_value, @off),
...
send(Done, selection, N),
send(Done, synchronise),
...
```

object — stream — socket

socket(*address=file|tuple|int*, domain=[unix,inet]***)**

Communication end-point for a TCP/IP or Unix-domain interprocess communication stream. XPCE supports both 'server' and 'client' sockets. On the Win32 platform, only TCP/IP sockets are provided and only Windows-NT supports server sockets. The library(pce_server) provides a good starting point for defining server sockets. The support executable xpce-client may be used to communicate with XPCE server sockets. See also PceEmacs server mode as defined in library('emacs/server').

object — source_location

source_location(*file=name, line=[int]****)**

Specifies the location in a source file. Used by method objects to register the location they are defined.

object — source_sink

source_sink()

Abstract super-class of file, resource and text_buffer for type-checking purposes. All source_sink objects may be used for storing and retrieving image-data. Notably a resource can be used for creating images (see section 9 and text_buffer can be used to communicate images over network connections (see section 11.9).

object — relation — spatial

spatial(*x1=[=]*, y1=[=]*, x2=[=]*, y2=[=]*, width=[=]*, height=[=]****)**

A spatial defines a geometry relation between two objects. The first two equations express the *reference* point of the 1st graphical in terms of its x, y, w and h. The second pair does the same for the second graphical, while the remaining two equations relate the mutual widths and heights. The example defines the second graphical to be 10 pixels wider and higher than the first, to share the same lower edge and be centered horizontally.

```
new(_, constraint(Gr1, Gr2,
                  spatial(xref=x+w/2, yref=y+h,
                          xref=x+w/w, yref=y+h,
                          w2=w+2, h2=h+2)))
```

object — stream

stream(*rfd=[int], wfd=[int], input_message=[code], record_separator=[regex|int]***)**

Class `stream` is the super-class of `socket` and `process`, defining the stream-communication. It handles both synchronous and asynchronous input from the socket or process. It is not possible to created instances of this class.

object — char_array — string

string(*format=[char_array], argument=any ...***)**

A `string` represents a string of characters that may be modified. This class defines a large number of methods to analyse the text that are inherited from `char_array` and a large number of methods to manipulate the text. Class `regex` can analyse and modify string objects. There is no limit to the number of characters in a string. Storage is (re)allocated dynamically and always is 'just enough' to hold the text. For large texts that need many manipulations, consider the usage of `text_buffer` that implement more efficient manipulation.

Strings are commonly used to hold descriptions, text entered by the user, etc.

object — style

style(*icon=[image]*, font=[font], colour=[colour], highlight=[bool], underline=[bool], bold=[bool], grey=[bool], background=[colour|pixmap|elevation], hidden=[bool], left_margin=[int], right_margin=[int]***)**

A `style` defines attributes for a text `fragment` as handled by a `editor`/classtext_buffer or a `dict_item` as handled by a `list_browser`/`dict`. It defines the font, fore- and background colours as well as underlining, etc. The example defines a browser that displays files using normal and directories using bold font.

```
make_browser(B) :-
        new(B, browser),
        send(B, style, file, style(font := normal)),
        send(B, style, directory, style(font := bold)),
        send(B, open).

append_file(B, Name) :-
        send(B, append,
             dict_item(Name, style := file)).
append_dir(B, Name) :-
        send(B, append,
             dict_item(Name, style := directory)).
```

object — syntax_table

syntax_table(*name=[name]*, prototype=[syntax_table]***)**

Syntax tables are used by class `text_buffer` to describe the syntax of the text. They describe the various syntactical categories of characters (word-characters, digit-characters), the syntax for quoted text, for comments as well as a definition for the end of a sentence and paragraph. Syntax tables are introduced to support the implementation of modes in PceEmacs. See also the `emacs_begin_mode/5` directive as defined in library(`emacs_extend`).

object < **visual — graphical — device — dialog_group — tab**
layout_interface

tab(*name=[name]***)**

A `tab` is a subclass of `dialog_group`, rendering as a collection of `dialog_items` with a 'tag' associated. Tabs are normally displayed on a `tab_stack`, which in turn is displayed on a `dialog`. Skeleton:

```
new(D, dialog(settings)),
send(D, append, new(TS, tab_stack)),
send(TS, append, new(G, tab(global))),
send(TS, append, new(U, tab(user))),
...,
<fill G and U>
...,
send(D, append, button(ok)),
send(D, append, button(cancel)).
```

object < **visual — graphical — device — tab_stack**
layout_interface

tab_stack(*member=tab ...***)**

Defines a stack of tagged sub-dialogs (`tab` objects) that can be displayed on a `dialog`. See `tab` for an example.

object — layout_manager — table

table()

A `table` defines a two-dimensional tabular layout of graphical objects on a graphical `device`. The functionality of XPCE tables is modelled after HTML-3. Example:

```
simple_table :-
        new(P, picture),
        send(P, layout_manager, new(T, table)),
        send(T, border, 1),
        send(T, frame, box),
        send(T, append, text('row1/col1')),
        send(T, append, text('row1/col2')),
        send(T, next_row),
        send(T, append,
             new(C, table_cell(text(spanned, font := bold)))),
        send(C, col_span, 2),
        send(C, halign, center),
        send(P, open).
```

object — layout_interface — table_cell

table_cell(*graphical***)**

Provides the `layout_interface` to a `table`. Table cells are automatically created if a graphical is appended to a `table`. Explicit creation can be used to manipulate spanning, background and other parameters of the cell.

object — vector — table_slice < table_column, table_row

table_column(*[left,right,center,reference,stretch]***)**
table_row(*[top,bottom,center,reference,stretch]***)**
table_slice()

These classes are used for storing row- and column information in `table` objects. They are normally created implicitly by the table. References to these objects can be used to change attributes or delete rows or columns from the table. Example:

```
...,
get(Table, column, 2, @on, Column),
send(Column, halign, center),
...
```

object — hbox — tbox

tbox(*text=char_array, style=[style]***)**
Add text using a defined `style` to a `parbox`. Part of the document-rendering infrastructure. See section 11.10.

object — relation_table

relation_table(*names=vector, keys=vector***)**
A `relation_table` defines a multi-column table data object that can have one or more indexed *key* fields. They are (infrequently) used for storing complex relational data as XPCE objects.

object < visual — graphical — text
layout_interface

text(*string=[char_array], format=[left,center,right], font=[font]***)**
Graphical representing a string in a specified font. Class text defines various multi-line and wrapping/scrolling options. It also implements methods for editing. Class `editable_text` as defined in library(`pce_editable_text`) exploits these methods to arrive at a flexible editable text object.

object — source_sink — text_buffer

text_buffer(*contents=[char_array]***)**
A `text_buffer` provides the storage for an `editor`. Multiple editors may be attached to the same `text_buffer`, realising shared editing on the same text. A text_buffer has an associated `syntax_table` that describes the character categories and other properties of the text contained. It can have `fragment` objects associated that describe the properties of regions in the text.

See class `editor` for an overview of the other objects involved in editing text.

object < visual — graphical — text_cursor
layout_interface

text_cursor(*for=[font]***)**
Cursor as displayed by an `editor`. Not intended for public usage. The example hides the caret from an editor.

```
...,
send(Editor?text_cursor, displayed, @off),
...
```

object < **visual — graphical — text_image** / *layout_interface*

text_image(*text=object, width=int, height=int***)**

A `text_image` object is used by the classes `editor` and `list_browser` to actually display the text. It defines the tab-stops and line-wrapping properties. It also provides methods to translate coordinates into character indices and vise-versa. The user sometimes associates `recogniser` objects with the `text_image` to redefine event-processing.

object < **visual — graphical — dialog_item — text_item** / *layout_interface*

text_item(*name=[name], default=[any|function], message=[code]****)**

A `text_item` is a controller for entering one-line textual values. Text items (text-entry-field) can have an associated `type` and/or *value-set*. If a value-set is present or can be extracted from the type using '`type` ← `value_set`', the item will perform completion, which is by default bound to the space-bar. If a type is specified, the typed value will be converted to the type and an error will be raised if this fails. The following text-item is suitable for entering integers:

```
...,
new(T, text_item(height, 0)),
send(T, type, int),
send(T, length, 8),
...
```

object < **visual — graphical — text_margin** / *layout_interface*

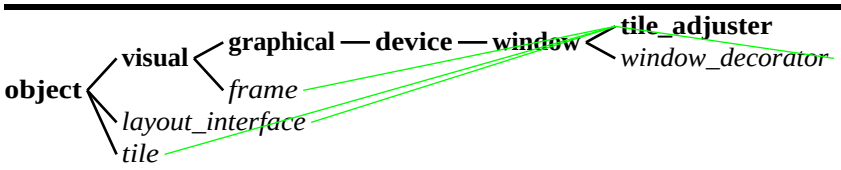
text_margin(*editor=editor, width=int, height=int***)**

A `text_margin` can be associated with an `editor` using '`editor` → `margin_width`' > 0. If the `text_buffer` defines fragments, and the `style` objects define '`style` ← `icon`', the margin will show an icon near the start of the fragment. After the introduction of multiple fonts, attributes and colour this mechanism has become obsolete.

object — tile

tile(*object=object*, width=[int], height=[int]***)**

Tiles are used to realise the 'tile-layout' of windows in a `frame`. Section 10.6 explains this in detail. Tiles can also be used to realise tabular layout of other re-sizable graphical objects.



tile_adjuster(*tile*)
Small window displayed on top of a `frame` at the right/bottom of a user-adjustable window. Dragging this window allows the user the adjust the subwindow layout.

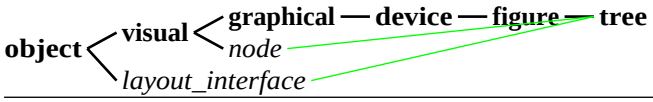
object — timer

timer(*interval=real, message=[code])**
Timers are used to initiate messages at regular intervals, schedule a single message in the future or delay execution for a specified time. The example realises a blinking graphical. Note that prior to destruction of the graphical, the timer must be destroyed to avoid it sending messages to a non-existing object.

```
...,
new(T, timer(0.5, message(Gr, inverted,
                          Gr?inverted?negate))),
...
```

object — tokeniser

tokeniser(*syntax=[syntax_table]*)
A `tokeniser` returns tokens from the input source according to the syntax specified. It is part of the XPCE object parser and its specification is not (yet) public.



tree(*root=[node])**
Trees realise hierarchical layout for `graphical` objects. Class `tree` itself is a subclass of `figure`. The hierarchy is built from `node` objects, each of which encapsulates a graphical. Trees trap changes to the geometry of the displayed graphicals and will automatically update the layout. For an example, see `node`.

object — tuple

tuple(*first=any, second=any*)
Anonymous tuple of two objects. Commonly used by get-methods that have to return two values.

object — program_object — type

type(*name=name, kind=[name], context=[any], supers=[chain*]*)
A `type` defines, implicitly or explicitly, a set of object that satisfy the type, as well as optional conversion rules to translate certain other object to an object that satisfies the type. The basic set consists of a type for each class, defining the set of all instances of the class or any of its sub-classes, a few 'primitive' types (int, char and event_id are examples). Disjunctive types can be created. See also section 3.2.1 and section 7.5.1.

```
?- get(type(int), check, '42', X). X = 42
```

object — program_object — code — function — var

var(*type=[type], name=[name], value=[any]***)**

A `var` object is a `function` that yields the stored value when evaluated. Vars in XPCE have global existence (like any object), but local, dynamically scoped, binding. Scopes are started/ended with the execution of (user-defined) methods, '`code` → `forward`' and the execution of a `block`.

The system predefines a number of `var` objects providing context for messages, methods, etc: `@arg1`, ... `@arg10` for argument forwarding, `@event` for the current event, `@receiver` for the receiver of an event or message and `@class` for the class under construction are the most popular ones. Class `block` and `and` give examples of using these objects.

object — program_object — behaviour — variable

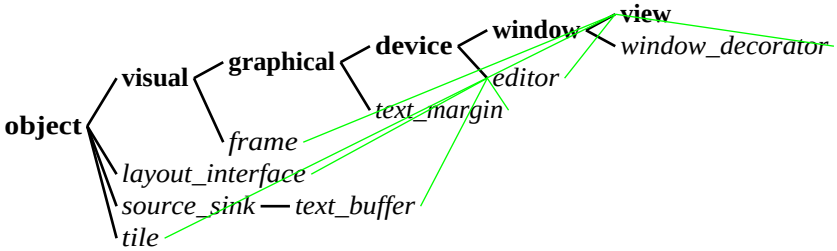
variable(*name=name, type=[type], access=[none,send,get,both], summary=[string]*, group=[name], initial_value=[any|function]***)**

A `variable` is a class' instance-variable. They are, like `send_method` and `get_method`, normally defined through the user-defined classes preprocessor described in chapter 7.

object — vector

vector(*element=any ...***)**

Vector of arbitrary objects. Vectors can be dynamically expanded by adding objects at arbitrary indices. Vectors are used at various places of XPCE's programming world: specifying the types of methods, the instance variables of a class, to pack the arguments for variable-argument methods, etc. They share a lot of behaviour with `chain` can sometimes be an attractive alternative.



view(*label=[name], size=[size], display=[display], editor=[editor]***)**

A `view` is a `window` displaying an `editor`. View itself implements a reasonable powerful set of built-in commands conforming the GNU-Emacs key-bindings. See also *PceEmacs* and `show_key_bindings/1`.

object — visual

visual(*unchecked ...***)**

Visual is the super-class of anything in XPCE that can 'visualise' things. The class itself defines no storage. Each subclass must implement the '`visual` ← `contains`' and '`visual` ← `contained_in`' methods that define the visual consists-of hierarchy as shown by the *Visual Hierarchy*. Class visual itself plays a role in the →`report`

mechanism as described in section 10.7 and defines '`visual → destroy`' to ensure destruction of a sub-tree of the visual consists-of hierarchy.

---

object — program_object — code — function — when

when(*condition=code, then=any|function, else=any|function***)**

Class `when` realises a function version of `if`. It evaluates the *condition* and then returns the return-value of either of the two functions. It is commonly used to define conditional class-variable values.

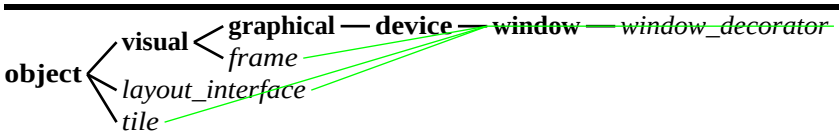
```
editor.selection_style: \
        when(@colour_display, \
             style(background := yellow), \
             style(highlight := @on))
```

---

object — program_object — code — while

while(*condition=code, statement=[code]******)**

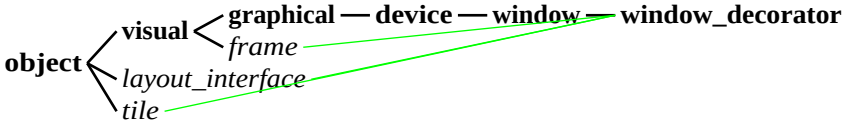
Code statement executing *body* as long as *condition* executes successfully. Not used frequently. Most iteration in XPCE uses the →`for_all`, →`for_some`, ←`find` and ←`find_all` methods defines on most collection classes.



window(*label=[name], size=[size], display=[display]***)**

The most generic XPCE window class. A `window` is a sub-class of `device` and thus capable of displaying graphical objects. One or more windows are normally combined in a `frame` as described in section 10.6. The four main specialisations of window are `dialog` for windows holding controllers, `view` for windows holding text, `browser` for windows displaying a list of items and finally, `picture` for displaying graphics.

Class `window` can be used as a graphics window if no scrollbars are needed.



window_decorator(*window=window, scrollbars=[none,vertical,horizontal,both], label=[char_array]***)**

A `window_decorator` is a window that displays another window and its 'decorations': scrollbars and label. A `picture` for example is actually a window displayed on a window-decorator displaying the scrollbars. Almost never used directly by the application programmer.

Index

'emacs/server' *library*, 273
(+), 21
+ *class*, 49
? *class*, 14, 15, 80, 243
@2535252, 223
@772024, 10
@= *class*, 244
@Atom, 8, 240
@Integer, 226, 240
@Object, 181
@Reference, 10
@_dialog_bg, 244
@_win_pen, 244
@arg1, 17, 79, 80, 107, 120, 164, 239, 241, 250, 270, 279
@arg10, 80, 279
@arg2, 80, 107, 120, 239, 270
@arg3, 120
@bo, 40
@br, 181, 183
@class, 52, 53, 65, 279
@class_default, 68
@classes, 260
@default, 10, 11, 19, 30, 54, 59, 61, 62, 106, 165, 224, 251, 254, 265
@demo, 9
@display, 10, 77, 90, 101, 233, 255
@display_manager, 255
@dragbox_recogniser, 164
@errors, 105
@event, 80, 81, 96, 101, 165, 241, 279
@event_tree, 257
@fill_pattern_menu, 89
@finder, 129
@h, 181
@hfill, 181
@my_diagram, 223
@nbsp, 181
@nil, 19, 57, 95, 156, 163, 165, 170, 202, 251, 254
@off, 76, 80, 136, 141, 234, 247, 251
@on, 30, 31, 35–37, 76, 99, 131, 136, 146, 184, 247, 251
@pce, 17, 90, 107, 233, 267
@persons, 58
@prolog, 14, 15, 17, 33, 76, 77, 80, 90, 205, 220, 221, 233, 260
@receiver, 35, 80, 81, 241, 279
@space_rubber, 181
@stones_bitmap, 85
`@display`
    →`dispatch`, 76
    →`inform`, 102
`@pce`
    ←`home`, 67
    ←`last_error`, 106, 227
    →`exception_handlers`, 107
    →`exception`, 107
    →`format`, 101

absolute_file_name/[2 3], 71
action/3, 182, 186
agent, 189
Alladin GsView, 127
alt, 265
and *class*, 80, 247, 279
annotations, 255
append_style/3, 35
application
    ←`member`, 95
    →`append`, 95
    →`delete`, 95
application *class*, 95, 96
arc *class*, 41, 245, 262
area
    ←`size`, 233
    →`set`, 11
area *class*, 268, 272
argument
    named, 11
    optional, 10
    type, 19

arguments
typed, 11
arithmetic, 81
arrow *class*, 41, 245, 262
ask questions, 32
ask_style/3, 35
assert/1, 78
assert_employee/5, 28
assoc
redefined, 107
undefined, 107
atom, 9, 266
attribute
editing, 36
attribute *class*, 239
auto_call/1, 228
autoloading, 228
axis, 149

balloon, 133
bar
→drag_message, 156
→initialise, 155
→message, 156
→range, 155
→value, 155
bar *class*, 155, 156
bar_button_group
←bar, 156
→initialise, 156
bar_button_group *class*, 156
bar_chart
⇆value, 155
→append, 155
→clear, 155
→delete, 155
→event, 155
→initialise, 153
→selection, 155
→select, 155
bar_chart *class*, 153, 155
bar_group
→initialise, 156
bar_group *class*, 155
bar_stack
←selection, 155
→initialise, 156
bar_stack *class*, 155, 156
barchart, 149, 153
behaviour
⇆group, 54
bezier *class*, 41
binary relation, 261
bitmap *class*, 22, 41, 116, 261
blackboard, 189
blinking, 278
block *class*, 247, 279
BMP
file format, 114
body/1, 173
bool *class*, 247, 251
box
→colour, 12
box *class*, 24, 41, 62, 82, 124, 155, 180, 256
branch, 261
break, 194
breakpce/1, 270
broadcast, 189
broadcast *library*, 130
broadcast/1, 189
broadcast_request/1, 189
browser *class*, 14, 15, 24, 29, 247, 252, 254, 263, 280
bug
reporting, 2
bullet_list *class*, 183
button *class*, 14, 29, 30, 75, 79
button-bar, 135

c *class*, 248
C++, 2, 11
c_pointer *class*, 51, 248
call-back, 79
call/1, 228
canonical *class*, 183
catch
errors, 105
catch-all, 23
cell
⇆note_mark, 145
CGI, 167
chain
→for_all, 79, 252, 261

→initialise, 62
→sort, 270
chain *class*, 15, 49, 79, 82, 141, 207, 223, 246, 248, 254, 279
chain_list/2, 229
chain_list/2, 230, 248
char_array *class*, 20, 249, 270, 274
chart, 149
checkpce/0, 57, 193, 230
chess *demo*, 269
circle *class*, 41, 62, 256
class
←class_variable, 67
←instance_size, 234
←instance, 224
←un_answer, 234
→get_method, 259
hierarchy, 17
structure, 17
undefined, 107
class *class*, 53, 65, 249
class_variable
⇆value, 67
class_variable *class*, 67, 250
class_variable/[3
4], 53
class_variable
←type, 67
ClassBrowser *tool*, 67
click_gesture *class*, 44, 250
clipping, 257
close/1, 227
code
→forward, 79, 80, 250, 279
code *class*, 48, 51, 156, 202, 203, 234, 239, 250, 258, 265, 267, 270
code object, 79
code_vector *class*, 250
colour
16-bits, 113
256, 113
images, 113
true, 113
colour *class*, 22, 251
colour_map *class*, 114
combo-box, 29
completion, 21, 277
compound graphical, 253
connect_gesture *class*, 42–44, 251, 263
connection *class*, 41–43, 45, 117, 251, 262, 263
consists-of, 18
constant *class*, 53, 68, 251
constraint
←relation, 252
vs. method, 60
constraint *class*, 60, 261, 271
constructor, 11, 22
control, 265
structure, 75
controller, 253
controllers, 27, 254
built-in, 29
controls
GUI tool for, 199
CORBA, 211
create *class*, 82
CUR
file format, 114
cursor, 114
cursor *class*, 116, 252, 261
Cursors *demo*, 252
curves, 267
cut-and-paste, 211

daemon, 189
date *class*, 253
DDE, 211
default arguments, 10
default/3, 61
define_style/1, 35
definition_list *class*, 183, 184
delegate_to/1, 53
delegate_to/1, 63
delegation, 218
demo programs, 18
destructor, 22
device
⇆layout_manager, 144
←graphicals, 248
←member, 92, 95
←pointed, 58
←selection, 79
→append_dialog_item, 30

→event, 121, 145
→format, 42, 143
→geometry, 121
→initialise, 121
→layout_dialog, 31, 143
→layout_dialog, 31
device *class*, 32, 41, 42, 59, 82, 121, 122, 124, 143, 144, 151, 179, 184, 247, 253, 255, 257, 258, 262, 275, 280
diagram, 149
dialog
←gap, 31
→_compute_desired_size, 31
→append, 30, 31
→apply, 37
→gap, 30
→layout, 31
→restore, 37
layout in, 29
resizing, 32
dialog *class*, 14, 27, 28, 31, 101, 135, 143, 248, 253, 254, 262, 263, 275, 280
dialog/2, 199, 202, 203, 207, 210
dialog_group
←label_name, 159
dialog_group *class*, 29, 143, 156, 159, 254, 262, 275
dialog_item
⇆label_format, 30
⇆label_width, 30
⇆reference, 30
⇆value_width, 31
←label_name, 159
→above, 29
→alignment, 30
→apply, 37
→below, 29
→default, 36
→hor_stretch, 30
→left, 29
→restore, 36
→rigth, 30
dialog_item *class*, 14, 27, 28, 58, 116, 159, 253, 254, 275
dialog_item
←alignment, 31
←label_name, 159
→alignment, 32
→label, 116
→value_width, 31
built-in types, 29
dict *class*, 14, 248, 254, 263, 274
dict_item *class*, 14, 15, 163, 208, 248, 252, 254, 263, 274
dict_item
⇆object, 254
⇆style, 254
←key, 208
dictionary, 254
direct-manipulation, 4
directory
←files, 207
←file, 208
directory *class*, 15, 140, 207, 255, 257
display
⇆address, 237
⇆font_alias, 111
←win_file_name, 131
→dispatch, 77, 257
→load_font_aliases, 111
→report, 102
display *class*, 95, 101, 109, 255
display_manager
→dispatch, 77
DLL, 212
doc/emit *library*, 180
doc/html *library*, 186
doc/objects *library*, 181
doc/vfont *library*, 182
doc:action/3, 182
doc_mode
→colour, 182
→initialise, 182
→set_font, 182
→underline, 182
doc_mode *class*, 181, 183
doc_table *class*, 184
doc_mode
←link_colour, 184
←space_mode, 181
drag-and-drop, 211
drag_and_drop_dict_item_gesture *class*, 163

drag_and_drop_gesture
→initialise, 164
drag_and_drop_gesture *class*, 163–165
dragbox, 164
→event, 164
dragdrop *library*, 211
dragdropdemo/0, 164
draw/canvas *library*, 128
drop_picture, 163
→drop, 163
→preview_drop, 163

editable_text *class*, 276
editable_graphical
→event, 64
editor
→margin_width, 277
→style, 258
editor *class*, 20, 21, 29, 60, 116, 128, 208, 218, 248, 255, 256, 258, 262, 270–272, 274, 276, 277, 279
editpce/1, 24
elevation *class*, 256, 257
ellipse *class*, 41, 256
emacs_extend *library*, 274
emacs_begin_mode/5, 274
EMF, 128
emit/3, 180–183, 186
enum_list *class*, 183
error
←feedback, 105
description of, 105
error *class*, 105, 159, 256
Error Browser *tool*, 256
errors
catching, 105
reporting, 135
event
←receiver, 59, 101
→is_a, 257
→post, 58, 59, 256
port, 202
processing, 44
event *class*, 256
event-driven, 75
event_node *class*, 257
Events *demo*, 257
exception, 107
exception-handling, 86
executable
object, 79
expand/1, 172, 175
expand_file_name/2, 67

figure
→status, 257
figure *class*, 42, 253, 256, 257, 278
file
←object, 107
get name in Windows, 131
prompting for, 131
file *class*, 15, 71, 140, 165, 171, 208, 257, 273
file_search_path/2, 72
file_type/2, 131
fill_pattern_menu/1, 91
find_file *library*, 131
finder
←file, 131
finder *class*, 129, 131
finding
graphical, 10
graphicals, 92
float, 270
floating point, 9
folder, 255
font *class*, 11, 22, 110, 111, 144, 179, 182, 257, 264
font_map/7, 182
format *class*, 42, 143, 144, 258
format/2, 172, 177
fragment *class*, 116, 256, 258, 274, 276
frame
←confirm_centered, 35, 96
←confirm, 32, 76, 77
←name, 95
→application, 95
→delete, 99
→fit, 31, 99
→icon, 116
→label, 184
→modal, 95, 96
→open, 32
→report, 101

→return, 32, 35, 205
→status, 96
→transient_for, 95
→transient_for, 35
layout, 97
frame *class*, 14, 95, 97, 99, 101, 244, 253, 258, 262, 277, 278, 280
free/1, 10, 15, 86, 225, 233
function, 49
function *class*, 19, 36, 49, 80, 250, 258, 269, 270, 279

garbage collection, 233
gesture *class*, 44, 252, 258, 259, 265, 270
get
port, 201
get/3, 9, 10, 12, 57, 225
get/[3-13], 12, 15, 67, 218, 229, 234, 237, 240, 249
get/[4-13], 57
get_chain/3, 230
get_class/3, 225
get_class/4, 56
get_implementation/4, 64
get_method *class*, 265, 279
get_object/[3-13], 229
get_super/3, 56, 225, 226
get_class/3, 56, 57
get_class/4, 226
get_implementation/4, 64
get_super/3, 57
get_super/[3-13], 56, 226
GIF
file format, 114
global objects, 17
with reconsult, 86
GNU-Emacs, 255
go/1, 167
goal_expansion/2, 12, 177
graph, 41
graphical
⇆layout_interface, 144
→_redraw_area, 61, 121
→_redraw_area, 121
→alert, 102
→compute, 60
→draw, 122
→drop, 165
→event, 58, 59, 86, 259
→flush, 40, 102
→geometry, 59
→preview_drop, 165
→preview_drop, 163
→recogniser, 86, 257
→request_geometry, 60
finding, 10, 92
name, 92
window, 39
graphical *class*, 11, 22, 24, 42, 57, 58, 116, 127, 133, 163, 167, 171, 179, 245, 253, 254, 265, 266, 278
grbox *class*, 179, 259, 260
groupware, 167
gsview, 127

handle
←kind, 251
handle *class*, 42, 43, 55, 251, 259, 263
handle/3
4, 54
handler *class*, 44, 259, 270, 271
handler_group *class*, 44, 260
hash_table *class*, 249, 254, 260
hbox *class*, 179–181, 260, 272
head/1, 173
help
on manual, 18
help_message *library*, 129, 133
hierarchy
of UI components, 194
home
web, 2
host
←messages, 76
←message, 76
→call_back, 76
→catch_all, 76
host *class*, 15, 76, 77, 248, 260
host-language, 260
html/1, 172–175
html_begin/1, 174
html_end/+End, 174
html_print_length/2, 174
html_quoted/1, 175

html_quoted_attribute/1, 175
html_write:expand/1, 175
html_write:layout/3, 175
html_begin/1, 174
html_print/[1
2], 174
html_quoted/1, 172
http/html_write *library*, 128, 171
http/httpd *library*, 170, 177
httpd
→accepted, 170
→authorization_required, 171
→forbidden, 171
→initialise, 170
→moved, 172
→not_found, 172
→reply_html, 171
→reply, 171
→request, 170
→server_error, 172
httpd *class*, 129, 170, 171
hyper
→initialise, 119
→unlink_from, 119
→unlink_to, 119
hyper *class*, 60, 117, 252, 261
hyper *library*, 117, 249

ICO
file format, 114
icon, 114
icon *class*, 143
identity *class*, 252, 261, 271
if *class*, 80, 243, 246, 261, 280
image
←hot_spot, 114
←mask, 114
→save_in, 171
file formats, 114
shape, 114
image *class*, 11, 22, 114, 116, 124, 127, 135, 159, 247, 252, 261, 262, 264, 268
image/gif, 171
image/jpeg, 171
inheritance
multiple, 218
of classes, 218
initialisation
failed, 107
inspecting instances, 18
inspector, 196
Inspector *tool*, 257
int, 9, 266
int_item *class*, 29
interpolated, 267
ispell *demo*, 269

joint
→arrows, 245
joint *class*, 245, 262
JPEG
file format, 114

key_binding *class*, 44, 255, 259, 262, 270
keyboard-accelerators, 262

label
→report, 62, 102
→selection, 116
label *class*, 29, 101, 116, 135, 262
lambda, 250
lambda functions, 79
layout
in dialog, 29
manager, 144
window, 97
layout of graphicals, 258
layout/3, 175
layout_interface *class*, 275
layout_manager *class*, 262
lbox *class*, 179, 183
line *class*, 41, 43, 245, 262, 263
line-wrapping, 277
link *class*, 43, 251, 263
list_browser *class*, 29, 116, 163, 207, 247, 248, 254, 262, 263, 271, 272, 274, 277
list_browser
→members, 207
listen/2, 190
listen/3, 190, 191
listening/3, 191
look-and-feel, 3

make_dialog/2, 202
make_dialog/2, 199, 202, 203, 205
make_dragbox_recogniser/1, 164
make_picture/1, 169
make_verbose_dialog/0, 50
manpce/0, 2, 199, 231
manpce/1, 24, 231
mathematical symbols, 111
memory usage, 234
menu
    →kind, 264
menu *class*, 29, 31, 133, 264, 265
menu_bar *class*, 29, 136, 268
menu_item
    ←label_name, 159
menu_item *class*, 265, 269
menu_item
    ⇆accelerator, 262
    →selection, 116
message
    →execute, 82
    service, 189
message *class*, 14, 15, 28, 44, 76, 79, 80, 239, 248, 265
messages
    reporting, 135
meta, 265
method *class*, 241, 245, 265, 273
mime-type, 171
MIT_MAGIC_COOKIE, 237
modal, 77, 95, 96
    dialog, 32
modes, 274
modifier *class*, 22, 164, 265
Motif, 3
mouse, 252
move_gesture *class*, 44, 265, 271
move_outline_gesture *class*, 44, 265
moving graphicals, 44
MS-Windows, 3
multi-segment line, 267
multiple inheritance, 218
my_httpd, 167
    →request, 167

name
    ←label_name, 135
name *class*, 9, 11, 20, 22, 47, 144, 223, 249, 254, 265, 266, 270
name_of/2, 190
named argument, 11
named pipes, 211
Netscape, 114
new
    failed, 107
new/2, 7–10, 15, 22, 27, 85, 106, 203, 218, 223, 224, 226, 234, 237, 249
node *class*, 141, 218, 278
notation
    in this manual, 12
notracepce/1, 194, 230
number *class*, 266, 270

object
    ⇆attribute, 245
    ⇆slot, 54
    ←catch_all, 23
    ←class_variable_value, 67
    ←convert, 23, 57
    ←get_super, 57
    ←hypered, 120
    ←lookup, 22, 58
    ←slot, 56
    →_check, 230
    →_save_in_file, 53
    →catch_all, 23
    →done, 77, 234
    →error, 106, 256
    →free, 233
    →initialise, 22, 57
    →protect, 225, 233
    →report, 101, 256
    →save_in_file, 210
    →send_hyper, 120
    →send_super, 57
    →slot, 56
    →unlink, 22, 57
    inspecting, 18, 196
    management, 234
    memory usage, 234
    reference, 8
    remove, 10, 22
    removing, 233
object *class*, 19, 63, 101, 119, 267, 270

object/1, 226
object/2, 53, 55, 226
objects
    creating, 22
obtainer, 15, 80
OLE, 211
once/1, 106
open
    icon, 44
    object as stream, 227
open_resource/4, 71
open_resource/3, 71
OpenWindows, 3
optional arguments, 10
or *class*, 267

page/1, 173
page/2, 173
page/[1
    2], 173
paragraph, 258
parbox
    ←content, 182
    →cdata, 180
parbox *class*, 179–181, 259, 267, 272, 276
path *class*, 41, 153, 245, 246, 262, 263, 267
pbox
    ←anchor, 182, 184
    →event, 182
    →show, 181
pbox *class*, 181, 184
pce
    ←exception_handlers, 107
    ←window_system, 212
    →exception, 107
pce *class*, 267
pce_begin_class/0, 54
pce_begin_class/[2
    3], 51
pce_catch_error/2, 106, 227
pce_class_directive/1, 53, 65
pce_editable_text *library*, 255, 276
pce_end_class/1, 54
pce_global/2, 226
pce_group/1, 54
pce_image_browser *library*, 143
pce_open/3, 227
pce_portray *library*, 10
pce_renew *library*, 107
pce_report *library*, 135
pce_server *library*, 273
pce_autoload/2, 107
pce_begin_class/[2
    3], 51, 54, 69
pce_catch_error/2, 105, 106, 256
pce_class_directive/1, 53, 65
pce_end_class/0, 51, 53, 54, 69
pce_end_class/1, 54
pce_global/2, 85, 86, 107, 226
pce_group/1, 51, 56
pce_image_directory/1, 67, 72
pce_open/3, 227
PceEmacs *tool*, 263, 269, 279
percent-done, 273
picture *class*, 39, 268, 280
pie-chart, 244
pie-slice, 41
pixmap *class*, 41, 171, 247, 251, 268
pl/html_write *library*, 172
plot, 149
plot/axis *library*, 149
plot/barchart *library*, 153
plot/demo *library*, 153
plot/plotter *library*, 151
plot_axis
    ←location, 149
    ←value_from_coordinate, 149
    →format, 149
    →initialise, 149
    →label, 149
plot_axis *class*, 149, 151, 153
plot_graph
    →append, 153
    →initialise, 153
plot_graph *class*, 151, 153
plot_point *class*, 153
plotter
    ←translate, 149, 153
    ←value_from_x, 153
    ←value_from_y, 153
    →axis, 151
    →clear, 151
    →graph, 151

plotter *class*, 149, 151, 153
PNM
    file format, 114
point *class*, 8, 62, 153, 272
pointer, 252
popup *class*, 29, 140, 264, 265, 268, 269
popup_gesture *class*, 268, 269
portray/1, 10
PostScript, 82, 127
predicate/5, 175
print_html/1, 174
print_html/2, 174
print_html/[1
    2], 172, 175
printing, 127
private colourmap, 114
process *class*, 269, 274
progn *class*, 269
program
    as object, 79
program_object *class*, 270
Prolog interface, 17
prolog-term
    life-time, 50
prolog_term *class*, 48, 50, 261
prompt, 32
pulldown, 268
push-button, 29

quit/0, 77
quote_function *class*, 270

radio-button, 29
re-usability, 91
read
    text from object, 227
real *class*, 9, 11, 47, 270
recogniser
    →event, 59
Recogniser *class*, 44
recogniser *class*, 45, 58, 250, 257–260,
        270, 277
recorda/2, 78
redefined
    object reference, 107
reference
    object, 8
regex *class*, 270, 274
region *class*, 271
Regular Expression, 270
relation *class*, 252, 261, 271, 273
relation_table *class*, 276
remove
    objects, 10
reply/2, 168, 169
report_dialog *class*, 135
reporter *class*, 135
reporting, 135
require/1, 228
resize_gesture *class*, 44, 271
resize_outline_gesture *class*, 44, 271
resource *class*, 71, 72, 171, 271, 273
resource/3, 71
reusability, 22
RGB, 212
rubber *class*, 179, 180
runtime generation, 65
runtime-system, 105

screen, 255
scroll_bar *class*, 272
send
    port, 201
send/2, 8–10, 12, 57, 224, 229
send/3, 49, 50
send/[2-12], 12, 15, 27, 53, 61, 68, 69,
        218, 224, 225, 228, 234, 237,
        241, 249
send/[3-12], 57
send_class/3, 56, 225
send_implementation/3, 64
send_list/2
    3, 228
send_method *class*, 252, 265, 272, 279
send_super/2, 56, 225, 226
send_class/2, 56, 57
send_class/3, 56, 226
send_implementation/3, 64
send_list/2, 228
send_super/2, 56, 57
send_super/[2-12], 56, 226
sensitive, 44
set_style/2, 35
set_verbose/0, 49

sformat/3, 173
sgml *library*, 175
shared editing, 256
sheet *class*, 171, 272
shift, 265
shift-click, 265
show_slots/1, 230
show_key_bindings/1, 279
show_slots/1, 194
size *class*, 146, 268
slider *class*, 29
smooth, 41
socket
    ←peer_name, 170
socket *class*, 129, 170, 269, 274
source code
    from dialog editor, 202
source_sink *class*, 71, 273
spatial *class*, 44, 246, 252, 271, 273
spy, 194
spy/1, 194
spypce/1, 194
statement
    as object, 79
stdarg, 62
stream
    open Object as, 227
stream *class*, 274
string
    →format, 62, 101
string *class*, 11, 20, 81, 82, 171, 249, 270, 274
style
    ←icon, 277
    →icon, 116
style *class*, 11, 248, 251, 258, 274, 276, 277
summary line, 18
SVG, 128
syntax_table *class*, 276

tab *class*, 29, 159, 254, 275
tab-stops, 277
tab_stack *class*, 29, 275
table
    →border, 146
    →cell_padding, 146
    →cell_spacing, 146
    →frame, 146
    →next_row, 146
    →rules, 146
table *class*, 42, 143–145, 184, 262, 271, 275, 276
table_cell *class*, 185
table_column *class*, 145
table_row *class*, 145, 185
table_row
    →end_group, 185
tabular, 258
    ←table, 145
    →append_label_button, 145
    →append, 144
    →event, 145
    →initialise, 144
    →sort_rows, 145
    →table_width, 145
tabular *class*, 144, 145
tabular *library*, 143
tbox *class*, 179, 180, 185, 260
TCP/IP, 273
template *class*, 63, 241
term_expansion/2, 64
text *class*, 41, 67, 124, 144, 248, 255, 262
text-entry field, 29
text-entry-field, 277
text_buffer *class*, 102, 171, 255, 256, 258, 270, 273, 274, 276, 277
text_cursor *class*, 255
text_image *class*, 60, 128, 248, 255, 277
text_item *class*, 29, 30, 207, 261, 262, 277
text_margin *class*, 255, 277
text_buffer
    ←read_as_file, 227
    ←size_as_file, 227
    →format, 227
    →write_as_file, 227
text_item
    ⇆type, 201, 207
    ←selection, 202
    →message, 207
tick-box, 29
tile
    →hor_shrink, 98
    →hor_stretch, 98

→ideal_height, 98
→ideal_width, 98
→ver_shrink, 98
→ver_stretch, 98
tile *class*, 97, 98, 258
toc_file
→indicate, 141
toc_file *class*, 140, 141
toc_folder
→initialise, 141
toc_folder *class*, 140, 141
toc_window
←node, 141
←popup, 140
←selection, 141
→expand_node, 140
→expand_root, 141
→open_node, 140
→root, 141
→select_node, 140
→son, 141
toc_window *class*, 139, 140
toc_window
→son, 140
tokeniser *class*, 278
tool_bar
→activate, 135
→append, 135
→initialise, 135
tool_bar *class*, 135
tool_button
→activate, 136
→active, 136
→initialise, 135
tool_button *class*, 135
tool_status_button *class*, 135, 136
tool_bar
←client, 135
→append, 135
toolbar *library*, 135
trace/0, 194
tracepce/1, 24, 193, 194, 230, 231, 270
transient
frame, 95
tree *class*, 129, 139, 220, 266, 278
triangle, 267
type, 11, 19
←convert, 19
←value_set, 277
conversion, 11, 23
type *class*, 19, 61, 277, 278

UI
structure of, 18
undefined
class, 107
object reference, 107
undo, 255
unlisten/1, 191
unlisten/2, 191
unlisten/3, 191
use_class_template/1, 53
use_class_template/1, 63, 64, 241
user-defined graphics, 253
user_help/0, 199

var *class*, 47, 65, 80, 245, 250, 279
vararg, 62
variable
→clone_style, 65
variable *class*, 240, 245, 279
variable number of arguments, 62
variable/[3
4], 53
vector *class*, 250
version, 7
vfont
←font, 182
vfont *class*, 182, 183
view
→load, 15
view *class*, 15, 29, 218, 255, 279, 280
view/1, 14, 15
visual
←contained_in, 279
←contains, 279
←help_message, 133
→destroy, 280
→help_message, 133
→report, 101
visual *class*, 95, 101, 133, 244, 255
Visual Hierarchy *tool*, 255, 279
visuals, 113

WAIS, 24

when *class*, 280
while *class*, 243, 246
win_metafile *class*, 128
win_printer *class*, 128
window
    ←`height`, 12
    →`_redraw_area`, 122
    →`above`, 97
    →`create`, 97
    →`open`, 97
    →`resize_message`, 32
    coordinates, 39
    layout in frame, 97
    size specification, 98
window *class*, 14, 27, 39, 42, 97, 139, 144, 218, 253, 262, 268, 272, 279, 280
window_decorator *class*, 280
Windows 95, 3, 211
Windows NT, 3, 211
WinSock, 211
WMF, 128
write
    text to object, 227
WWW
    addresses, 2
wysiwyg/1, 33

X-windows, 3
X11, 252
xauth, 237
XBM
    file format, 114
xhost, 237
xpce-client, 273
XPM
    file format, 114