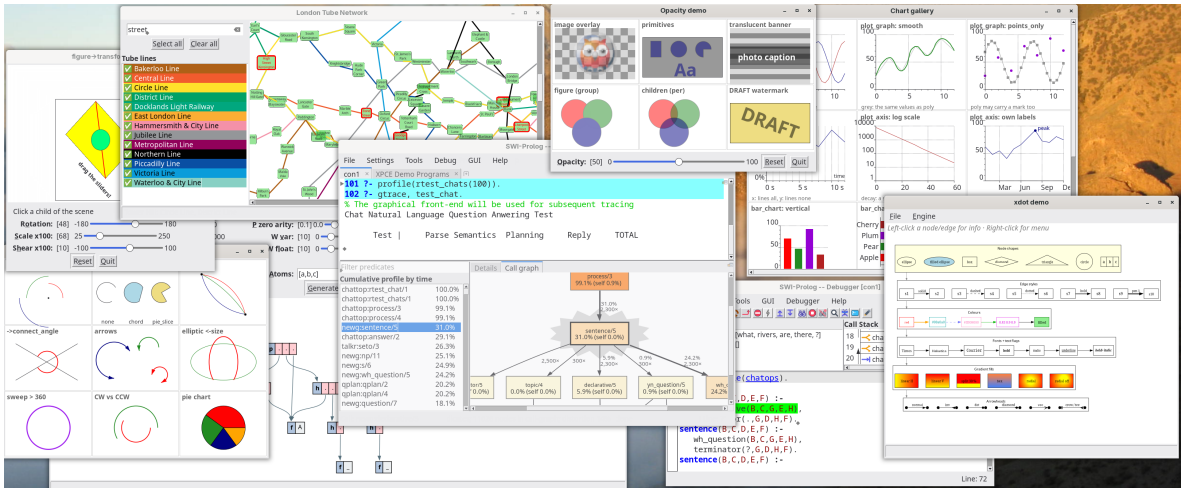
Programming in XPCE/Prolog



Jan Wielemaker
Anjo Anjewierden

Programming in XPCE/P**rolog**

Jan Wielemaker jan@swi-prolog.org
Anjo Anjewierden†

XPCE/Prolog is a hybrid environment integrating logic programming and object-oriented programming for building Graphical User Interfaces. Applications in XPCE/Prolog are fully compatible across the supported platforms.

This document applies to XPCE/Prolog 10.0.0 distributed as integrated package to SWI-Prolog. Online, the XPCE classes are documented by the class browser and reference manual that are started using `manpce/0` (see chapter 3).

Much of this manual was written for XPCE versions 5 and 6. It has been revised for the current version, but some parts may still reflect older versions. Appendix A (SDL backend), appendix B (Windows notes), section 10 (multi-threading) and appendix F (status and future) are new. The main changes since version 6 are below.

- While version 6 has two *backends*, one for X11 and one for Windows, version 7 and later are based on portable libraries. Notably
  - SDL3 is used for window management and accessing devices.
  - Cairo is used for 2D graphics
  - Pango is used for rendering of text, with an emphasis on internationalization.

  With this change of backend, XPCE now supports X11, Wayland, MacOS Cocoa and Windows. Notably the support for Wayland and Cocoa rather than X11 makes XPCE future proof and provides a better user experience, notably on MacOS.
- The Windows `swipl-win.exe` and Qt `swipl-win` app used on Linux and MacOS have been replaced by terminal code added to XPCE, resulting in the *Epilog* Prolog console that runs on all platforms.
- Font, colour and image management is modernized and synchronized over all platforms. Pango is used for font management. Colours use the 32-bit RGBA model (Red/Green/Blue/Alpha). All images are RGBA images. Loading images is done by the SDL_image library, supporting many more formats, including SVG. Graphics are exported as PDF; PostScript support has been removed.
- XPCE supports *themes*. Colours used by the user interface are *semantic* colours that follow the selected (light or dark) theme and, where possible, the colours of the desktop. Dialog items have a flat, themed look and can be operated using the keyboard.
- Coordinates and sizes are no longer restricted to integers. The type `num` accepts both integers and floats.
- The development tools are integrated in a single window that holds the tools as tabs and panes.

- XPCE 7 relaxes multi-threaded access. When XPCE calls out to Prolog its global lock is released, to be re-acquired when Prolog has finished and control returns to XPCE. This model is similar to the Python *GIL*.
- The C and C++ interfaces for embedding XPCE in other languages than Prolog have been removed.

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 BSD-2 for more details.

Last updated November 2026 for XPCE version 10.0.0

Copyright © 1992-2026 University of Amsterdam and SWI-Prolog Solutions b.v.

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 . . . 2
- 1.5 Look-and-feel . . . 2
- 1.6 A brief history of (X)PCE . . . 3
- 1.7 About this manual . . . 4
- 1.8 Acknowledgements . . . 4

2 Getting started **6**
- 2.1 Starting XPCE/Prolog . . . 6
- 2.2 Prolog ... and what? . . . 6
  - 2.2.1 Creating objects: new . . . 6
  - 2.2.2 Modifying object state: send . . . 7
  - 2.2.3 Querying objects: get . . . 8
  - 2.2.4 Removing objects: free . . . 9
- 2.3 Optional arguments . . . 9
- 2.4 Named arguments . . . 9
- 2.5 Argument conversion . . . 10
- 2.6 Send and get with more arguments . . . 10
- 2.7 Notation . . . 11
- 2.8 Example: show files in directory . . . 11
- 2.9 Summary . . . 14

3 Using the online manual **15**
- 3.1 Overview . . . 15
- 3.2 Notational conventions . . . 16
  - 3.2.1 Argument types . . . 17
- 3.3 Guided tour . . . 18
  - 3.3.1 Class browser . . . 18
  - 3.3.2 Reading cards . . . 21
  - 3.3.3 Search tool . . . 22
  - 3.3.4 Class hierarchy . . . 22
- 3.4 Summary . . . 24

4 Dialog (controller) windows **25**
- 4.1 An example . . . 25
- 4.2 Built-in dialog items . . . 26
- 4.3 Layout in dialog windows . . . 28
  - 4.3.1 Practical usage and problems . . . 29

4.4 Modal dialogs: prompting for answers . . . 30
4.4.1 Example: a simple editor for multiple fonts . . . 31
4.5 Editing attributes . . . 33
4.5.1 Example: editing attributes of a graphical . . . 34

5 Simple graphics **36**
5.1 Graphical building blocks . . . 36
5.1.1 Available primitive graphical objects . . . 37
5.2 Compound graphicals . . . 38
5.3 Opacity . . . 39
5.4 Connecting graphical objects . . . 39
5.5 Constraints . . . 39
5.6 Activating graphicals using the mouse . . . 41
5.7 Summary . . . 41

6 Paths, paint and text **43**
6.1 Paths . . . 43
6.2 Paint . . . 45
6.3 Line styles . . . 46
6.4 Text with markup . . . 47
6.5 Transforming, clipping and blending . . . 48
6.6 The canvas . . . 50
6.7 Saving drawings . . . 52

7 Input devices and media **53**
7.1 Events and input devices . . . 53
7.2 Positions and scrolling . . . 54
7.3 Pens, touch screens and gestures . . . 54
7.3.1 Touch screens . . . 55
7.3.2 Pinch gestures . . . 56
7.4 Gamepads . . . 56
7.5 Sound . . . 57
7.6 Video from a camera . . . 58
7.6.1 Showing video without Cairo . . . 59
7.6.2 Drawing with the GPU . . . 59
7.6.3 Playing and recording video . . . 61
7.6.4 Streaming video . . . 62
7.7 The desktop . . . 63
7.7.1 Images on the clipboard . . . 63
7.7.2 The system tray . . . 64
7.7.3 Power and language . . . 64
7.8 Testing input without the devices . . . 64

8 XPCE and Prolog **66**
8.1 XPCE is not Prolog! . . . 66
8.2 Dealing with Prolog data . . . 67
8.2.1 Life-time of Prolog terms in XPCE . . . 69

9 Defining classes **70**
9.1 The class definition skeleton . . . 70
9.1.1 Definition of the template elements . . . 70
9.2 Accessing instance variables (slots) . . . 73
9.3 Refining and redefining methods . . . 75
9.3.1 General redefinitions . . . 76
9.3.2 Redefinition in graphical classes . . . 77
9.4 Handling default arguments . . . 80
9.5 Advanced topics . . . 80
9.5.1 More on type declarations . . . 80
9.5.2 Methods with variable number of arguments . . . 81
9.5.3 Implementation notes . . . 83

10 Using Prolog threads **84**

11 Class Variables **86**
11.1 Accessing Class Variables . . . 86
11.2 Class variable and instance variables . . . 87
11.3 The 'Defaults' file . . . 87
11.4 Class variables in User Defined Classes . . . 88

12 Panes: the windows of the IDE **90**
12.1 Anatomy of a window . . . 90
12.2 Rearranging the panes . . . 91
12.2.1 Keys . . . 91
12.3 Where a new pane goes . . . 92
12.4 Learning how you arrange your windows . . . 93
12.4.1 The store . . . 93
12.4.2 Saying it yourself . . . 93
12.5 Writing a window down . . . 94
12.6 Class variables . . . 94

13 Program resources **97**

14 Programming techniques **99**
14.1 Control-structure of XPCE/Prolog applications . . . 101
14.1.1 Event-driven applications . . . 101
14.1.2 XPCE and existing applications . . . 103
14.2 Executable objects . . . 105
14.2.1 Procedures . . . 105
14.2.2 Functions . . . 106
14.2.3 Example: Finding objects . . . 108
14.3 Defining global named objects . . . 109
14.3.1 Using directives . . . 109
14.3.2 Inline testing . . . 109
14.3.3 The 'pce_global' directive . . . 110
14.3.4 Global objects for recognisers . . . 110

14.4 Using object references: “Who’s Who?” . . . 111
14.4.1 Global named references . . . 111
14.4.2 Using the prolog database . . . 112
14.4.3 Using object-level attributes . . . 113
14.4.4 Using window and graphical behaviour . . . 113
14.4.5 Using user defined classes . . . 114
14.4.6 Summary . . . 115
14.5 Relating frames . . . 117
14.5.1 Class application . . . 117
14.5.2 Transient frames . . . 117
14.5.3 Modal operation . . . 118
14.6 Window layout in a frame . . . 119
14.6.1 Windows sizes and automatic adjustment . . . 120
14.6.2 Manipulating an open frame . . . 121
14.7 Informing the user . . . 123
14.7.1 Aim of the report mechanism . . . 123
14.7.2 The report interface . . . 123
14.7.3 Redefining report handling . . . 124
14.7.4 Example . . . 124
14.8 Errors . . . 127
14.8.1 Handling errors in the application . . . 128
14.8.2 Raising errors . . . 128
14.8.3 Repairable errors . . . 129
14.9 Specifying fonts . . . 131
14.9.1 Logical fonts . . . 131
14.9.2 Unicode . . . 132
14.10 Using images and cursors . . . 133
14.10.1 Colour handling . . . 133
14.10.2 Supported Image Formats . . . 143
14.11 Using hyper links to relate objects . . . 145
14.11.1 Programming existence dependencies . . . 145
14.11.2 Methods for handling hyper objects . . . 147
14.12 User defined graphicals . . . 149
14.12.1 (Re)defining the repaint method . . . 149
14.12.2 Example-I: a window with a grid . . . 150
14.12.3 Example-II: a shape with text . . . 152
14.13 Printing . . . 153

15 Commonly used libraries **154**
15.1 Library “find_file” . . . 155
15.2 Showing help-balloons . . . 157
15.3 Dialog support libraries . . . 159
15.3.1 Reporting errors and warnings . . . 159
15.3.2 Toolbar support . . . 159
15.3.3 Example . . . 160
15.4 Library “pce_toc”: displaying hierarchies . . . 163
15.5 Tabular layout . . . 167

15.5.1 Using `format` . . . 167
15.5.2 Using `table` using the "tabular" library . . . 168
15.6 Plotting graphs and barcharts . . . 173
15.6.1 Painting axis . . . 173
15.6.2 Plotting graphs . . . 174
15.6.3 Drawing barcharts using "plot/barchart" . . . 176
15.6.4 Hierarchical distributions using "plot/hierarchy_chart" . . . 179
15.7 Multi-lingual applications . . . 183
15.8 Drag and drop interface . . . 187
15.8.1 Related methods . . . 188
15.8.2 Dropping files from other applications . . . 189
15.9 Document rendering primitives . . . 191
15.9.1 The rendering library . . . 192
15.9.2 Predefined objects . . . 193
15.9.3 Class and method reference . . . 193
15.9.4 Using the "doc/emit" library . . . 197

16 Development and debugging tools **200**
16.1 Object-base consistency . . . 200
16.2 Tracing methods . . . 201
16.3 Breaking (spy) on methods . . . 201
16.4 Visual hierarchy tool . . . 201
16.5 Inspector tool . . . 202

A SDL video drivers **205**

B Notes on XPCE for MS-Windows **206**
B.1 Currently unsupported features in the Windows version . . . 206
B.2 Interprocess communication, extensions and interaction . . . 206
B.3 Accessing Windows Graphics Resources . . . 206
B.4 Accessing Windows Colours . . . 207
B.5 Unicode text on Windows . . . 207

C XPCE/Prolog architecture **209**
C.1 What is "Object-Oriented"? . . . 209
C.2 XPCE's objects . . . 209
C.2.1 Classes . . . 209
C.3 Objects and integers . . . 210
C.4 Delegation . . . 210
C.5 Prolog . . . 210
C.6 Executable objects . . . 212
C.7 Summary . . . 213

D Interface predicate definition **214**
D.1 Basic predicates . . . 214
D.1.1 Declaration of required library predicates . . . 219
D.2 Additional interface libraries . . . 219

D.2.1 Library "pce_util" . . . 219
D.2.2 Library "pce_debug" . . . 221
D.2.3 Accessing the XPCE manual . . . 222

E Memory management **223**
E.1 Lifetime of an object . . . 223
E.2 References held by Prolog . . . 224
E.2.1 Porting code that inspects references . . . 224
E.3 Practical considerations . . . 225
E.4 Memory usage of objects . . . 225

F XPCE status and plans **227**
F.1 The future for XPCE . . . 228

G Glossary **229**

H Class summary descriptions **232**

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 target language for XPCE is *Prolog*. XPCE is bundled with SWI-Prolog and this document describes XPCE/Prolog. 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. The SWI-Prolog console *Epilog*, that is used by `swipl-win`, is written in XPCE as well.

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.

https://www.swi-prolog.org/packages/xpce/UserGuide/

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

- The XPCE Reference Manual
  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. The sources of the reference manual are markdown files that are rendered as HTML by the online manual tools. See also chapter 3.

- PceDraw: An example of using XPCE
  The sources of the drawing tool PceDraw (`pcedraw`, located in the directory `draw` of the XPCE Prolog library) illustrate the (graphical) functionality of XPCE and are 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 and the XPCE wiki:

```
https://www.swi-prolog.org/packages/xpce/
https://github.com/SWI-Prolog/packages-xpce/wiki
```

Questions and problems can be discussed with other programmers on the SWI-Prolog Discourse forum, which is also used to announce new releases and developments. Bugs should be reported using the issue tracker of the XPCE GitHub repository.

```
https://swi-prolog.discourse.group/
https://github.com/SWI-Prolog/packages-xpce/issues
```

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, although SWI-Prolog is currently the only maintained host language. The C and C++ embedding API of older versions has been removed. 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.

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, 64-bits memory model. XPCE's graphical classes (including windows, etc.) interface to the hosting platform using widely supported portable libraries: SDL, Cairo and Pango. XPCE is actively maintained for Linux, MacOS and Windows.

1.5 Look-and-feel

XPCE is not implemented on top of a standard UI library such as GTK, Qt, or Win32. Instead, it is built on top of its own base layer 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 class variables, whose values may be changed in the user's *Defaults* file. See section 11. The colours are defined by a *theme* that, by default, follows the light or dark appearance and the accent colour of the desktop. The GUI preferences can be edited interactively using Settings/Edit GUI preferences of the development tools.

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 Edinburgh (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

https://www.swi-prolog.org for details.

As of XPCE 7.0, the license terms are changed to BSD-2 and portability of the system was handed to the widely supported libraries: SDL, Cairo and Pango. XPCE was extended to provide the "app" Prolog console (named *Epilog*). XPCE 7 provides more flexible support for Prolog's multi-threading.

XPCE 8 and 9 continued the modernisation. The C and C++ embedding API and PostScript generation were removed and the reference manual was converted to markdown. Colours follow themes that can be switched at runtime, the dialog items got a flat look and can be operated from the keyboard, and the development tools became panes that can be combined in the windows of a single IDE.

1.7 About this manual

This 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. Chapter 6, "Paths, paint and text" describes the drawing facilities based on Cairo and Pango and chapter 7, "Input devices and media" describes input from pens, touch screens and game controllers as well as sound, video, the clipboard and the system tray.

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 8, "The relation between XPCE and Prolog" is a brief intermezzo, explaining the relation between XPCE and Prolog data in more detail. Chapter 9, "Defining classes" explain the definition of new XPCE classes from Prolog and thus brings object-oriented programming to the user. Chapter 14, "Programming techniques" is an assorted collection of hints on how XPCE can be used to solve real-world problems elegantly. Chapter 15, "Commonly used libraries" documents some of the commonly used XPCE/Prolog libraries.

Chapter 16, "Development and debugging tools" introduces the tools for finding problems in XPCE applications: checking the consistency of the object base, tracing methods and examining the structure of a running user interface.

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 (no longer maintained) interface to SICStus Prolog was implemented in cooperation with Stefan Andersson and Mats Carlsson from SICS.

The (no longer maintained) 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)).
```

XPCE can be used from `swipl` running in a terminal as well as from `swipl-win`, which runs the Prolog toplevel in an *Epilog* console window. On Windows, Prolog programs are normally loaded and started by double-clicking a `..pl` file, which starts `swipl-win`. XPCE, being a normal library, does not change this.

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 = <pce>(0x7f8a12c400,point)

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

The first example creates an instance of class point from the arguments '10' and '20'. A XPCE generated reference is an opaque handle, written ⟨*pce*⟩`(Address,Class)`. It is unique, and it may be typed or pasted back into a later query as the examples below do: the toplevel looks the handle up again. It is not a term to take apart, however; see section E.2. 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 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(<pce>(0x7f8a12c400,point), 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 desktop.

[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.

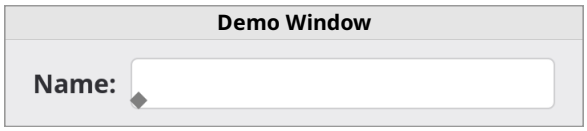


Figure 2.1: Example Dialog Window

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 floating point numbers (type `num`) as well as `real` objects, that are translated to Prolog floating point numbers. Examples:

```
6 ?- get(<pce>(0x7f8a12c400,point), y, Y).
Y = 20.
7 ?- get(@demo, display, D).
D = <pce>(0x7f8a12c290,display).
8 ?- get(<pce>(0x7f8a12c400,point), 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 the screen (monitor) showing the window.[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 the point 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. Here, @display refers to the *current* display, i.e., the screen on which XPCE opens new windows by default.

```
9 ?- get(@display, size, Size),
     get(Size, width, W),
     get(Size, height, H).
Size = <pce>(0x7f8a12c5d0,size),
W = 1920,
H = 1080.
```

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).
```

[3] An anonymous reference prints itself as ⟨*pce*⟩(*Address*,*Class*). For named references the `pce_portray` defines a clause for the Prolog predicate `portray/1` that prints them as '@Reference/Class'. This library is used throughout all the examples of this manual.

```
TextItem = <pce>(0x7f8a12c7a0,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(<pce>(0x7f8a12c400,point)).
13 ?- free(@demo).
14 ?- free(@display_manager).
false.
```

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.

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 13 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 9.3.1), 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/[2-8]` and `get/[3-8]`. 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(10,10), 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`. In compiled code 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.

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

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

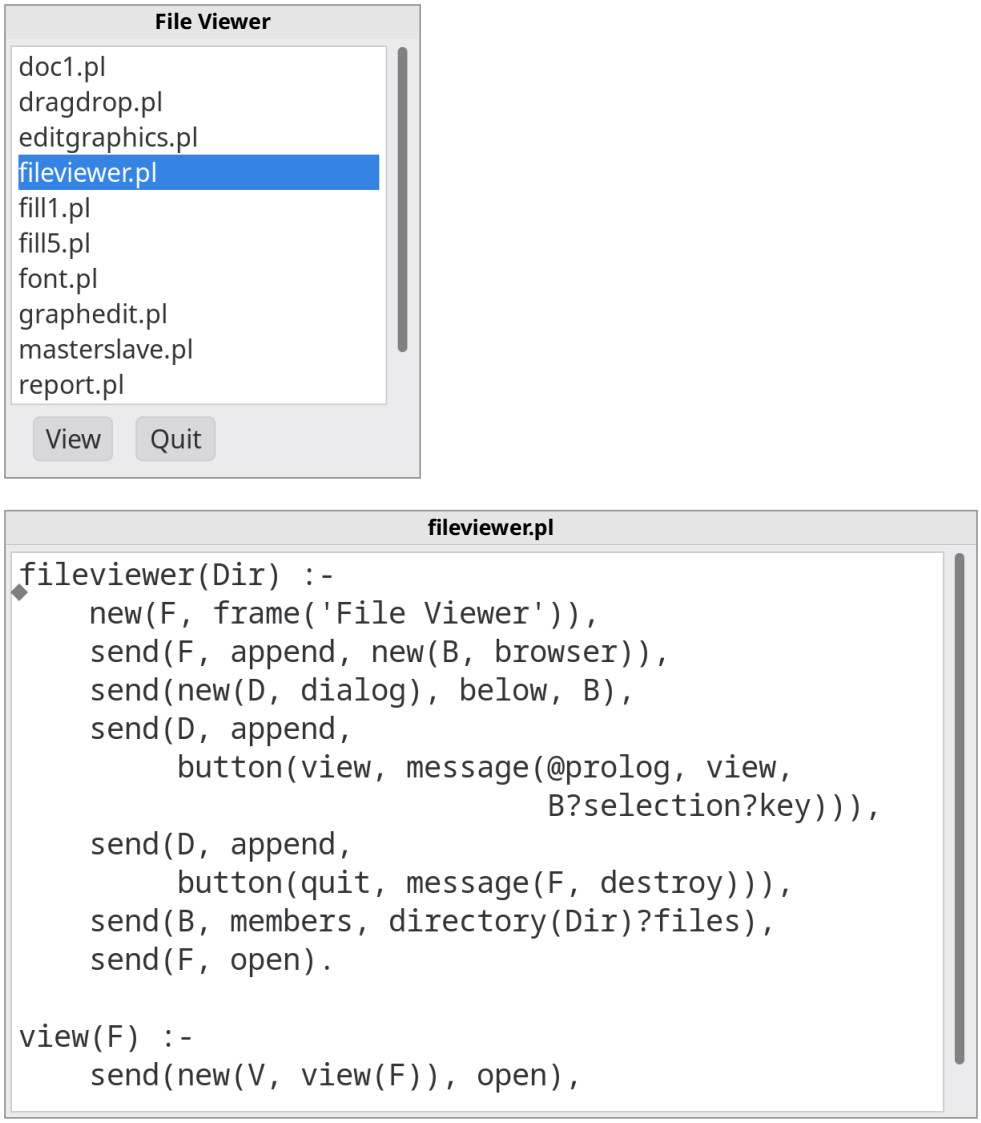


Figure 2.2: The FileViewer demo

frame ensures that its member windows are properly aligned and handles resizing of the frame. See also section 14.6.

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 15.7, 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, DirObj, 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 (2 here).

The second 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`.

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 `DirObj?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 chain is sent to the members-method of the browser *B*.

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 →`open`ed. 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/2` callback predicate opens an instance of `view`, a window specialised in text-editing, gets the file-object for the given filename F. 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.

[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.

[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`.

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/[2-8]` to change the state of an object and succeeds if the requested change could be made, `get/[3-8]` 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.

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 190 built-in classes defining over 3000 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 natural language descriptions are taken from the XPCE reference manual. For classes defined in Prolog, the PlDoc comments in the source are used.

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.

The manual is started using `manpce/0`, which opens a window with a menu bar from which the tools described below can be started. Like the other development tools, each manual tool is a *pane* that appears as a tab or beside other panes in a window of the development environment. The most commonly used tools may also be started from the GUI menu of the Epilog console and the other windows of the development environment.

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 view-points.

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.

Errors Browsers/Errors

This tool lists the errors and warnings XPCE may raise. It may be used to examine an error and to change its ←kind, i.e., how the system reacts if the error occurs.

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.

GUI preferences Tools/Class Variables

This tool allows the user to pick an object from the screen and edit the class variables (see section 11) that determine its appearance. Changes apply immediately and may be saved to the user's *Defaults* file.

The manual itself (help) File/Help

The manual tools are documented by itself. Most tools have 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/swipl/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 14.2.2.
- ⟨*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:

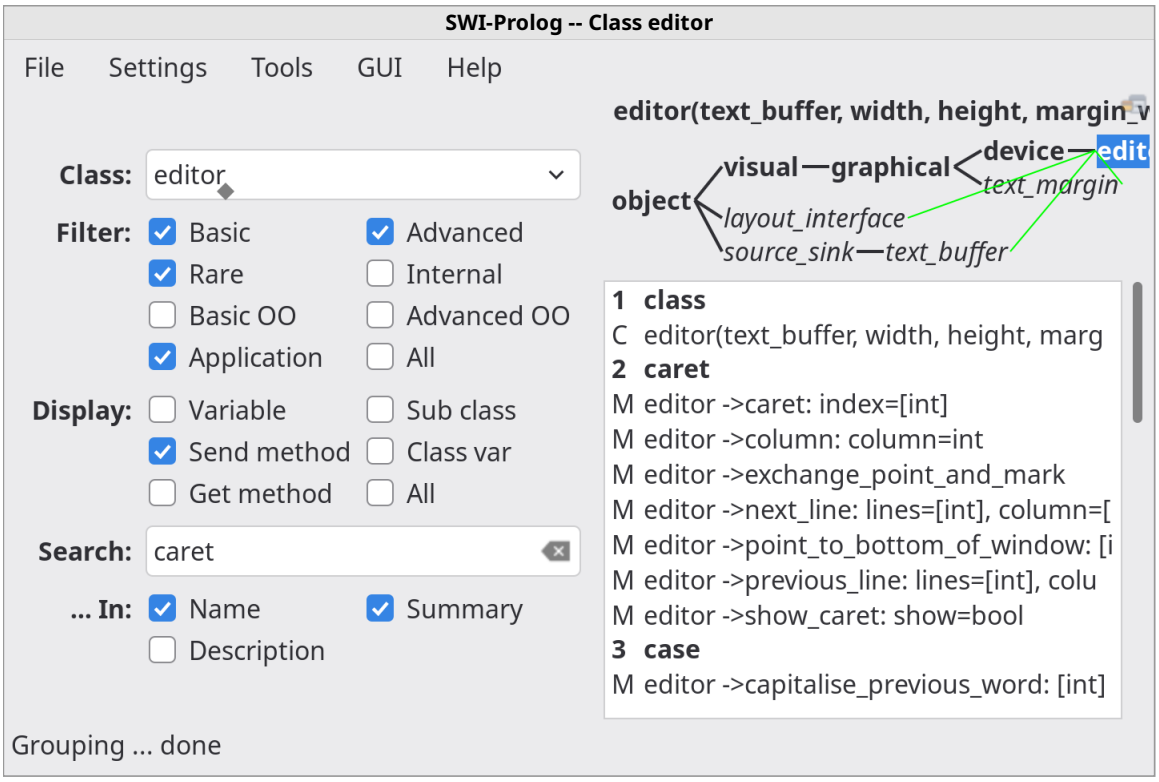


Figure 3.1: The Class Browser

```
[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

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. Clicking a class selects or deselects it, after which the class-browser immediately searches 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. The class itself is always shown at the top. If an item is tagged with an open book, the reference manual provides a description that may be read 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 9.
- 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. 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 a match for the specified regular expression in one of the specified fields. The list is updated while typing. Searching is case sensitive. Searching in the Name field is useful to find a particular method if the name (or part of it) is known.

Changes to any of the controllers of the dialog are applied immediately.

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 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, **...** **In** 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(mono, normal, 13)).
X = @mono_normal_13
2 ?- new(Y, font(mono, normal, 13)).
Y = @mono_normal_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 `cursor`.

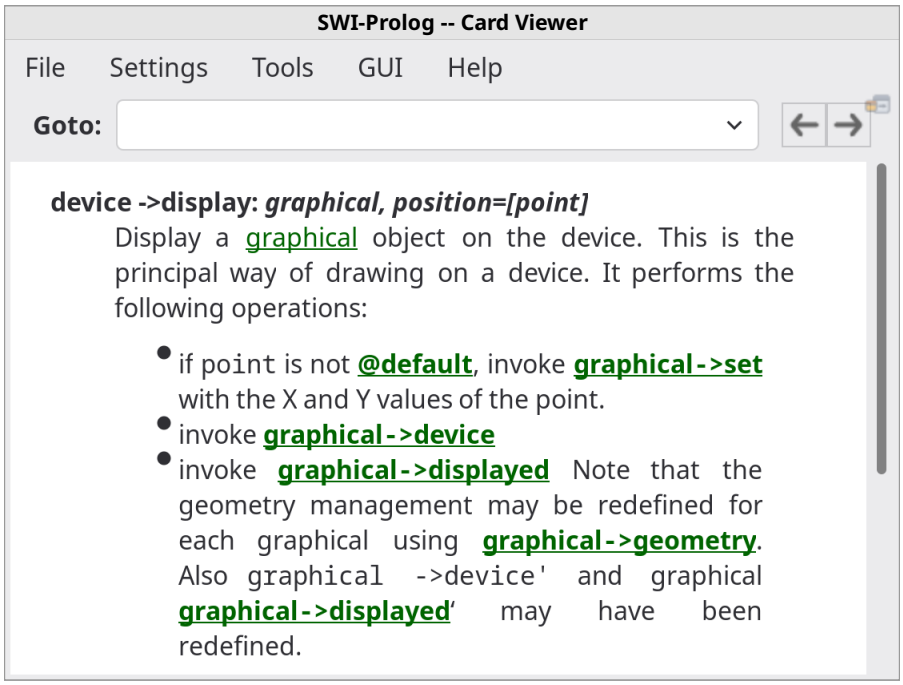


Figure 3.2: The Card Viewer

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

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 an open book 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.

The "Card Viewer" displays the description of the displayed object (method, variable, class, ...) from the reference manual, rendered as HTML. For classes defined in Prolog, the PlDoc comments of the class and its members are shown. References to other manual material are hyper-links: clicking a link jumps to the related material. The arrow buttons navigate back and forward through the history of visited cards. If a method or class variable is described by the instance variable with the same name, the card says so and shows the description of the variable.

The **Goto** field allows for switching to a new card. The syntax for this field is similar to the `manpce/1` and `tracepce/1` predicates description in section 16. It consists of a classname, fol-

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

Typing a class name or a wildcard pattern in the **Class** field expands the tree to show the matching classes and selects them. Figure 3.4 shows the place of the classes `browser` and `list_browser`. The popup menu on a class opens its documentation or the class browser, expands or prunes the tree and, for a user-defined class, shows the source.

[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.

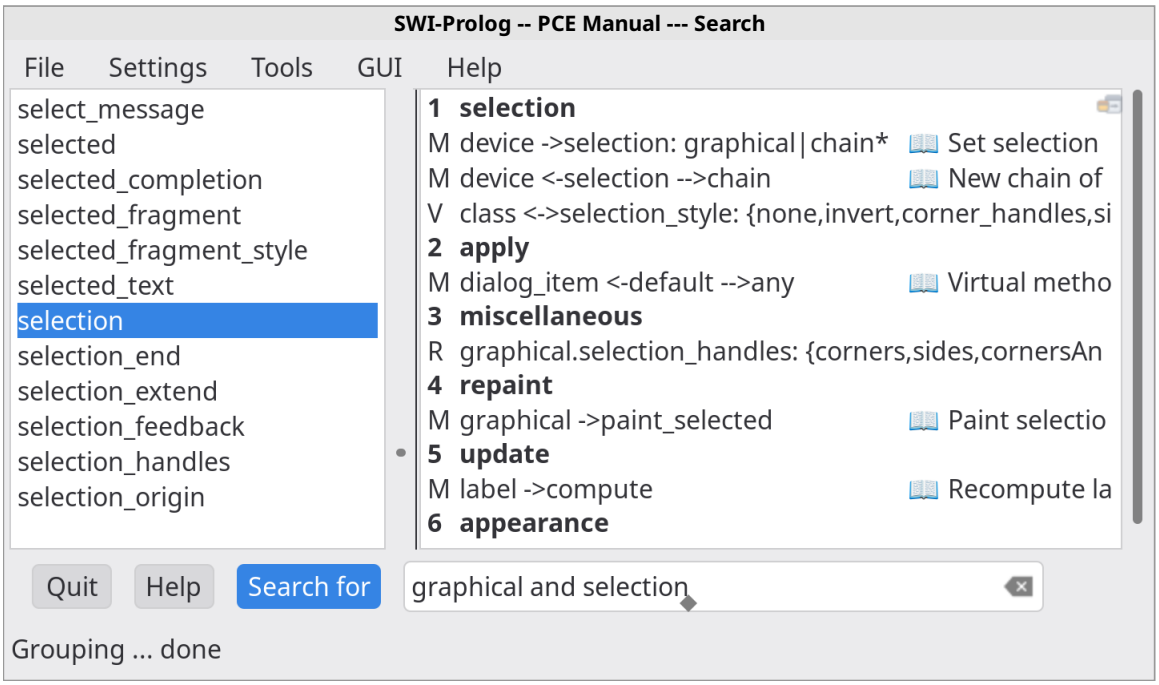


Figure 3.3: Manual search tool

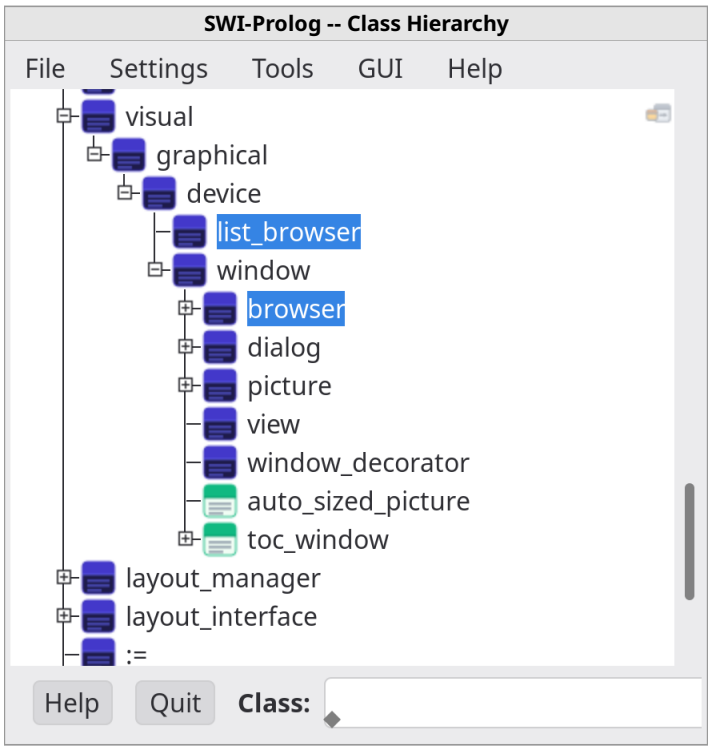


Figure 3.4: Class Hierarchy Tool

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_item`*s* 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.

Dialogs are created using the `new/2` and `send/[2-8]` operations as described in this chapter. Dialog items have a flat look that follows the active theme (see section 14.10.1) and dialogs can be operated completely from the keyboard: TAB and SHIFT-TAB move the keyboard focus, SPACE or RETURN press the focussed button, otherwise RETURN activates the default button, ESCAPE activates a button named `cancel` and typing the underlined letter of a label while holding ALT activates the item. The demo `dialog_gallery` shows the available dialog items and how they are operated from the keyboard. It is started from File/Demo programs in the online manual (see section D.2.3) or using `?- use_module(demo(dialog_gallery)), dialog_gallery.`

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 is shown in figure 4.2.

```
ask_employee :-
        new(Dialog, dialog('Define employee')),
        send_list(Dialog, append,
                  [ new(N1, text_item(first_name)),
                    new(N2, text_item(family_name)),
                    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]),
```

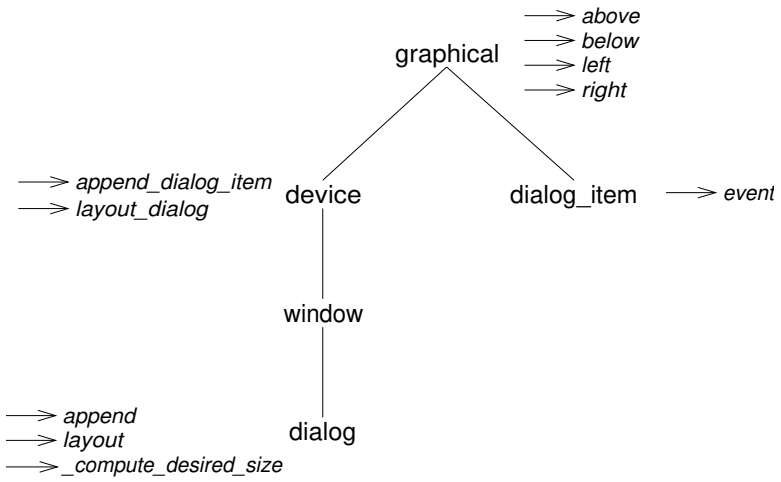


Figure 4.1: Dialog Inheritance Hierarchy

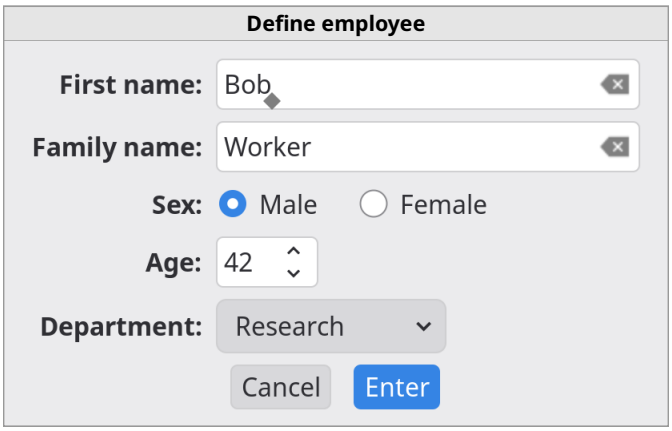


Figure 4.2: Enter employee

```
21          send(Dialog, default_button, enter),
22          send(Dialog, open).
23
24  assert_employee(FirstName, FamilyName, Sex, Age, Depth) :-
25          format('Adding ~w ~w ~w, age ~w, working at ~w~n',
26                 [ 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 14.2 explains the use of `message` objects in detail.

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.

`button`	Simple push-button. Executes ←`message` when pressed. '`button` → `danger`' marks a button that starts a destructive action.
`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, in which case it is shown as a combo-box.
`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. '`menu` → `kind`' selects the presentation: radio-buttons or tick-boxes (`marked`), a segmented control (`choice`), a drop-down list (`cycle`) or a `popup`. '`menu` → `multiple_selection`' allows selecting multiple items.
`bool_item`	A switch to edit a boolean value (@on or @off).
`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`
 	
`tab_stack`
`dialog_group`
	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.
	Stack of `tab` objects.
	Group of dialog items, possible with border and label.

Table 4.1: Built-in dialog items

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 11) 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 → right: 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.

graphical ← 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 `dialog_group` 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),
```

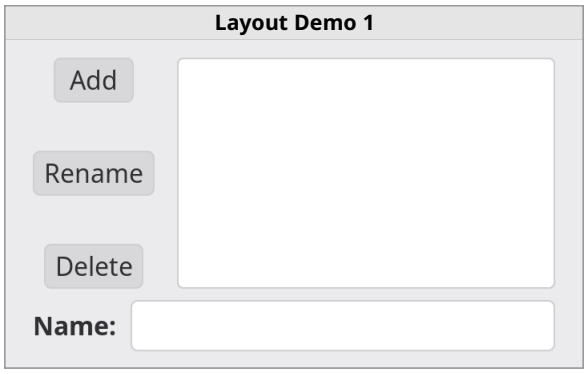


Figure 4.3: Aligning multiple items

```
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).
```

In line 3, a `dialog_group` is added to the dialog to contain the stack of buttons. This group 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 group 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 group. Now we enforce the layout and adjust the bottom and right sides of the list_browser to the group 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`', for example to invoke '`dialog → layout`' with the new size.

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 14.5 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, so the module does not depend on xpce being loaded already.

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

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`.

```
wysiwyg(File) :-
    new(Fr, frame(File)),
    send(Fr, append, new(D, dialog)),
    send(new(V, view), below, D),
    send(new(report_dialog), below, V),
    send(V, font, normal),
    send(D, append,
```

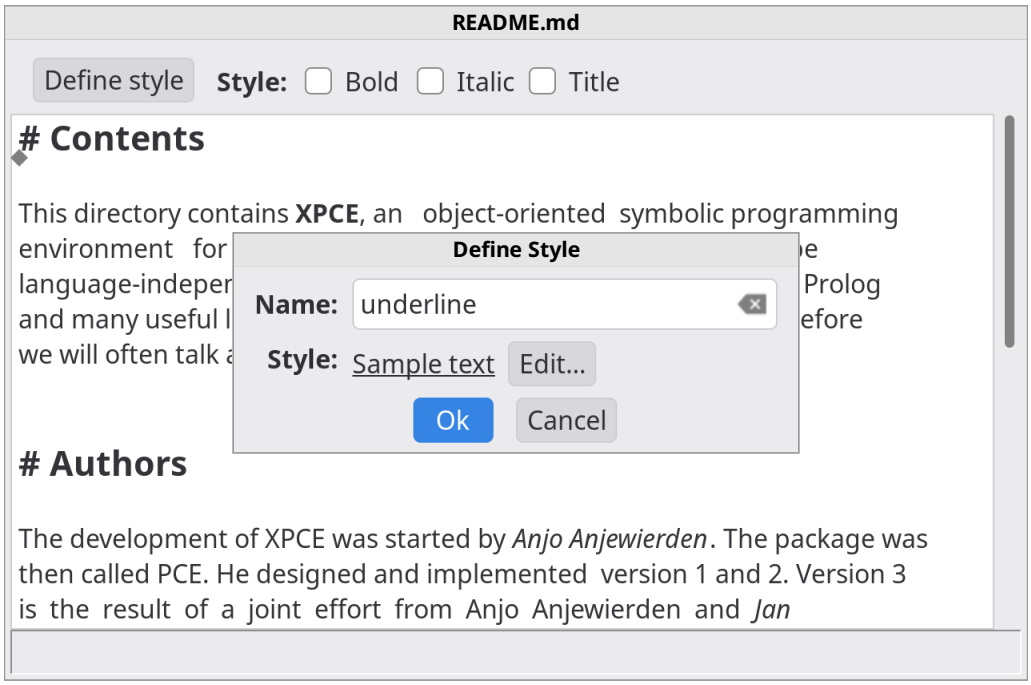


Figure 4.4: Very simple WYSIWYG editor

```
14          button(define_style,
15                 message(@prolog, define_style, Fr))),
16      send(D, append,
17           menu(style, toggle,
18                and(message(@prolog, set_style, Fr, @arg1),
19                    message(@receiver, clear_selection))),
20           right),
21      append_style(Fr, bold,   style(font := bold)),
22      append_style(Fr, italic, style(font := italic)),
23      send(V, load, File),
24      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.

```
25  set_style(Fr, Style) :-
26      get(Fr, member, view, V),
27      get(V, selection, point(Start, End)),
28      (   get(V, mark_status, active)
29      ->  get(V, text_buffer, TB),
30          new(_, fragment(TB, Start, End-Start, Style))
31      ;   send(Fr, report, warning, 'No selection')
32      ).
```

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

```
33  define_style(Fr) :-
34      ask_style(Fr, Name, Style),
```

```
35      append_style(Fr, Name, Style).
36  append_style(Fr, Name, Style) :-
37      get(Fr, member, dialog, D),
38      get(D, member, style, Menu),
39      send(Menu, append, Name),
40      send(Menu, active, @on),
41      get(Fr, member, view, View),
42      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.

```
43  ask_style(Fr, Name, Style) :-
44      new(D, dialog('Define Style')),
45      send(D, append,
46           new(N, text_item(name, ''))),
47      send(D, append,
48           new(S, style_item(style))),
49      send(D, append,
50           button(ok, message(D, return, ok))),
51      send(D, append,
52           button(cancel, message(D, return, cancel))),
53      send(D, default_button, ok),
54      send(D, transient_for, Fr),
55      repeat,
56      get(D, confirm_centered, Fr?area?center, Answer),
57      (   Answer == ok
58      ->  get(N, selection, Name),
59          (   Name == ''
60          ->  send(D, report, error,
61                   'Please enter a name'),
62              fail
63          ;   !,
64              get(S, selection, Style),
65              send(Style, lock_object, @on),
66              send(D, destroy)
67          )
68      ;   !,
69          send(D, destroy),
70          fail
71      ).
```

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, 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,
```

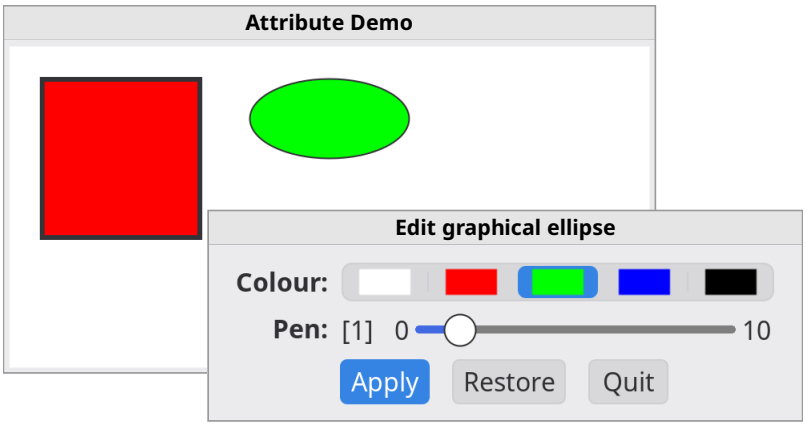


Figure 4.5: Attribute editor for graphical objects

```
16                          message(Gr, fill, @arg1)))),
17          send(M, layout, horizontal),
18          forall(colour(C), append_colour(M, C)),
19          send(M, default, Gr?fill),
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, 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).
```

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.png')), 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, colour(orange)).
```

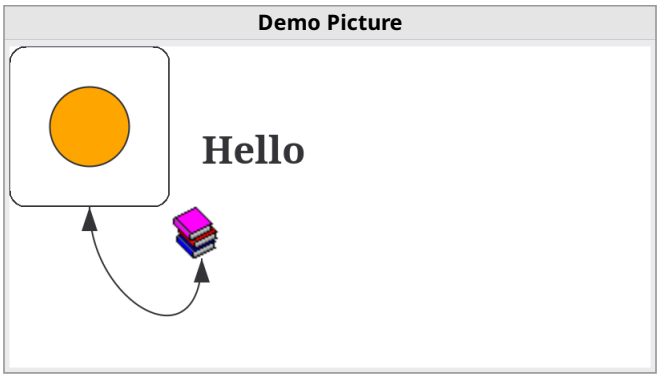


Figure 5.1: Example graphics

```
9  ?- send(@tx, font, font(times, bold, 18)).
10 ?- send(@bz, arrows, both).
```

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.

```
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 '`window → flush`' on the window displaying the graphical to explicitly force redraw right now:

```
square_to_circle(Box) :-
        get(Box, height, H),
        MaxRadius is H // 2,
        get(Box, window, Window),
        forall(between(0, MaxRadius, Radius),
               (   send(Box, radius, Radius),
                   send(Window, flush),
                   sleep(0.02)
               )).
```

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.

arrow	Arrow-head. Normally used implicitly by class line.
bezier_curve	Bezier curve. Both quadratic and cubic Bezier curves are supported.
bitmap	Visualisation of an image. Images may be loaded from various formats, including PNG, JPEG and SVG, and can have an alpha channel (transparency). See section 14.10.
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

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 table	A format object specifies a two-dimensional table layout. Formats may be associated to graphical devices using 'device → format'. The successor of format realises tabular layout compatible to the HTML-3 model for tables. See section 15.5.

Table 5.2: Compound graphical classes

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.png')),
     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 16.4. The classes related to compound graphical objects are shown in table 5.2.

5.3 Opacity

Every `graphical` has an `←opacity` slot, a number between `0.0` (fully transparent) and `1.0` (fully opaque, the default). Values outside that range are clamped. On a primitive such as a `box` or `text`, opacity blends the shape against whatever is behind it; on a `device` or `figure` the children are composited fully opaque first and the whole group is then blended, so overlapping children do not darken each other. This is different from giving each child its own opacity, which makes overlaps visible. Per-pixel alpha in a `colour` (the alpha component of an RGBA specification) modulates only that colour; `←opacity` modulates everything the graphical and its children draw. The `Opacity` demo in the demo browser shows both modes side by side.

5.4 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.

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 14.3. 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`.

5.5 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:

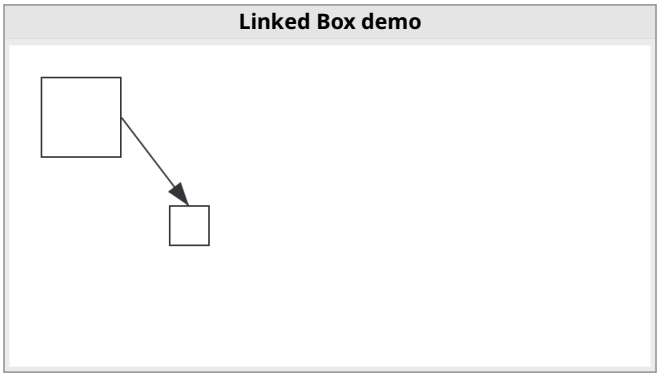


Figure 5.2: A connection between two boxes

`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 5.6) used to connect two graphical objects.

Table 5.3: Classes used to define connections

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

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 9.

5.6 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 15.8. A pen or a finger on a touch screen generates the same mouse events as the mouse, so these gestures also work with a pen or touch screen. Chapter 7 describes how to use the additional information of these devices, such as the pressure of a pen.

5.7 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.

`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

6 Paths, paint and text

chapter 5 introduced the primitive graphicals such as `box`, `circle` and `text`. XPCE draws using the Cairo graphics library and Pango for text. This chapter describes the parts of these libraries that XPCE makes available: `path` objects made of curves and arcs, paint made of colours with transparency, gradients and images, line styles, text formatted using markup, clipping, blending and a `canvas` to draw on using Cairo's drawing model.

The examples of this chapter are in the file `examples/drawing.pl` of the User Guide. Each example displays its graphicals on a device, normally a picture:

```
?- [examples/drawing].
?- new(P, picture), send(P, open), paths_example(P).
```

The demo browser (`?- manpce.`, menu File/Demo programs) holds galleries that show the features of this chapter with their source: the Path gallery, Paint gallery and Text gallery, as well as the Canvas demo.

6.1 Paths

A `path` is a line through a chain of points. The class of each point determines how the line gets there from the previous point. A plain `point` is reached by a straight line. The subclasses of `point` in table 6.1 describe the other segments, using the vocabulary of Cairo and SVG. As they are points, everything that works on the points of a path, such as '`path` → `append`', moving and resizing the path, keeps working. The control points of curves and the ellipses of arcs move and scale with the path.

The example below creates the paths of figure 6.1: a curve with an arrow, an arch, a heart and a ring. The arrows of a path follow the direction of the curve at its ends.

`point`	A straight line to the point.
`curve_point`	A quadratic Bezier curve (one control point) or a cubic Bezier curve (two control points) to the point.
`arc_point`	An elliptical arc to the point. As in SVG, the arc is defined by the radii and rotation of the ellipse and two flags that select one of the four arcs through the two points.
`move_point`	No line: start a new sub-path at the point.

Table 6.1: Points that define the segments of a path

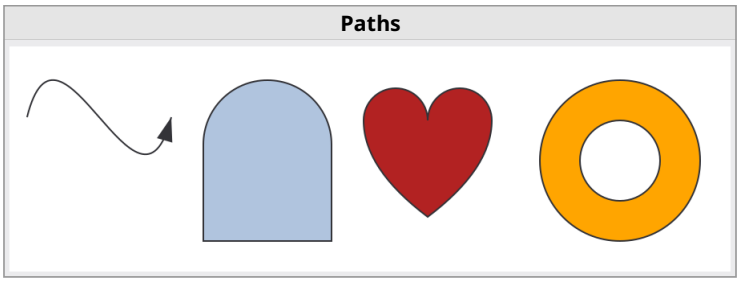


Figure 6.1: Paths with curves, arcs and a hole

```
paths_example(P) :-
    new(Curve, path),
    send_list(Curve, append,
              [ point(0, 80),
                curve_point(90, 80, point(20, 0), point(70, 160))
              ]),
    send(Curve, arrows, second),
    send(P, display, Curve, point(10, 20)),

    new(Arch, path),
    send_list(Arch, append,
              [ point(0, 100), point(0, 40),
                arc_point(80, 40, 40),
                point(80, 100)
              ]),
    send(Arch, closed, @on),
    send(Arch, fill, colour(light_steel_blue)),
    send(P, display, Arch, point(120, 20)),
    ...
```

Paths are often available in the notation of SVG, for example from an icon. ‘`path → svg_data`’ sets the points of a path from the `d` attribute of an SVG ⟨*path*⟩ element and ‘`path ← svg_data`’ returns them in this notation. All SVG path commands are supported. The heart of figure 6.1 is created using

```
    new(Heart, path),
    send(Heart, svg_data,
         "M 10,30 A 20,20 0,0,1 50,30 A 20,20 0,0,1 90,30 \c
          Q 90,60 50,90 Q 10,60 10,30 z"),
    send(Heart, fill, colour(firebrick)),
```

A `move_point` starts a sub-path. A path whose sub-paths overlap can be filled in two ways, selected using ‘`path → fill_rule`’. Using `winding` (the default), a point is inside if the path goes

around it more often in one direction than in the other. Using `even_odd`, a point is inside if a ray from it crosses the path an odd number of times, so a sub-path inside another one is a hole. The ring of figure 6.1 is made of two circles, each consisting of two arcs:

```
    new(Ring, path),
    send(Ring, svg_data,
         "M 0 50 A 50 50 0 1 1 100 50 A 50 50 0 1 1 0 50 Z \c
          M 25 50 A 25 25 0 1 1 75 50 A 25 25 0 1 1 25 50 Z"),
    send(Ring, fill_rule, even_odd),
    send(Ring, fill, colour(orange)),
```

'`path` → `closed`' closes all sub-paths, unless a `move_point` specifies otherwise. '`path` → `radius`' rounds the corners between straight segments. A path of kind `smooth` draws an interpolated curve through its points and ignores the segment classes. The area of a path, '`path` → `inside`' and selecting a path with the mouse are exact for curves and arcs, and take the shape of the corners and ends of the line into account.

6.2 Paint

The '→`fill`' of a graphical such as a `box`, `ellipse` or `path` and the background of a `figure` may be a `colour`, a `gradient` or an `image`.

Colours may be partially transparent. The alpha value ranges from 0 (transparent) to 255 (opaque) and is given as the fifth argument of a colour created from its components or using the notation `'#RRGGBBAA'`:

```
    send(Circle, fill, colour(@default, 255, 0, 0, 128)),
    send(Circle, fill, colour('#ff000080')),
```

Gradients are linear or radial colour transitions. A linear gradient runs from point *p0* to *p1*. A radial gradient runs from a circle around *p0* with radius *r0* to a circle around *p1* with radius *r1*. The colours are added as stops at a fraction between 0 and 1. '`gradient` ← `extend`' determines what is painted beyond the first and last stop: nothing (`none`), the colour of the nearest stop (`pad`, the default), the gradient repeated (`repeat`) or repeated and mirrored (`reflect`).

The points and radii of a gradient are interpreted according to '`gradient` ← `units`'. If `user` (the default), they are coordinates of the device on which the shape is displayed. The gradient does not move with the shape: moving a shape shows another part of the gradient. If `bounding_box`, they are fractions of the box around the shape: (0,0) is its top-left and (1,1) its bottom-right corner. Such a gradient moves and scales with the shape and can be shared by shapes of any size. The example below fills three boxes of different sizes with the same gradient and uses a radial gradient with a small start circle that is not centred, so the circle looks like a ball. The result is in figure 6.2.

```
paint_example(P) :-
    new(Sky, gradient(linear, point(0,0), point(0,1),
```

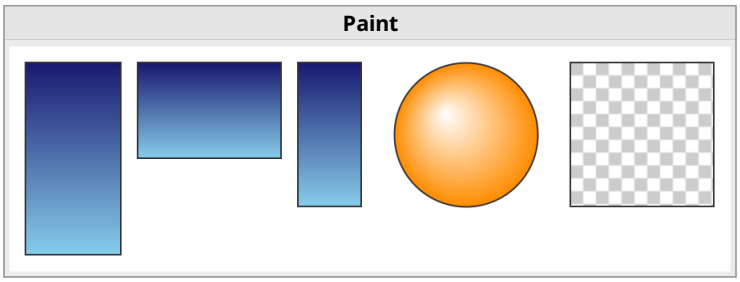


Figure 6.2: A shared gradient, a radial gradient and an image fill

```
                    units := bounding_box)),
    send(Sky, add_stop, 0, colour(midnight_blue)),
    send(Sky, add_stop, 1, colour(sky_blue)),
    forall(member(X-W-H, [10-60-120, 80-90-60, 180-40-90]),
           ( new(B, box(W, H)),
             send(B, fill, Sky),
             send(P, display, B, point(X, 10))
           )),

    new(Glow, gradient(radial, point(0.35,0.35), point(0.5,0.5),
                       0, 0.5, units := bounding_box)),
    send(Glow, add_stop, 0, colour(white)),
    send(Glow, add_stop, 1, colour(dark_orange)),
    send(P, display, new(C, circle(90)), point(240, 10)),
    send(C, fill, Glow),

    checkers(Tile),
    send(P, display, new(T, box(90, 90)), point(350, 10)),
    send(T, fill, Tile),
    send(T, fill_offset, point(0, 0)).
```

Images used as fill are repeated. The pattern starts at the origin of the window, so it does not move with the shape. '`box → fill_offset`' makes the pattern start at the given offset from the box, so it moves along. The image *Tile* above is a 16x16 pixmap with a checkers pattern. See section 14.10 for creating images.

Opacity applies to a graphical as a whole. See section 5.3.

6.3 Line styles

The line of a graphical is drawn using its '`graphical ← pen`' (the width) and '`graphical ← colour`'. '`graphical ← texture`' is the name of a dash pattern such as `dotted` or `dashed`, or a `stroke_style` that also determines the shape of the ends (`butt`,

round or square) and corners (miter, round or bevel) of the line. A stroke_style accepts a vector with the lengths of dashes and gaps and a *paint* that replaces the colour of the line, for example a gradient. The example below creates figure 6.3.

```
lines_example(P) :-
    forall(nth0(I, [miter, round, bevel], Join),
           ( new(Z, path),
             send_list(Z, append,
                       [point(0,40), point(30,0), point(60,40)]),
             send(Z, pen, 10),
             send(Z, texture, stroke_style(butt, Join)),
             X is 20+I*80,
             send(P, display, Z, point(X, 20))
           )),
    forall(nth0(I, [butt, round, square], Cap),
           ( new(L, line(0, 0, 80, 0)),
             send(L, pen, 10),
             send(L, texture, stroke_style(Cap)),
             Y is 90+I*22,
             send(P, display, L, point(20, Y))
           )),
    new(Rainbow, gradient(linear, point(0,0), point(1,0),
                          units := bounding_box)),
    forall(nth0(I, [red, orange, gold, green, blue, purple], C),
           ( F is I/5,
             send(Rainbow, add_stop, F, colour(C))
           )),
    new(S, path),
    send(S, svg_data,
         "M 0 60 C 40 -20 80 140 120 60 S 180 -20 200 60"),
    send(S, pen, 6),
    send(S, texture, stroke_style(round, round, vector(18, 10),
                                  Rainbow)),
    send(P, display, S, point(230, 30)).
```

6.4 Text with markup

Class markup_text displays text formatted using Pango markup. This markup can mix fonts, sizes, styles and colours in one text. The most common tags are <b>, <i>, <u>, <s>, <tt>, <big>, <small>, <sub> and <sup>. The tag accepts attributes such as font, size, foreground and background. The characters <, > and & are written as <, > and &. Text without a colour in the markup is drawn using 'graphical ← colour'.

Pango also does the layout. If a 'markup_text ← width' is given, the lines are wrapped at this width. The lines are aligned according to 'markup_text ← alignment' and may be

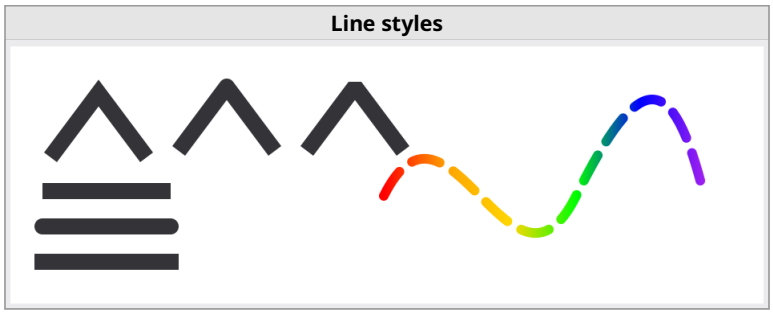


Figure 6.3: Joins, caps and a dashed line painted with a gradient

justified. ‘`markup_text ← ellipsize`’ shortens lines that do not fit by replacing part of the text by ‘...’. Unlike class `text`, a `markup_text` cannot be edited. A `figure` with a transformation rotates or scales the text. The example below creates figure 6.4.

```
markup_example(P) :-
    send(P, display,
         markup_text("Text with <b>bold</b>, <i>italic</i>, \c
                      <span foreground='red'>red</span> and \c
                      <span size='x-large'>large</span> parts"),
         point(10, 10)),
    new(Wrapped,
        markup_text("This paragraph is wrapped at a width \c
                     of 200 pixels and <i>justified</i>, \c
                     so both edges line up.", 200)),
    send(Wrapped, justify, @on),
    send(P, display, Wrapped, point(10, 45)),
    new(Ellipsized,
        markup_text("A line that is too long to fit is \c
                     ellipsized", 160)),
    send(Ellipsized, ellipsize, middle),
    send(P, display, Ellipsized, point(10, 150)),
    new(F, figure),
    send(F, display,
         markup_text("<span size='large'>Rotated \c
                      <b>text</b></span>")),
    send(F, rotate, -25),
    send(P, display, F, point(260, 50)).
```

6.5 Transforming, clipping and blending

A `figure` can do more with its graphicals than display them:

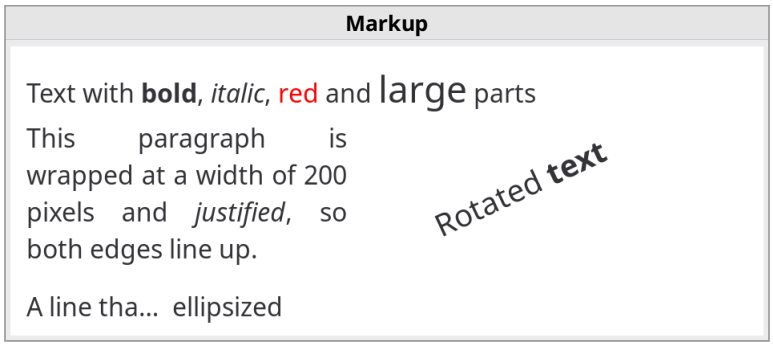


Figure 6.4: Text formatted using Pango markup

- ‘`figure → transform`’ rotates, scales and shears the graphicals. ‘`figure → translate`’, ‘`figure → rotate`’, ‘`figure → scale`’ and ‘`figure → shear`’ modify the transformation. Events and the area of the figure follow the transformation.

- ‘`figure → clip_path`’ clips the graphicals to a path, using its fill rule. The path is in the coordinate system of the graphicals, so it is transformed with them. Clipping only affects drawing: graphicals that are clipped away still receive events.

- ‘`figure → blend`’ determines how the figure is composited onto what is below it. The blend modes are those of Cairo and the CSS property `mix-blend-mode`: `normal`, `multiply`, `screen`, `overlay`, `darken`, `lighten`, `color_dodge`, `color_burn`, `hard_light`, `soft_light`, `difference`, `exclusion`, `hue`, `saturation`, `color`, `luminosity` and `add`.

- ‘`graphical → opacity`’ fades the figure as a whole. See section 5.3.

The example below clips stripes to the shape of a heart and mixes three circles using `multiply`, which combines colours as transparent inks do. The result is in figure 6.5.

```
compose_example(P) :-
    new(F, figure),
    forall(between(0, 9, I),
           ( Colour = colour(@default, 255, I*25, 0),
             new(Stripe, box(110, 10)),
             send(Stripe, pen, 0),
             send(Stripe, fill, Colour),
             Y is I*10,
             send(F, display, Stripe, point(0, Y))
           )),
    new(Heart, path),
    send(Heart, svg_data,
         "M 10,30 A 20,20 0,0,1 50,30 A 20,20 0,0,1 90,30 \c
          Q 90,60 50,90 Q 10,60 10,30 z"),
    send(F, clip_path, Heart),
```

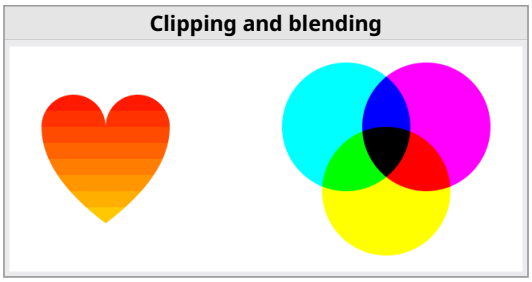


Figure 6.5: A figure clipped to a heart and circles blended using multiply

```
    send(P, display, F, point(10, 20)),

    forall(member(X-Y-Colour,
                  [ 0-0-cyan, 50-0-magenta, 25-40-yellow ]),
           ( new(Fig, figure),
             send(Fig, display, new(C, circle(80))),
             send(C, pen, 0),
             send(C, fill, colour(Colour)),
             send(Fig, blend, multiply),
             PX is 170+X, PY is 10+Y,
             send(P, display, Fig, point(PX, PY))
           )).
```

6.6 The canvas

Graphical objects are the natural way to build a drawing that the user manipulates. Some drawings are computed instead, such as the plot of a function, and are not manipulated afterwards. A `canvas` is a graphical to draw such drawings on using Cairo's drawing model:

1. Build a path using '`canvas → move_to`', '`canvas → line_to`', '`canvas → curve_to`', '`canvas → arc`', '`canvas → rectangle`', etc. '`canvas → path`' adds the points of a `path` object.

2. Draw it using '`canvas → stroke`' or fill it using '`canvas → fill`'. These use the drawing state: the paint set using '`canvas → source`' (a colour, gradient or image), '`canvas → line_width`', '`canvas → stroke_style`', '`canvas → fill_rule`' and the transformation set using '`canvas → translate`', '`canvas → scale`' and '`canvas → rotate`'.

Further, '`canvas → text`' draws markup, '`canvas → image`' draws an image, '`canvas → clip`' restricts drawing to the path and '`canvas → paint`' paints the source everywhere. '`canvas → save_state`' and '`canvas → restore_state`' save and restore the drawing state (Cairo's `save` and `restore`). '`canvas → draw_in`' draws a graphical object on the canvas.

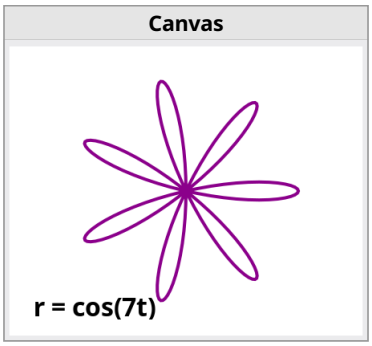


Figure 6.6: A rose curve drawn on a canvas

What is drawn is recorded, so the canvas redraws itself without help of the application. The recording consists of vectors rather than pixels, so a canvas remains sharp when its figure is scaled and when it is saved as SVG or PDF. '`canvas → clear`' erases the drawing. The example below draws a rose curve, shown in figure 6.6. The Canvas demo draws other curves under control of sliders.

```
canvas_example(P) :-
    new(C, canvas(200, 160)),
    send(C, translate, 100, 80),
    rose(C, 7, 70),
    send(C, line_width, 2),
    send(C, source, colour(dark_magenta)),
    send(C, stroke),
    send(C, identity),
    send(C, source, colour(black)),
    send(C, text, "<b>r = cos(7t)</b>", 5, 140),
    send(P, display, C, point(10, 10)).

%   Add the rose curve r = R*cos(K*t) to the path of canvas C.

rose(C, K, R) :-
    forall(between(0, 360, I),
           ( T is I*pi/180,
             Rad is R*cos(K*T),
             X is Rad*cos(T), Y is Rad*sin(T),
             (   I =:= 0
             ->  send(C, move_to, X, Y)
             ;   send(C, line_to, X, Y)
             )
           )),
    send(C, close_path).
```

The alternative to a canvas is a graphical that redefines →`_redraw_area` (see section 14.12).

Such a graphical draws itself each time it must be repainted, using ‘graphical → draw_path’ to draw a path object and ‘graphical → draw_markup’ to draw markup, besides the methods of table 14.5. This is the better choice if the drawing depends on data that changes, so it must be drawn anew anyway.

6.7 Saving drawings

‘graphical → svg’ and ‘graphical → pdf’ save a graphical, including a device or figure with everything on it, as an SVG image or a PDF document. ‘window → svg’ and ‘window → pdf’ save the full contents of a window. As XPCE draws using Cairo for the screen as well as for these files, they look the same as the screen and hold vectors, so they can be scaled without loss. See also section 14.13. ‘display → copy’ puts a graphical on the clipboard as SVG and PNG, so it can be pasted into other applications (see section 7.7.1).

7 Input devices and media

The window system reports more than keys and a mouse: a graphics tablet reports the pressure and tilt of a pen, a touch screen reports several fingers, a trackpad reports precise scrolling and zoom gestures and a game controller has its own buttons and sticks. This chapter describes how XPCE delivers these as events, so applications that only use the mouse keep working, while applications that wish to use the additional information can do so. It also describes the classes for other media and devices: `sound` and `audio_recorder` for audio, `camera` for video, images on the clipboard and `tray_icon` for the system tray.

The examples of this chapter are in the file `examples/devices.pl` of the User Guide.

7.1 Events and input devices

Each `event` refers to the device that generated it using 'event ← device'. This is an `input_device`, whose ←`kind` is one of `mouse`, `pen`, `touch`, `gamepad` or `keyboard`. Events of the mouse refer to @mouse_device and keyboard events to @keyboard_device. A pen, a touch screen or a gamepad is represented by its own `input_device` object, which is created when the window system reports the device. 'display ←`input_devices`' lists the known devices.

Data that is specific to the kind of event is held by 'event ←`detail`'. This is an `event_detail` object or @nil. The event *delegates* to its detail: methods that class `event` does not define are executed on the detail. Thus, the pressure of a pen is requested from the event itself:

```
event(W, Ev:event) :->
    (   send(Ev, is_a, ms_left_drag),
        get(Ev, device, Device),
        get(Device, kind, pen)
    ->  get(Ev, pressure, Pressure),
        ...
    ;   send_super(W, event, Ev)
    ).
```

table 7.1 lists the detail classes. Note that a method of the detail is only defined for the events that have this detail. Calling, e.g., 'event ← pressure' on a keyboard event raises an error, so test the kind of event first, as above.

When a device is added or removed, e.g., a gamepad is plugged in, the window that has the keyboard focus in each frame receives a `device_added` or `device_removed` event, whose 'event ←`device`' is the device concerned.

Releasing a key generates a `key_up` event. It goes to the keyboard focus, like the key-down, and 'event ← key' is the key of the key-down it releases. A key_up is not a keyboard event,

Class	Events and methods
drop_data	drop_* events of files or text dropped from other applications: ←path, ←text, ←source. See section 15.8.
key_data	keyboard events and key_up: ←key and ←repeat. See section 7.1.
wheel_data	wheel and horizontal_wheel: precise scroll distance ←dx and ←dy and ←flipped. See section 7.2.
pen_state	events of a pen: ←pressure, ←xtilt, ←ytilt, ←hover_distance, ←barrel_rotation, ←slider, ←tangential_pressure, ←eraser, ←barrel_buttons. See section 7.3.
touch_point	events of a touch screen: ←finger, ←pressure, ←dx, ←dy. See section 7.3.1.
pinch_data	pinch_* events of a trackpad zoom gesture: ←scale. See section 7.3.1.
gamepad_input	gamepad_* events: ←control, ←value. See section 7.4.

Table 7.1: Classes of 'event ← detail'

so key bindings, text items and editors do not see it; test for it using 'event → is_a: key_up'. 'event ← repeat' is @on for a key-down that is auto-repeated because the key is held. Together, these allow a window to track which keys are down. See key_data.

Each pointing device has its own state. For example, if a pen and the mouse are used at the same time, a click of the mouse between two clicks of the pen does not make the second click of the pen a double click.

7.2 Positions and scrolling

The window system reports positions as floating point numbers. By default, 'event ← x', 'event ← y', 'event ← position', etc. return whole pixels, as XPCE always did. A window with 'window ⇆subpixel_events' set to @on gets the fractional position. This is useful for drawing with a pen or on a high-resolution screen, where the fraction of a pixel matters.

A mouse wheel generates a wheel event for each notch, and tilting the wheel or scrolling sideways on a trackpad a horizontal_wheel event. 'event ← rotation' is 15 for each notch. Trackpads and high-resolution wheels scroll in smaller steps. By default, these steps are added up and an event is generated for each whole notch. Its wheel_data detail holds the precise distance in both directions as 'event ← dx' and 'event ← dy'.

A window with 'window ⇆ smooth_scroll' set to @on receives a wheel event for every step. The ←rotation of these events may be a fraction of a notch. The built-in handlers add up the fractions, so a window scrolls by whole pixels, an editor or list browser by whole lines and a cycle menu by whole notches. Code that handles wheel events itself in a window with smooth scrolling should do the same or use the precise distance, e.g., to zoom smoothly.

7.3 Pens, touch screens and gestures

A pen of a graphics tablet or touch screen acts as a mouse. Its tip generates ms_left_down, ms_left_drag and ms_left_up, the first and second button on its barrel the ms_right_* and

ms_middle_* events and moving the pen while it is close to the surface generates loc_move. Thus, gestures such as click_gesture and move_gesture work with a pen. The pen_state detail of these events holds the state of the pen: the pressure of its tip, its tilt, whether its eraser end is used, etc. A pen also generates the events pen_proximity_in and pen_proximity_out if it comes near or leaves the surface and pen_axis if, e.g., the pressure changes while the pen does not move.

The example below draws with a pen. The width of the line follows the pressure. The mouse, which has no pressure, draws with a width of 2.

```
:- pce_begin_class(sketch_pad, picture, "Draw using a pen").

event(P, Ev:event) :->
    (   send(Ev, is_a, ms_left_down)
    ->  get(Ev, position, P, point(X, Y)),
        send(P, attribute, last, point(X, Y))
    ;   send(Ev, is_a, ms_left_drag),
        get(P, attribute, last, point(X0, Y0))
    ->  get(Ev, position, P, point(X, Y)),
        stroke_width(Ev, W),
        new(L, line(X0, Y0, X, Y)),
        send(L, pen, W),
        send(P, display, L),
        send(P, attribute, last, point(X, Y))
    ;   send_super(P, event, Ev)
    ).

stroke_width(Ev, W) :-
    get(Ev, device, Dev),
    get(Dev, kind, pen),
    !,
    get(Ev, pressure, Pressure),
    W is max(1, round(Pressure*8)).
stroke_width(_, 2).

:- pce_end_class(sketch_pad).
```

7.3.1 Touch screens

The first finger that touches a touch screen also acts as the mouse: it generates ms_left_down, ms_left_drag and ms_left_up. While it is down, other fingers generate touch_down, touch_motion, touch_up and touch_cancel. These are delivered to the window of the first finger. The touch_point detail of all these events holds 'event ← finger', which identifies the finger while it touches the screen, as well as its pressure and motion. If the first finger is lifted while other fingers are still on the screen, no finger acts as the mouse until all fingers are lifted.

Face buttons	south, east, west, north
Other buttons	back, guide, start, left_stick, right_stick, left_shoulder, right_shoulder, dpad_up, dpad_down, dpad_left, dpad_right, misc1 ... misc6, left_paddle1, left_paddle2, right_paddle1, right_paddle2, touchpad
Sticks (-1..1)	left_x, left_y, right_x, right_y; positive is right and down
Triggers (0..1)	left_trigger, right_trigger

Table 7.2: Names of the buttons and axes of a gamepad

7.3.2 Pinch gestures

Moving two fingers apart or together on a trackpad generates pinch_begin, a pinch_update for each step and pinch_end. These are mouse events at the position of the mouse. 'event ← scale' of a pinch_update is the change of scale since the previous update: above 1 zooms in. The recogniser of the library pan_zoom zooms a figure on a pinch as well as on the mouse wheel and pans it when dragged with the left button:

```
?- new(P, picture), send(P, open),
   new(F, figure), send(F, display, box(100, 100)),
   send(F, recogniser, @pan_zoom_recogniser),
   send(P, display, F, point(50, 50)).
```

7.4 Gamepads

'display ← gamepads' enables the support for game controllers and returns a chain holding the connected gamepad objects. Until it is called, XPCE does not use gamepads, as starting the gamepad support may be slow and, on some systems, asks the user for permission. Gamepads that are connected later are added, and the windows with the keyboard focus receive a device_added event (see section 7.1).

As keys, the buttons and sticks of a gamepad generate events for the window with the keyboard focus, and only while the application has the keyboard focus. The events are gamepad_button_down, gamepad_button_up and gamepad_axis. Their gamepad_input detail holds 'event ← control', the name of the button or axis, and 'event ← value': 1 or 0 for a button and the position of an axis. The names follow the position of the controls rather than their label, so they are the same for controllers of different brands: south is the bottom face button, labelled A on an Xbox controller and with a cross on a PlayStation controller. table 7.2 lists the names.

A game typically polls the state of the controller from a timer instead. The same applies to the keyboard: 'keyboard ←key_down' tells whether a key is held down, e.g., get(@keyboard_device, key_down, cursor_left, @on). 'gamepad ← button' and 'gamepad ← axis' return the current state of a button or axis. 'gamepad → rumble'

makes the controller vibrate and 'gamepad → led' sets the colour of its light, if it has these. The example below moves a box using the left stick or the direction pad:

```
gamepad_example :-
    get(@display, gamepads, Pads),
    (   get(Pads, head, Pad)
    ->  true
    ;   print_message(warning, format('No gamepad', [])), fail
    ),
    new(P, picture('Gamepad')),
    send(P, display, new(B, box(20, 20)), point(100, 100)),
    send(B, fill, colour(red)),
    send(P, open),
    new(T, timer(0.02, message(@prolog, move_box, Pad, B))),
    send(P, attribute, timer, T),           % freed with the picture
    send(T, start).

move_box(Pad, B) :-
    get(Pad, axis, left_x, X0),
    get(Pad, axis, left_y, Y0),
    dpad(Pad, dpad_left, dpad_right, X0, X),
    dpad(Pad, dpad_up, dpad_down, Y0, Y),
    DX is round(X*5), DY is round(Y*5),
    send(B, relative_move, point(DX, DY)).

dpad(Pad, Minus, Plus, V0, V) :-
    (   get(Pad, button, Minus, @on) -> V = -1
    ;   get(Pad, button, Plus, @on)  -> V = 1
    ;   V = V0
    ).
```

7.5 Sound

A sound holds audio, loaded from a sound file. If XPCE is built with SDL3_sound, it loads WAV, MP3, Ogg Vorbis, FLAC, AIFF and various other formats. Otherwise it only loads WAV files. 'display ← sound_file_extensions' lists the extensions of the files that can be loaded. 'sound → play' plays it on the default audio output and returns immediately. A sound may be played several times at the same time, e.g., for the sound effects of a game. 'sound → stop' stops all plays of the sound and 'sound ⇆ message' is executed when a play is finished:

```
?- new(S, sound('click.wav')),
   send(S, message, message(@pce, write_ln, done)),
   send(S, play, 0.5).                  % at half volume
```

An audio_recorder records from the default audio input, normally the microphone, into its ←sound. While recording, 'audio_recorder ⇆level_message' is executed about 40 times per

second with the recorder and the input level, a number between 0 and 1. The example below shows the level using a `slider` and saves the recording when the user presses Stop:

```
record_example :-
    new(D, dialog('Recorder')),
    send(D, append, new(Level, slider(level, 0, 100, 0))),
    new(R, audio_recorder(
               message(Level, selection, @arg2*100))),
    send(D, append,
         button(stop, and(message(R, stop),
                          message(R?sound, save, 'recording.wav'),
                          message(D, destroy)))),
    send(D, open),
    send(R, record).
```

The system may ask the user for permission to use the microphone. '`sound ← peaks`' returns the peak level of a given number of parts of the sound, which is useful to draw its waveform. The demo Sound records, plays, opens and streams sound.

7.6 Video from a camera

A `camera` delivers the video of a camera, e.g., a webcam, as an `image`. '`display ← cameras`' lists the names of the available cameras. '`camera → open`' starts the camera. Without arguments, the camera chooses the size of the frames, often its largest. 'camera ←`sizes`' lists the supported sizes and '`camera → open`' accepts the desired size. Each new frame is copied into the same '`camera ← image`', so a `bitmap` that displays this image shows the video, and '`camera ⇆ message`' is executed for each frame, e.g., to analyse it:

```
camera_example :-
    new(C, camera),
    new(P, picture('Camera')),
    send(P, display, bitmap(C?image)),
    send(P, open),
    send(C, open, size(640, 480)),
    send(P, attribute, camera, C).          % freed with the picture
```

The user may have to give the application permission to use the camera. Until this is decided, '`camera ← status`' is `waiting`; it is `open` when the camera delivers frames and `denied` if the user refused. '`camera ⇆ status_message`' is executed when the status changes. The demo Camera of the demo browser (`?- manpce.`, menu File/Demo programs) shows the video, takes snapshots and saves them or copies them to the clipboard.

On Linux, SDL uses the V4L2 camera driver by default. Its PipeWire driver (`SDL_CAMERA_DRIVER=pipewire`) is broken in current SDL versions (up to at least 3.4): the camera stays `waiting` without delivering frames and '`camera → close`' hangs. This also affects sandboxed installations such as Flatpak, which cannot access the V4L2 devices (`/dev/video*`) and must use PipeWire.

7.6.1 Showing video without Cairo

Copying each frame into an `image` converts it to RGB and has Cairo draw it, which is expensive for large frames. If 'camera ⇆`texture`' is an `sdl_texture`, the frames go there instead, keeping the format of the camera. An `sdl_layer` shows the texture in an `sdl_window`. The SDL renderer of the frame then converts and scales the video, on the GPU if it can. The layers are drawn below the graphicals of the window, so these can annotate the video without being redrawn for each frame:

```
sdl_camera_example :-
    new(C, camera),
    send(C, texture, new(T, sdl_texture)),
    new(W, sdl_window('Camera', size(640, 480))),
    send(W, background, black),
    send(W, append, new(L, sdl_layer(T))),  % fit the window
    send(L, flip, horizontal),              % as a mirror
    send(W, display, text('Live'), point(10, 10)),
    send(W, open),
    send(C, open, size(640, 480)),
    send(W, attribute, camera, C).
```

An `sdl_layer` places its texture in an area of the window and can scale, rotate, flip and fade it. '`sdl_texture` ← `image`' converts the pixels to a new `image`, e.g., to save a snapshot, and '`image` → `copy`' copies them into an existing image, e.g., to analyse each frame. An `sdl_texture` can also be made from an image, which allows for fast moving sprites, and '`sdl_texture` → `draw_in`' paints a graphical on its pixels, as '`image` → `draw_in`' does.

7.6.2 Drawing with the GPU

Layers are not limited to video. The renderer draws them without Cairo, so they are suitable for anything that changes on every frame, such as games and animations. A layer becomes a *sprite* using scale `none` and a `transform`: '`sdl_layer` ⇆ `transform`' moves and rotates the texture around its '`sdl_texture` ⇆ `hot_spot`', as '`figure` → `transform`' does for a figure. Creating an `sdl_texture` from a graphical draws it once with Cairo and makes the origin of the graphical the hot spot. Moving many such sprites costs the renderer little, while moving as many figures has Cairo draw each of them for every move. Layers are drawn in the order of '`sdl_window` ← `layers`'; 'sdl_layer →`expose`' and '`sdl_layer` → `hide`' move a layer to the top or the bottom, as for graphicals. The demo Asteroids draws its ship, rocks, bullets and stars as sprites.

A layer can also combine its texture with what is below it in other ways than alpha blending: '`sdl_layer` ⇆ `blend`' `add` makes sparks and light glow, `mod` and `mul` darken. '`sdl_layer` ⇆ `tint`' shows one texture in many colours and 'sdl_layer ⇆`scale_mode`' `nearest` or `pixelart` keeps pixel art sharp. Scale `tile` repeats the texture over the area, e.g., for a background that scrolls by changing '`sdl_layer` ⇆ `offset`', and '`sdl_layer` ⇆ `slices`' scales a texture as nine parts that keep the size of its corners, as is common for the frames of buttons and panels.

An `sdl_mesh` holds triangles whose corners (*vertices*) have a position, a colour and a position in the texture. A layer whose '`sdl_layer` ⇆ `mesh`' is set draws these triangles, interpolating the colours and mapping its texture onto them. This makes gradients, warps a texture,

e.g., a camera image, and draws many particles using a single layer. Small meshes are built using 'sdl_mesh → append' and 'sdl_mesh → triangle'. The predicate below replaces all triangles in one call, which is fast enough to rebuild a mesh of a few thousand triangles on every frame. The demo Particles uses it, and the demo Ripples warps a picture through waves on a grid of 14,000 triangles.

pce_sdl_mesh_set(*+Mesh, +Vertices, +Indices***)**

Replace the vertices and triangles of the sdl_mesh *Mesh*. *Vertices* is a list of terms v(*X,Y*), v(*X,Y,Colour*), v(*X,Y,U,V*) or v(*X,Y,U,V,Colour*). X,Y is the position in the window and U,V the position in the texture in its pixels, which is 0,0 if omitted. Colour is rgba(*R,G,B,A*), where R, G, B and A are integers 0..255, or an integer 0xRRGGBBAA, e.g., 0xFF0000FF is opaque red. The default colour is opaque white. *Indices* is a list of indices in *Vertices*, counting from 0, three for each triangle. If *Indices* is [], each three consecutive vertices make a triangle.

The example draws a rectangle as two triangles with a colour in each corner:

```
gradient_example :-
    new(M, sdl_mesh),
    pce_sdl_mesh_set(M,
                     [ v(0,   0,   0, 0, rgba(255,0,0,255)),
                       v(300, 0,   0, 0, rgba(255,255,0,255)),
                       v(300, 200, 0, 0, rgba(0,0,255,255)),
                       v(0,   200, 0, 0, rgba(0,255,0,255))
                     ],
                     [0,1,2, 0,2,3]),       % two triangles
    new(W, sdl_window('Gradient', size(300, 200))),
    send(W, append, new(L, sdl_layer)),
    send(L, mesh, M),
    send(W, open).
```

An sdl_canvas holds layers as an sdl_window does, but the renderer draws them into a texture of the size of the canvas. A layer shows the canvas as it shows an sdl_texture: this moves or rotates many sprites as one, shows the same layers again at another scale, e.g., as a minimap, or leaves trails if 'sdl_canvas ⇆clear' is @off. The canvas is drawn again only after it changed. The demo Asteroids draws the exhaust trail of the ship this way. The example draws a ring of dots into a canvas and shows it twice: rotated and as a small copy.

```
canvas_example :-
    new(Dot, circle(16)),
    send(Dot, fill, colour(orange)),
    send(Dot, pen, 0),
    new(T, sdl_texture(Dot)),               % hot spot at its top-left
    new(C, sdl_canvas(size(200, 200), colour(navy))),
    forall(between(0, 11, I),               % a ring of 12 dots
           ( A is I*pi/6,
             X is 100+70*cos(A)-8, Y is 100+70*sin(A)-8,
             send(C, append, sdl_layer(T, area(X, Y, 0, 0), none))
```

```
        )),
    new(W, sdl_window('Canvas', size(400, 300))),
    send(W, background, black),
    send(W, append, new(Big, sdl_layer(C, area(40, 40, 220, 220)))),
    send(Big, angle, 15),                    % rotates the ring as one
    send(W, append, sdl_layer(C, area(310, 20, 80, 80))), % a minimap
    send(W, open).
```

7.6.3 Playing and recording video

A `video` plays a video file or stream, e.g., an MP4, MKV or WebM file, with its sound. Its frames go into '`video ← texture`', an `sdl_texture`, which is shown using an `sdl_layer` as above. '`video → play`', '`video → pause`' and '`video → stop`' control it and '`video ⇆ position`' tells and changes the time. The sound plays on the default playback device and keeps the frames in time. The example plays a file in a window and plays or pauses when the window is clicked:

```
video_example(File) :-
    new(V, video(File)),
    new(W, sdl_window('Video', size(640, 360))),
    send(W, background, black),
    send(W, append, sdl_layer(V?texture)),  % fit the window
    send(W, recogniser,                     % click to play or pause
         click_gesture(left, '', single,
                       if(V?status == playing,
                          message(V, pause),
                          message(V, play)))),
    send(W, open),
    send(W, attribute, video, V),           % freed with the window
    send(V, play).
```

'`video ← image`' copies the current frame or, given a time, the frame at that time, e.g., to make a thumbnail. A `video_writer` writes a video file from images, textures, frames and windows, by default as H.264 in MP4. '`video_writer → append`' adds a frame and 'video_writer →`append_audio`' adds samples to its sound. It can also record: 'video_writer →`record`' appends the image of a window or frame at its frame rate, '`camera ⇆ writer`' makes a camera append its frames and 'audio_recorder →`record`' with a `video_writer` adds the sound of the microphone. The example starts recording a window. '`video_writer → close`' stops recording and finishes the file:

```
record_window_example(Window, File, Writer) :-
    get(Window, size, Size),
    new(Writer, video_writer(File, Size, 25)),
    send(Writer, record, Window).
```

Videos are decoded and encoded using ffmpeg on Linux, AVFoundation on macOS and Media Foundation on Windows. XPCE built with `-DXPCE_VIDEO=ffmpeg` uses ffmpeg on all platforms; it then encodes using the encoders of the operating system. 'display ←`video_backend`' tells which is used, and '`display ← video_decoders`' and '`display ← video_encoders`' list the codecs that are available. The demo Video player plays files and network streams, and the demo Camera records the camera and the microphone or streams them to a URL.

7.6.4 Streaming video

The source of a `video` may be a URL, e.g., `https://...`, an HLS playlist (`.m3u8`), an IP camera (`rtsp://...`) or `udp://...`. A network stream is opened in the background: '`video ← status`' is `opening` until it is open, which 'video ⇆`status_message`' tells. A live stream ('`video ← live`') cannot seek; if playing falls behind, frames are dropped so the delay stays small. '`video → option`' passes options to the backend, e.g., `rtsp_transport tcp` for an IP camera.

A `video` without a source plays the encoded data written to it, e.g., using `pce_open/3`. This plays video that arrives on a Prolog stream, e.g., from an HTTP request or a WebSocket. Formats such as MPEG-TS, WebM and fragmented MP4 play while the data arrives.

The target of a `video_writer` may also be a URL, which streams the video live: `rtmp://...` for streaming services, and `srt://`, `udp://`, `tcp://` or `rtp://` for MPEG-TS. A target ending in `.m3u8` writes HLS segments that a web server can serve to browsers. Finally, the target may be code that receives the encoded data. library(pce_video) uses this to write to a Prolog stream. The example serves a window live over HTTP using the HTTP server of SWI-Prolog. The writer gives the data to a message queue in the GUI thread; the thread of the HTTP request copies it to the client. `video('http://localhost:8080/live.ts')`, VLC or ffplay show it.

```
stream_window(Window, Port) :-
    http_handler(root('live.ts'), send_window(Window), [spawn([])]),
    http_server(http_dispatch, [port(Port)]).

send_window(Window, _Request) :-
    message_queue_create(Queue),
    format('Content-type: video/mp2t~n'),
    format('Transfer-encoding: chunked~n~n'),
    in_pce_thread_sync(start_stream(Window, Queue, Writer)),
    call_cleanup(relay(Queue),
                 in_pce_thread(stop_stream(Writer, Queue))).

start_stream(Window, Queue, Writer) :-
    get(Window, size, Size),
    new(Writer, video_writer(message(@prolog, queue_data,
                                     prolog(Queue), @arg1),
                             Size, 25)),
    send(Writer, format, mpegts),
    send(Writer, live, @on),
    send(Writer, record, Window).
```

```
queue_data(Queue, Data) :-
    get(Data, value, String),
    thread_send_message(Queue, String).

relay(Queue) :-
    thread_get_message(Queue, String),
    write(String),
    flush_output,
    relay(Queue).

stop_stream(Writer, Queue) :-
    send(Writer, close),
    free(Writer),
    message_queue_destroy(Queue).
```

Streaming needs the ffmpeg backend. Media Foundation plays files and progressive downloads over http(s) and AVFoundation plays files, but neither reads written data or writes to anything but a file. See `-DXPCE_VIDEO=ffmpeg` above to stream on Windows and macOS.

7.7 The desktop

This section describes how XPCE cooperates with the desktop and other applications. Drag-and-drop of files and text from other applications is described in section 15.8.

7.7.1 Images on the clipboard

'`display → copy`' puts text on the clipboard. It also accepts an `image`, which other applications can paste as a PNG image, or a `graphical`, which is offered as SVG and as PNG, so applications can paste the format they prefer. Using 'graphical →`svg`' as described in section 6.7, the SVG holds vectors:

```
?- send(@display, copy, Figure).
```

'`display ← selection`' with the target `image` returns an image from a PNG, JPEG or other image format on the clipboard and with the target `svg` the SVG text. '`display ← selection_targets`' returns the MIME types of the data on the clipboard, which allows a paste command to decide what to paste:

```
paste(P) :-
    get(@display, selection_targets, Targets),
    (   send(Targets, member, 'image/png'),
        get(@display, selection, clipboard, image, Image)
    ->  send(P, display, bitmap(Image))
    ;   get(@display, paste, Text)
    ->  send(P, display, text(Text))
    ).
```

7.7.2 The system tray

A `tray_icon` shows an icon in the system tray, also known as the notification area, e.g., for an application that runs in the background. The icon can have a tooltip and a menu, which is built from a `popup`. Selecting an entry of the menu executes the popup as if the item was selected in the popup, with the `tray_icon` as context. Sub-popups become submenus. After changing the popup, use '`tray_icon → update`' to rebuild the menu. Freeing the tray_icon removes the icon.

```
tray_example(Frame) :-
    new(P, popup(tray)),
    send_list(P, append,
              [ menu_item(show, message(Frame, expose)),
                menu_item(quit, message(Frame, destroy))
              ]),
    new(I, image(@nil, 16, 16, pixmap)),
    send(I, fill, colour(steel_blue)),
    new(T, tray_icon(I, 'My application', P)),
    send(Frame, attribute, tray, T).
```

On Linux, the system tray requires the AppIndicator library (libayatana-appindicator) and a desktop that shows such icons.

7.7.3 Power and language

'`display ← power_state`' tells whether the system runs on a battery and whether it is charging. '`display ← battery_percent`' and 'display ←`battery_seconds`' tell how much charge is left. An application may use this, e.g., to do less work while running on a battery. 'display ←`preferred_locales`' returns the languages the user prefers, most preferred first, as names such as `en_US`.

7.8 Testing input without the devices

Tests should not depend on the devices of the machine that runs them. XPCE provides methods for testing that make it process input as if a device generated it:

- '`frame → sdl_test_event`'
  Builds an SDL event of a given type from a `sheet` with the fields of the event and processes it as if SDL reported it. This allows testing mouse, wheel, pen, touch, pinch and keyboard input, e.g.,

  ```
  new(S, sheet), send(S, value, x, 50), send(S, value, y, 50),
  send(S, value, which, 7),                % pen 7
  send(Frame, sdl_test_event, pen_motion, S)
  ```

- '`display → sdl_test_gamepad`'
  Attaches a virtual gamepad and sets the state of its buttons and axes. SDL generates the same

events for it as for a real gamepad. ‘`display ← sdl_test_gamepad`’ returns the rumble and LED colour sent to it.

- ‘`camera → sdl_test_frame`’
  Delivers a frame of a given size and colour, as a camera does.

- ‘`frame ← sdl_test_image`’
  Reads back what the SDL renderer draws for the frame, including the layers of an `sdl_window`. ‘`frame ← image`’ composites the windows using Cairo instead.

- SDL drivers
  SDL selects its drivers from environment variables, which must be set before `pce` is loaded. The `disk` audio driver (`SDL_AUDIO_DRIVER=disk`) plays to and records from files and the `dummy` camera driver (`SDL_CAMERA_DRIVER=dummy`) has no cameras. See also chapter A.

The tests of XPCE in the directory `tests` of the package use these methods and are examples of their use.

8 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.

8.1 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; many methods use the type `num`, which accepts both integers and floats). 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 = <pce>(0x7f8a12c400,person)
```

[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 9.

These two representations have very different properties:

- 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.

8.2 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.

XPCE can also embed arbitrary Prolog data in an object. There are three cases where Prolog data is passed natively embedded in an 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 9) 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)).
<pce>(0x7f8a12c400,chain)
4 ?- send(@m, forward, 3 + 4).
7
5 ?- send(@m, forward, [hello, world]).
<pce>(0x7f8a12c5d0,code_vector)
```

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 object as well: a `chain` or `vector` if the argument type demands so and a `code_vector` otherwise.

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)).
_26380

X = _26380
```

Below is a another realistic example of this misconception.

```
?- new(D, dialog('Bug')),
   send(D, append, button(verbose,
                          message(@prolog, assert,
                                  verbose(on)))),
   send(D, open).
[PCE warning: verbose: Unknown class
        ...
]
```

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

```
make_verbose_dialog :-
        new(D, dialog('Correct')),
        send(D, append,
             button(verbose,
                    message(@prolog, assert,
                            prolog(verbose(on))))),
        send(D, open).
```

8.2.1 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.

9 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.

9.1 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 9.1.

9.1.1 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 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*

[1] XPCE defines three implementation techniques for methods. *C-function pointers* are used for almost all the built-in behaviour. Instances of `host_data` 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 9.1: Skeleton for user-defined classes

will be used, which is normally class `class`. An example of meta-class programming can be found in PceDraw's file `shapes.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 9.5.2.

:- 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 9.5.3.

:- pce_class_directive(+*:Goal*)
Define *Goal* to be a goal that manipulates the class instance directly. See section 9.5.3.

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 9.5.1. The type may be postfixed with `:=` *Value* to specify an initial value. If *Value* can be modified (i.e. is not a `constant`, int or `name`) it is often desirable to use `:=` `new`(*NewTerm*) to force each instance to create its own unique copy of the initial value. *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 9.2.

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 11 for details.

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), "Default size of object").
```

handle(*X, Y, Kind, Name*)
Equivalent to the expression below. See also section 5.4.

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

Syntax details

Table table 9.1 describes the details of the non-terminals in the above skeleton in more detail. The notation is an incomplete BNF notation.

9.2 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),
        send(D, slot, type, Type),
        send(D, display, new(T, text(Type))),
        send(T, name, type),
        ...

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

⟨*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 online manual. $<$ 40 characters, no newlines, Prolog string
⟨*TemplateClass*⟩	::=	⟨*Name*⟩	Import a template class. See section 9.5.2
⟨*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 9.5.1
⟨*PrologBody*⟩			Ordinary Prolog code
⟨*Value*⟩			Initial value for the instance variable. Either a constant (e.g., int, name, bool) or `new`(*Term*) to create a new object for each instance

Table 9.1: Syntax details for User Defined Classes

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.

9.3 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/[2-8]` and `get_super/[3-8]` to `send_class/3` and `get_class/4`.

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/[3-8]` and `get/[4-8]` this is possible for `send_super/2` and `get_super/3`. In addition the old '`object → send_super`' and '`object ← get_super`' are expanded to `send_class/3` and `get_class/4`. 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)
```

9.3.1 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.

9.3.2 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 resize/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 14.12.

9.4 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/[2-8]`). *Default* can also be the term

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

In which case the '`object ← class_variable_value`: ⟨*Name*⟩' from ⟨*Object*⟩ will be used as default value (see chapter 11). *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 class-variable 'volume' if @default is passed as an argument.

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

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

9.5 Advanced topics

9.5.1 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 syntax of the `Defaults` file, 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 9.5.2 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.

9.5.2 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.

```
:- pce_begin_class(coloured_reporter, label,
                   "Coloured reporter label").

report(L, Kind:name, Format:char_array, Args:any ...) :->
        Msg =.. [report, Kind, Format | Args],
        send_super(L, Msg),
        get(L, colour_from_report_category, Kind, Colour),
        send(L, colour, Colour).

colour_from_report_category(L, Kind:name, Colour:colour) :<-
        <left to the user>.

:- 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:

```
:- 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.

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

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

9.5.3 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('gnus->gnats', 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.[2]. 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 saved-state facilities of SWI-Prolog (see `qsave_program/2`) to create fast-starting and 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.

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

10 Using Prolog threads

XPCE 6 and earlier used a single global lock to prevent access from multiple Prolog threads. XPCE itself by default ran in a thread named `pce`. As of version 7, this changed substantially:

- The SDL library restricts many operations to the process *main* thread. The exact restrictions depend on the platform. To avoid problems, we run XPCE in the main thread.
- When SWI-Prolog reads from the terminal in the main thread, the system processes SDL events. This implies that XPCE GUI components are responsive when SWI-Prolog waits for the user to enter queries. XPCE *is not responsive* while Prolog executes a query in this scenario.
- XPCE provides *Epilog*, a XPCE/Prolog class that provides a console (terminal) attached to a Prolog *thread*. Epilog is used by `swipl-win`, which creates an Epilog console that runs the Prolog REPL loop in a separate thread, while the main thread processes SDL events. In this scenario XPCE remains responsive while Prolog executes a query. Multiple Epilog consoles may be opened simultaneously and an Epilog console can be split to accommodate multiple REPL loops, each in their own thread.

XPCE uses a single lock, preventing multiple threads from accessing it concurrently. Where XPCE 6 applied this strict, XPCE 7 releases the lock while XPCE calls out to Prolog. This design is similar to the Python *GIL*. Multiple Prolog threads may interact with XPCE, but only one may actively run XPCE code at any point in time.

XPCE operations that manage windows are handled by SDL and most of them must run in the main thread. Several XPCE methods solve this problem by running the SDL steps in the main thread while allowing non-SDL functionality to proceed in the calling thread. An example is ‘`window → open`‘. Code that manages non-UI related objects can always operate from any thread.

Code that manages *graphical* objects displayed on a window such as boxes, lines, text, etc., can be called from any thread. To understand the consequences, we first present the overall control loop of the main thread.

- while(true)
  - If there are pending changes to graphicals, for each window that has modified content do
    - Use ‘`graphical → compute`‘ to update sizes and layout. While in progress, the XPCE lock is *strict*, i.e., no Prolog thread is allowed to initiate methods on XPCE.
    - Use Cairo and Pango to update the Cairo *surface* acting as backing store for the Window.
    - Use SDL to render the Cairo surface to the SDL window.
  - Wait for max 250ms for an SDL event
  - Process the event

The *Process the event* step invokes methods on XPCE that may in turn call Prolog. The processing may create and destroy windows as well as make arbitrary many changes to the contents of windows. These changes only become visible after processing the event completes.

If a thread other than the main thread modifies graphical objects, these changes will be picked up by the main thread on an event or after the 250ms timeout elapses. This can lead to multiple screen updates for a single batch of modifications as well as delaying updates for up to 250ms.

It is typically good practice to run XPCE code that affects the GUI in the main thread. This can be accomplished using the predicates below. These predicates pose only a small overhead in case the calling thread is the XPCE main thread.

in_pce_thread(*:Goal*) *[det]*

Execute *Goal* in the main (XPCE) thread. This call always succeeds immediately, pushing *Goal* onto the SDL event loop.

in_pce_thread_sync(*:Goal*) *[semidet]*

Execute *Goal* in the main (XPCE) thread. This pushes *Goal* onto the SDL event loop and waits until the event is processed. Possible bindings resulting from executing *Goal* in the XPCE thread are propagated to this predicate. *Goal* is executed as `once/1`. If *Goal* raises an exception, this is propagated.

11 Class Variables

Class variables act as 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 may be overruled using a `Defaults` file. XPCE reads two such files, both of which are optional:

- The site-wide file `Defaults` in the XPCE home directory ('`@pce ← home`'). This file is not part of the distribution.
- The personal file `Defaults` in the XPCE application data directory ('`@pce ← application_data`'), which is read last and thus takes precedence. On Unix systems this is normally `~/.config/swi-prolog/xpce/Defaults`, on MacOS `~/Library/ApplicationSupport/swi-prolog/xpce/Defaults` and on Windows `\%APPDATA\%\bsl{}swi-prolog\bsl{}xpce\bsl{}Defaults`.

An application that must not depend on the preferences of its user can set the Prolog flag `xpce_defaults` to `none` to avoid loading the personal file, or to the name of a file to load instead. The flag must be set before `pce` is loaded, e.g., using `swipl -Dxpce_defaults=none ...`

Class variables are typed and documented. 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. The class variables that apply to a graphical object can be edited interactively using `class_variable_editor/[0,1]` from `pce_class_variable_editor`, which is also available as Settings/Edit GUI preferences in the IDE. This tool allows picking an object from the screen, applies changes immediately to the running application and saves them in the personal `Defaults` file. The appearance of XPCE applications is also determined by the colour *theme*, see section 14.10.1.

11.1 Accessing Class Variables

Class variables define *get*-behaviour and can be accessed using the normal `get/[3-8]` 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 to deal with changes of the class-variable value. pce_image_directory/1 is an example of a predicate modifying the image.path class-variable.

11.2 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 *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 11.1.

11.3 The 'Defaults' file

The file Defaults.user in the XPCE home directory is a commented template for this file. It is copied to the personal Defaults file if this does not yet exist when the user opens it for editing using the Edit defaults file button of the class variable editor.

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/[2-8]. 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*⟩|`''`} `'`

11.4 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/[2-8]`.

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, {novice,advanced,expert}, novice).

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


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

Panes: the windows of the IDE 12

The tools of the SWI-Prolog development environment — the Epilog console, PceEmacs, the source navigator, the thread monitor, the debugger — do not each open a window of their own. They are *panes*, and one window holds as many of them as you care to put in it: side by side, above each other, or in tabs. This chapter describes how such a window is put together, what you can do to it, how the system decides where a new pane goes and which class variables (chapter 11) change the defaults.

None of it is built into the kernel. `tab_frame` defines a tab that tiles a set of windows, `pane_frame` the window that holds such tabs, and `pane_layouts` decides where a new pane belongs. An application of your own can use all three; only section 12.3 and section 12.4 are about the Prolog IDE in particular.

12.1 Anatomy of a window

A window of the IDE is an instance of class `pane_frame`. It holds three things:

- A *menu bar* (class `pane_menu_bar`) along the top.
- A strip of *tabs* below it. Each tab is an instance of class `pane_tab`, a subclass of `tab_frame`. Where an ordinary tab holds exactly one window, a `tab_frame` keeps a whole `tile` hierarchy alive, the way class `frame` does for its members: the windows in it are laid out left/right and above/below and the gaps between them can be dragged. A window showing a single tab shows no tab label at all.
- Optionally a *status bar* at the bottom (class `pane_status_dialog`), which reports messages and, if you ask it to, prompts for values.

The windows tiled by a tab are the panes. Any window can be one: none of the pane protocol is compulsory, as `pane_frame` asks with →`has_send_method` before it sends. What a pane would otherwise have to repeat — reaching its frame and its tab, splitting, opening a new tab or window, closing — is in the `pane` template:

```
:- pce_begin_class(my_pane, window, "A pane of my own").
:- use_class_template(pane).
```

A pane need not be a single window. Class `pane_stack` holds its windows in tabs of its own: `tool_pane` uses one such tab to tile the windows of a tool (the navigator is a tree and a dialog), and `emacs_pane` uses one tab per source it shows. Such a pane is a *group*: '`pane_frame` ← `current_pane`' looks inside it for the window you are really working in and '`pane_frame` ← `pane_group`' is the way back out to the pane the tab tiles.

The menu bar is assembled from two sides. The application fills in the menus that every pane of the window shares (`File`, `Settings`, `Tools`, ...) and the pane in view adds its own; whenever the pane in view changes, the bar is rebuilt. Neither side can know what the other will put on, so the bar is not built in the order the menus arrive: '`pane_menu_bar ← menu_order`' names the menus in the order they belong and →`append` works out from the name which menu the new one has to come in front of. A menu whose name is not in the list takes the place of ⋆, which is where a mode's own menu (`Prolog`, `SGML`, ...) or a tool's ends up.

12.2 Rearranging the panes

Every pane carries a small *grip* (class `split_handle`) in its upper-right corner. It is the one thing a pane of any class is sure to have, and everything you can do to a pane by hand starts there.

- **Drag the grip onto another pane** to put this pane beside that one. Which of the four sides it lands on is decided by the edge of the target the pointer is nearest, and the half of the target that the drop would take is outlined while you drag. On a window system that does not tell a window where it is on the display — Wayland — dragging across windows cannot work, and the grip picks the pane up on a click instead: click it, then click where the pane is to go. The balloon on the grip says which of the two applies.
- **Drag the grip onto the label of a tab** to move the pane into that tab.
- **Click the grip with the right button** for a menu that offers, where each applies: move to a window of its own, move to a tab of its own, move into the tab before or after this one, and close.
- **Drag a gap** between two panes to redistribute the room. A line is drawn in every gap that can be dragged. A layout you set by hand is kept when the window itself is resized: the panes keep their share rather than being put back where the system had them.
- **Double-click a tab label** to rename that tab, drag it to reorder it, click the button on it to close it. Its right-button menu also offers to close the other tabs and to move the tab to a window of its own.

When a pane is dropped on another, the target is divided into a left, a right, a top and a bottom *zone*, and the pointer lands in one of them. The top and bottom zones are weighted by '`tab_frame ← split_bias`': above one they are narrower and the left and right zones wider, on the grounds that a left/right split is the commoner one.

From a program the same moves are '`pane_frame → append_pane`' (a tab of its own), '`pane_frame → split`' (beside a given pane) and '`pane_frame → split_beside`' (beside a set of panes, taking a share of their room).

12.2.1 Keys

The window sees a key before the pane with the keyboard focus does, using the key binding table `pane_frame` (see '`frame → shortcut`'). It binds:

Control-Page-Down, Control-Page-Up Show the next or the previous tab.

Control-X 1 Close the panes beside the one you work in, so it is all the tab shows. A pane inside a group, e.g., a view of PceEmacs, also closes the panes beside the group. A pane that does not agree to be closed without asking is moved into a new tab instead: a terminal running a query or holding a half-typed command, and the console that shows the messages of Prolog.

Control-X 0 Close the pane you work in, unless it is the only one of the window.

F6, Shift-F6, Control-X o Move the keyboard focus to the next or previous pane of the tab in view, from left to right and top to bottom. The panes of a group count, e.g., two views of PceEmacs.

An editor or a terminal handles Control-X itself; their own Control-X 0, Control-X 1 and Control-X o do the same. A pane says so using '`pane → handles_prefix`'. In a dialog, Control-X cuts the selection of the text item that has the keyboard focus, if it has one.

Closing a pane, a tab or the window first asks every pane involved →`can_close`. A pane that does not define it agrees. A terminal asks the user whether to close it if a query is running or a command is being typed, and the console that shows the messages of Prolog does the same, unless closing it ends Prolog. With the argument `@off`, →`can_close` does not ask but just fails.

12.3 Where a new pane goes

When you ask for a tool from the `Tools` menu, ask to see a source with `edit/1` or by clicking a location in a tool, or open a new terminal, the IDE has to find a place for it. The class variable '`prolog_ide ← tool_placement`' decides. It is also in the menu, under Settings/New tools and sources open, so you can try the four answers out before writing one down:

`as_arranged` The default. Read how you have arranged windows holding such panes before, and put the new pane where you would have put it yourself. See section 12.4. When nothing that has been learned has anything to say about this pane, it falls back on `tab`.

`frame` Always in a window of its own.

`tab` Always in a tab of the window you are working in.

`split` Always beside the pane you are working in.

With `split` the side comes from the tool itself: a tool says where it belongs by declaring a `pane_side` class variable of its own. The navigator declares `left`; a tool that says nothing goes `below`. The same answer is used when a pane is moved into the tab beside it. Being a class variable, you can overrule it:

```
prolog_navigator.pane_side:             left
prolog_thread_monitor.pane_side:        right
```

12.4 Learning how you arrange your windows

One setting cannot say much. Whether a new pane wants a window, a tab or a split is only the first of the questions; if it is a split, then beside *which* of the panes already there, on which edge of them and taking how much of their room? `pane_layouts` answers all four by watching how you arrange your own windows.

An *arrangement* is a window written down with the content left out: which kinds of pane are in it, how they are tiled, and their share of the room. This is one:

```
pane_frame([], [tab([], horizontal([ 0.2-prolog_navigator,
                                      0.8-vertical([ 0.7-editor,
                                                     0.3-terminal ]) ]))])
```

Not every window teaches something. A window the IDE placed and you never touched teaches nothing — the system would only be learning back its own guesses — so a window says nothing until the first time you move, split, resize, re-tab or close a pane in it by hand. From then on what is counted is the time an arrangement is *lived in*. Putting a window right takes several steps, and only the state you then work in means anything; each step closes off the one before it, and a state that lasted less than ten seconds is thrown away rather than credited.

What an arrangement has earned decays with a half-life of thirty days, so an arrangement you made once and never went back to fades on its own and a habit that changes re-ranks itself. A priority is thus "seconds of recent use", and the arrangement that answers for a new pane is the one that shares most of what the window will hold, weighted by that.

12.4.1 The store

What has been learned lives in the file `xpce/pane_layouts` of your application configuration directory, beside the `Defaults` file described in chapter 11. It is a log rather than a snapshot: each record credits an arrangement with the seconds it had earned at the moment the record was written, and reading the file plays the log back. Nothing waits for the end of the session, so several instances of the IDE running at once each add what they learn, and a crash costs at most the window that was on the screen. A log that has grown past two hundred records is rewritten as one record per arrangement, which says the same thing.

The file is plain Prolog with a header comment, and it may be edited by hand. A record that means nothing to the system — one written by a later version, or edited into something else — costs itself and no more.

Settings/Forget how I arranged windows throws the lot away; `forget_arrangements/0` does the same from a program.

12.4.2 Saying it yourself

The arrangements the system comes with are ordinary clauses of the multifile predicate `default_arrangement/1` in module `pane_layouts`, credited with a nominal five minutes so that any arrangement you have really worked in outranks them. Your initialisation file, or a project, can add its own:

```
:- multifile pane_layouts:default_arrangement/1.
```

```
pane_layouts:default_arrangement(
    pane_frame([], [tab([], horizontal([ 0.25-prolog_navigator,
                                         0.75-editor ]))])).
```

This says three things at once: that a navigator belongs beside an editor, that it goes on the left, and that it takes a quarter of the width. Ask for a navigator in a window showing an editor and that is what you get.

The store itself can be moved with the multifile hook `arrangements_file/1`, whose first clause wins. A project that wants arrangements of its own, or a test that must not touch yours, says so:

```
:- multifile pane_layouts:arrangements_file/1.
pane_layouts:arrangements_file('/path/of/my/project/layouts').
```

`pane_placement/3` is the question the IDE asks. Given the kind of pane to add and the kinds the window holds now it answers `window`, `tab` or `split(Kinds, Side, Share)`. It answers in kinds rather than in panes, so it is plain Prolog that can be tested without a window on the screen.

12.5 Writing a window down

A whole window — its tabs, the panes in them, how they are tiled, their share of the room and what each pane is showing — can be read as a Prolog term and built back from one. '`pane_frame` ← `pane_term`' writes it and `open_pane_frame/2` makes a window out of it:

```
?- Term = pane_frame([geometry('1200x800+40+40')],
                     [ tab([current(true)],
                           vertical([ 0.7-current(editor([file('foo.pl'),
                                                          line(120)])),
                                      0.3-terminal([profile(shell)]) ]))
                     ]),
   open_pane_frame(Term, _Frame).
```

Everything in the term is optional, so one can be written by hand. Each pane answers for its own part of it: an editor says which sources it shows, a terminal says what it runs. `pane_layouts` strips the term to the kinds of pane and their shares, which is what an arrangement is.

12.6 Class variables

All of the below are class variables, set in the `Defaults` file as described in chapter 11. The file `Defaults.user` that ships with XPCE carries all of them as commented examples. Settings/Edit GUI preferences opens a tool to edit the class variables of the pane in view and saves the changes to your own `Defaults` file.

prolog_ide.tool_placement: {as_arranged,frame,tab,split}
: Where a tool, a source or a terminal you ask for is put. See section 12.3. Default is `as_arranged`.

pane_stack.pane_side: {above,below,left,right}
Which side of what is already there a tool is added on when it is split in. Declared again by the tools that do not want the default, `below`; the navigator declares `left`.

pane_frame.label_format: name*
The title of the window, in which `%s` stands for the label of the pane in view. `@nil` leaves the title to the window manager. Default is `'SWI-Prolog -- %s'`. A pane can overrule it for the window it is in.

pane_frame.focus_on_enter: bool
When `@on`, a pane gets the keyboard focus as soon as the pointer enters it rather than when it is clicked. Default is `@off`.

pane_frame.inactive_opacity: num
Opacity of the panes that have not got the keyboard focus, which is a way to see at a glance which one the keyboard talks to. Default is `1.0`, which fades nothing. A pane that declares an `inactive_opacity` of its own overrules this for itself.

pane_frame.prompt_style: {status_bar,dialog}
Where the window asks for a value: all of them in a dialog (`dialog`, the default) or one at a time on the status bar.

pane_menu_bar.menu_order: chain
The order the pulldown menus appear in; see section 12.1. A menu whose name is not in the chain takes the place of `*`. Note that `GUI` and `*` have to be quoted:

```
pane_menu_bar.menu_order:       [ file, settings, tools, debug, 'GUI',     \
                                  edit, browse, compile, '*', help         \
                                ]
```

pane_popup.accelerator_font: font
Font the accelerator of a menu entry is written in. Default is `small`.

pane_tab.editable_label: bool
Whether a tab can be renamed by double-clicking its label. Default is `@on`.

pane_tab.closable: bool
Whether a tab carries a button to close it. Default is `@on`.

tab.edit_width: int
Room made to type a new tab label in, if the tab strip has it. Default is `200`.

tile.separator_colour: colour*
Colour of the lines drawn in the gaps that can be dragged. Default is the theme colour `ui_tile_separator`; `@nil` draws them in the foreground colour. This also applies to the panes of a frame.

tile.gap_colour: colour
Colour between and around the panes. Default is the theme colour `ui_tile_gap`.

tile.separator_pen: 0..
Thickness of these lines. Default is `1`; `0` draws no line.

tab_frame.tile_border_root: int
Margin between the tiled panes and the edges of the tab. Default is `0`: out to the edges.

tab_frame.split_bias: num
Weight of the top and bottom zones when a pane is dropped on another; see section 12.2. Above one the left and right zones are wider. Default is `2`.

tab_frame.horizontal_resize_cursor: cursor
tab_frame.vertical_resize_cursor: cursor
Cursors shown over a gap that can be dragged. Defaults are `ew_resize` and `ns_resize`.

pane_handle.handle_size: size
Size the picture on the grip is drawn at. Default is `size(16,16)`.

split_handle.grip_image: name
The picture on the grip. Default is `'tool/drag-pane.svg'`.

split_handle_gesture.cursor: [cursor]
Cursor while a pane is dragged. Default is `@default`: a picture of the pane itself.

split_handle_gesture.cursor_size: size
Largest that picture is drawn. Default is `size(96,96)`.

split_handle_gesture.cursor_border: [colour]*
Border around it. `@default` (the default) draws it in the foreground colour, `@nil` draws none.

13 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.png')).

        ...,
        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`.

Images are loaded using the SDL_image library (see section 14.10). Loading from a `resource` object supports the same image formats as loading from a file, including SVG.

14 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.

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

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

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

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

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

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

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

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

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

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

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

14.13 **Printing**
Discusses printing and exporting graphics as PDF.

14.1 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.

14.1.1 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 succeeds in this case.
- *timeout*
  No event arrived within a quarter of a second. The method fails in this case.

Events can only be dispatched by the thread that runs the XPCE event loop (see `pce_thread/1` and section 10).

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

```
:- dynamic
        quit/0.

process_to_quit :-
        repeat,
            send(@display, dispatch),
        quit, !.
```

14.1.2 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.

14.2 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?selection, 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.

14.2.1 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.

14.2.2 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).
```

```
Hello default
Hello world
Hello 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 arithmetic on integers and floating point numbers 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 warning: @default/constant: No implementation for: <-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`.

14.2.3 Example: 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`.

```
1          ...
2          get(Dev?graphicals, find_all,
3              message(@arg1, instance_of, box),
4              Boxes),
5          ...
```

14.3 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 14.3.3 describes the `pce_global/2` directive to solve this problem. We will use an image that is used as the label of several buttons as an example.

14.3.1 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.png')).

        ...,
        send(Button, label, @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.

14.3.2 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.png'))
        ),
        send(Button, label, @stones_image),
        ...
```

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

14.3.3 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.png'))).

        ...,
        send(Button, label, @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`.

14.3.4 Global objects for recognisers

Recogniser objects (see section 5.6) 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).
```

14.4 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 colours. The picture contains a box with a popup-menu that fills the interior of the box with the currently selected colour.

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

```
1  fill(white).
2  fill(light_grey).
3  fill(grey).
4  fill(dark_grey).
5  fill(black).
6
7  make_fill_menu(M) :-
8          new(M, menu(fill, marked)),
9          send(M, layout, horizontal),
10         forall(fill(C),
11                send(M, append, menu_item(C, @default, C))).
```

14.4.1 Global named references

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

```
1  fill_1(P) :-
2          new(D, dialog('Fill 1')),
3          make_fill_menu(@fill_menu),
4          send(D, append, @fill_menu),
5          send(new(P, picture), below, D),
6          send(D, open).
7
8  add_box_1(P) :-
9          send(P, display, new(B, box(100,100)), point(20,20)),
10         send(B, popup, new(Pop, popup)),
11         send(Pop, append,
12              menu_item(fill,
13                        message(B, fill,
14                                @fill_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 icons, 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)).
```

14.4.2 Using the prolog database

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

```
:- dynamic
        fill_menu/1.

fill_2(P) :-
        new(D, dialog('Fill 2')),
        make_fill_menu(M),
        send(D, append, M),
        asserta(fill_menu(M)),
        send(new(P, picture), below, D),
        send(D, open).

add_box_2(P) :-
        send(P, display, new(B, box(100,100)), point(20,20)),
        send(B, popup, new(Pop, popup)),
        fill_menu(M),
        send(Pop, append,
             menu_item(fill,
                       message(B, fill,
                               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_menu/1` predicate then holds an invalid reference.

14.4.3 Using object-level attributes

PCE object-level attributes provide another approach:

```
fill_3(P) :-
        new(D, dialog('Fill 3')),
        make_fill_menu(M),
        send(D, append, M),
        send(new(P, picture), below, D),
        send(P, attribute, fill_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_menu, M),
        send(Pop, append,
             menu_item(fill,
                       message(B, fill,
                               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_menu and this attribute refers to a menu holding fill colours.

14.4.4 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_menu(M),
```

```
        send(D, append, M),
        send(new(P, picture), below, D),
        send(D, open).

:- pce_global(@fill_with_current_colour,
              make_fill_with_current_colour).

make_fill_with_current_colour(G) :-
        Gr = @arg1,
        new(G, popup),
        send(G, append,
             menu_item(fill,
                       message(Gr, fill,
                               ?(?(Gr?frame, member, dialog),
                                 member,
                                 fill)?selection))).

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

```
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' that implements a ←`selection` method yielding a colour.

14.4.5 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_menu(M),
        send(D, append, M),
        send(new(picture), below, D).

current_fill_colour(F, C:colour) :<-
        get(F, member, dialog, D),
        get(D, member, fill, M),
        get(M, selection, C).
```

```
14
15 draw_box(F) :->
16         get(F, member, picture, P),
17         send(P, display, fillbox(100,100), point(20,20)).
18
19 :- pce_end_class.
20
21 :- pce_begin_class(fillbox, box).
22
23 :- pce_global(@fillbox_recogniser, make_fillbox_recogniser).
24 make_fillbox_recogniser(G) :-
25         Gr = @arg1,
26         new(G, popup_gesture(new(P, popup))),
27         send(P, append,
28              menu_item(fill,
29                        message(Gr, fill,
30                                Gr?frame?current_fill_colour))).
31
32 event(B, Ev:event) :->
33         (   send(B, send_super, event, Ev)
34         ;   send(@fillbox_recogniser, event, Ev)
35         ).
36 :- 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_colour, while the application (the frame) hides its internal window organisation using the methods ←current_fill_colour and →draw_box.

14.4.6 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 14.11 form an alternative to relate objects that is suitable if dependent objects cannot rely on each other's existence.

14.5 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.

14.5.1 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 14.4.

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, the 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 14.4.

14.5.2 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 window system by creating the window as a child of the main window, but the support varies between platforms. 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 changes (see '`frame → status`') of the main window are forwarded to the transient windows. If the main window is destroyed, so is the transient window.

14.5.3 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).
```

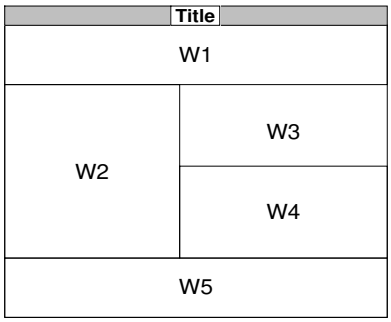


Figure 14.1: Demo window layout

14.6 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 14.1. 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)
```

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)
```

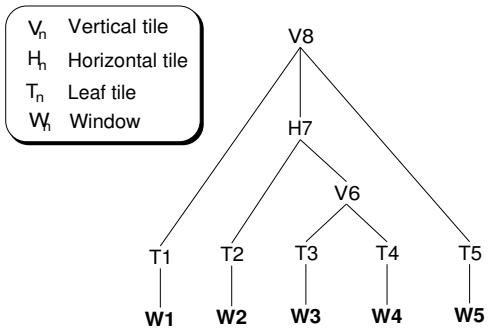


Figure 14.2: Tile hierarchy of example

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 14.1: The window types and their default sizes

Note that 'send(W2, left, W4)' is *exactly* the same as positioning **W2** left of **W3**. The resulting tile hierarchy is shown in figure 14.2. The numbers indicate the order of creation of the various objects.

14.6.1 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 14.1.

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.

14.6.2 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 '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)
        ).
```

14.7 Informing the user

14.7.1 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.

14.7.2 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 14.7.2 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 14.7.2 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`. 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.

14.7.3 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.

14.7.4 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 menu-bar, 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),
4          send(new(picture), right, B),
5          send(new(MD, dialog), above, B),
6          send(new(RD, dialog), below, B),
7          send(Frame, append, B),
8
9          send(MD, append, new(MB, menu_bar)),
10         send(MD, pen, 0),
11         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. The file is selected using '`frame ← open_file`', which uses the file dialog of the operating system.

```
load(Frame) :-
        get(Frame, open_file, 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 14.3.

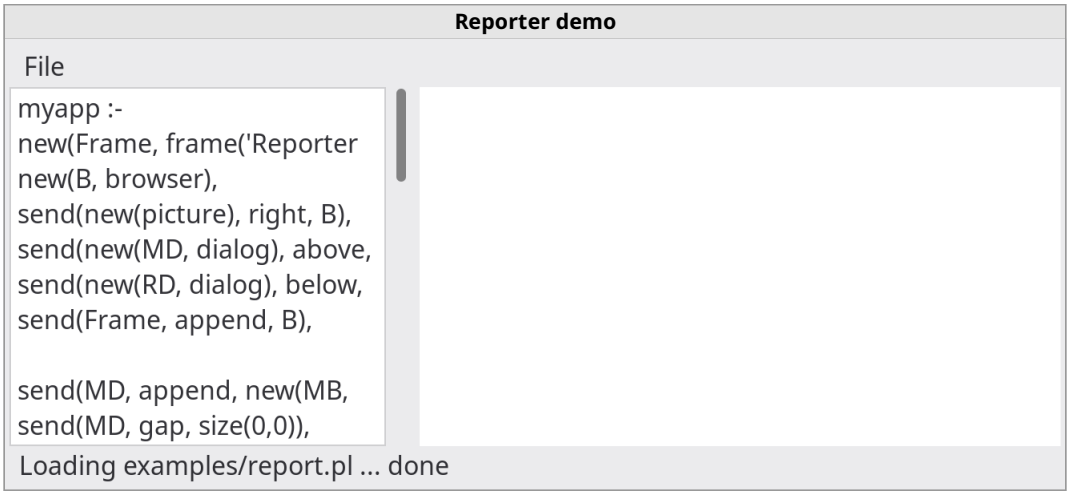


Figure 14.3: The ‘reporter’ demo

14.8 Errors

Errors are abnormalities that are detected during a programs execution. Errors may be caused by 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 normally prints a warning with a backtrace of the XPCE goal stack and the method returns 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 prints the goal in which the error occurred and switches XPCE into debug mode, 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 14.7. 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. Finally, `backtrace` is as `print`, but also prints a backtrace of the XPCE goal stack. Most errors that result from programming errors, such as calling a non-existing method or passing an argument of the wrong type, are warnings with feedback `backtrace`. A backtrace for any error may be forced using e.g., `send(error(no_behaviour), feedback, backtrace)`.

  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.

14.8.1 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`. If 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(file(foo), append, bar), E, true).
E = error(pce(not_open_file, [<pce>(0x13dcdda0,file), write]),
          send(file(foo), append(bar))).
```

14.8.2 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, report)).


        ...,
        (   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.

14.8.3 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 14.8.

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 `pce_renew`.

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

14.9 Specifying fonts

XPCE's font specification is a three-stage process based on the Pango library as well as XPCE's older legacy font handling. An XPCE font is represented by `font`, which is created from four parameters:

- **family** is the logical font family. The typical values are `sans`, `serif` and `mono`. These are mapped to Pango font families based on `font.pango_families`. The default mapping depends on the platform. It may be overruled in the `Defaults` file. Below is an example. See the Pango documentation for details.

```
font.pango_families: [ sans  := "Noto Sans,sans", \
                       serif := "Noto Serif,serif", \
                       mono  := "Noto Sans Mono,monospace" \
                     ]
```

- **style** is one of `normal`, `bold`, `italic` or `oblique`. Note that `bold` is a compatibility shorthand for `normal` with the weight `bold`.
- **size** is the *point-size* of the font.
- **weight** is either a Pango defined symbolic weight or an integer in the range 100...1000. Symbolic names are `thin`, `ultralight`, `light`, `semilight`, `book`, `normal`, `medium`, `bold`, `ultrabold`, `heavy`, or `ultraheavy`

14.9.1 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 class `font` defines a mapping between logical font names and `font` objects. 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(sans, normal, 12)
  The default font. Normally a proportional normal font. Should be easy to read.
- bold font(sans, bold, 12)
  Bold version of the normal font.
- italic font(sans, italic, 12)
  Slanted version of the normal font.
- label font(sans, bold, 12)
  Labels of dialog items, dialog groups, browsers, editors and windows. Change this alias to change the font of labels without affecting bold text, e.g., `label := font(sans, normal, 12)`.
- small font(sans, normal, 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(sans, normal, 14)
  Slightly larger version of the normal font.
- boldlarge font(sans, bold, 14)
  Bold version of large.
- huge font(sans, normal, 18)
  Even larger font. To be used for titles, etc.
- boldhuge font(sans, bold, 18)
  Bold version of huge.
- tt font(mono, normal, 14)
  Terminal font. To be used for code fragments, code editors, etc. Should be easy to read. The old name `fixed` refers to the same font.
- boldtt font(mono, bold, 14)
  Bold terminal font.
- itt font(mono, italic, 14)
  Slanted terminal font.
- bitt font(mono, italic, 14, bold)
  Bold slanted terminal font.

The end-user of an XPCE application can define the class-variable `font.system_fonts` to overrule fonts. Only the aliases that should change need to be mentioned; the others keep their default. The class-variable `font.scale` (default 1.0) scales all fonts, for example on a high-resolution screen. The method `font ← rescale` returns the same font scaled by a factor and the class method `font ← font_families`, called as `get(class(font), font_families, Sheet)`, lists the font families known to Pango.

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),
        ...,
```

14.9.2 Unicode

XPCE fully supports Unicode. Internally, text is stored as one code point per character on all platforms, including Windows, so characters above 0xffff are handled properly. On most platforms Pango supports coloured glyphs, which allows XPCE to use Unicode emoji symbols.

14.10 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.

XPCE loads images using the SDL_image library and draws them using Cairo. As a result, the same image formats are supported on all platforms, including the scalable SVG format. See section 14.10.2. Images can also be copied to and pasted from the clipboard (section 7.7.1) and a `camera` delivers its video as an image (section 7.6).

14.10.1 Colour handling

XPCE uses a simple colour model. A `colour` is a 32-bit RGBA value: 8 bits for each of the red, green and blue channels and 8 bits for the *alpha* channel that defines the opacity, where 255 is fully opaque. The value is available as '`colour ← rgba`'. Where a colour is expected, it may be specified by name, such as `red`, or in hexadecimal notation, `'#RRGGBB'` or `'#RRGGBBAA'`. A colour can also be created from its components, e.g., `colour(@default, 255, 0, 0, 128)`. There are no colour maps: all colours can be used at the same time. Colours cannot be created from images. Use a `gradient` to fill an area with more than one colour, or an `image` to fill it with a repeated pattern. This topic is handled in the section on images for historical reasons.

System colours

Besides the CSS and X11 colour names, XPCE defines colour names that reflect the colours chosen by the user in the desktop settings, such as the light or dark appearance, a high-contrast theme or the accent colour. There are two groups of system colours:

- *Common names*
  The names starting with `sys_` are defined on all platforms. They denote a role, such as the background of a window or the colours of a selection. XPCE's own defaults use these names, and portable applications should use them if they want to follow the desktop settings. Table 14.2 shows how they are mapped on each platform.
- *Platform specific names*
  Colours that only make sense on one platform use a platform prefix: `win_` on Windows, `mac_` on MacOS and `kde_` on KDE. On GNOME there are no platform specific names. These names are *not* defined on the other platforms, so using them makes an application non-portable.

The following notes apply to the mapping:

- *Text selection*
  `sys_text_selection_background` is the background of selected text in, e.g., the editor and the terminal. Selected text keeps its colour, so syntax highlighting remains visible. Except on MacOS it is derived by composing 30% of `sys_accent` over `sys_window_background`.
- *Windows*
  The `win_` names are documented in section B. Windows does not provide the accent colour through GetSysColor(), so `sys_accent` is the same as `sys_selection_background`.

Name	Windows	MacOS
sys_window_background	COLOR_WINDOW	textBackgroundColor
sys_window_foreground	COLOR_WINDOWTEXT	textColor
sys_dialog_background	COLOR_BTNFACE	windowBackgroundColor*
sys_dialog_foreground	COLOR_BTNTEXT	labelColor
sys_button_background	COLOR_BTNFACE	controlColor
sys_button_foreground	COLOR_BTNTEXT	controlTextColor
sys_button_pressed	COLOR_3DLIGHT	selectedControlColor
sys_selection_background	COLOR_HIGHLIGHT	selectedContentBackgroundColor
sys_selection_foreground	COLOR_HIGHLIGHTTEXT	alternateSelectedControlTextCo
sys_text_selection_background	(derived)	selectedTextBackgroundColor
sys_tooltip_background	COLOR_INFOBK	windowBackgroundColor*
sys_tooltip_foreground	COLOR_INFOTEXT	labelColor
sys_inactive	COLOR_GRAYTEXT	disabledControlTextColor
sys_link	COLOR_HOTLIGHT	linkColor
sys_accent	COLOR_HIGHLIGHT	controlAccentColor
sys_separator	COLOR_BTNSHADOW	separatorColor
sys_shadow	COLOR_BTNSHADOW	tertiaryLabelColor

Table 14.2: Mapping of the sys_ colour names. The Windows column names the argument to GetSysColor(), the MacOS column the NSColor class method and the KDE column the group (without Colors:) and key in kdeglobals. The Fallback column is used on other platforms and if the platform does not provide the colour. GNOME is described separately in table 14.3.

Windows dark mode does not change the colours of GetSysColor(). If the user selected dark mode for applications and no contrast theme is active, the sys_ names therefore use a dark palette that follows the Windows 11 dark appearance, while sys_accent and sys_selection_background use the accent colour selected by the user. The win_ names always use GetSysColor().

- *MacOS*
  All mac_ names are derived from an NSColor class method: the method name without the Color suffix, converted to snake case and prefixed with mac_. For example, selectedContentBackgroundColor becomes mac_selected_content_background. The names defined are mac_label, mac_secondary_label, mac_tertiary_label, mac_quaternary_label, mac_text, mac_placeholder_text, mac_selected_text, mac_text_background, mac_selected_text_background, mac_keyboard_focus_indicator, mac_unemphasized_selected_text, mac_unemphasized_selected_text_background, mac_link, mac_separator, mac_selected_content_background, mac_unemphasized_selected_content_background, mac_selected_menu_item_text, mac_grid, mac_header_text, mac_control_accent, mac_control, mac_control_background, mac_control_text, mac_disabled_control_text, mac_selected_control, mac_selected_control_text, mac_alternate_selected_control_text, mac_window_background, mac_window_frame_text, mac_under_page_background, mac_find_highlight, mac_highlight, mac_shadow, the fill colours mac_system_fill, mac_secondary_system_fill, mac_tertiary_system_fill, mac_quaternary_system_fill and mac_quinary_system_fill, and the adaptive colours mac_system_red, mac_system_orange, mac_system_yellow, mac_system_green, mac_system_mint, mac_system_teal, mac_system_cyan, mac_system_blue, mac_system_indigo, mac_system_purple, mac_system_pink, mac_system_brown and mac_system_gray. Colours that are not provided by the running version of MacOS are not defined.

  MacOS colours depend on the appearance (light, dark or increased contrast). They are resolved using the appearance of the application. Many of them are translucent. The mac_ colours are composed over mac_window_background and the sys_ colours over sys_dialog_background, so all system colours are opaque.

  *Up to MacOS 15, windowBackgroundColor is a grey that differs from the content background. Since MacOS 26 both are the same, which would make dialogs indistinguishable from content windows. If they are the same, sys_dialog_background is secondarySystemFillColor composed over windowBackgroundColor (#ebebeb light, #303030 dark). This is about the sidebar colour used on GNOME.

  disabledControlTextColor is translucent and disabled text appears on dialogs as well as on buttons. sys_inactive is this colour composed over either sys_dialog_background or sys_button_background, whichever remains most visible on both.

  MacOS has no tooltip colours. The tooltip colours are therefore the same as the dialog colours.

- *KDE*
  On Unix systems other than MacOS, XPCE uses the KDE colour scheme if the environment variable XDG_CURRENT_DESKTOP contains KDE or KDE_FULL_SESSION is true. The colour scheme is read from the kdeglobals files in the directories of XDG_CONFIG_DIRS (default /etc/xdg) and XDG_CONFIG_HOME (default \Stilde{}/.config), where the latter takes precedence. This does not require the KDE or Qt libraries.

  All colours of the groups [Colors:*Group*], including sub-groups such as [Colors:Header][Inactive], and of the group [WM] are defined as kde_⟨*group*⟩_⟨*key*⟩, converted to snake case. For example, BackgroundNormal in [Colors:View] becomes kde_view_background_normal and activeBackground in [WM] becomes kde_wm_active_background. AccentColor in [General] becomes kde_accent.

  If kdeglobals does not define a colour that is used for a sys_ name, XPCE uses the colour of the default KDE scheme, Breeze Light. If there is no AccentColor, sys_accent is DecorationFocus of [Colors:View]. KDE has no separator and shadow colours. These are computed by mixing sys_dialog_foreground into sys_dialog_background for 20% (separator) and 40% (shadow).

- *GNOME*
  GNOME does not publish its colours. If XDG_CURRENT_DESKTOP contains GNOME (and we are not running under KDE), XPCE uses a built-in copy of the libadwaita palette. It asks the XDG Desktop Portal (namespace org.freedesktop.appearance) for the following settings:

  - color-scheme
    If this is 1 (prefer dark), the dark palette is used. Otherwise the light palette is used.
  - contrast
    If this is 1 (high contrast), the text colours are opaque and the separator and shadow colours are stronger.
  - accent-color
    The accent colour, available since GNOME 47. If it is missing or out of range, XPCE uses the default libadwaita accent, #3584e4. The text on a selection is white, unless the accent colour is light.

  Table 14.3 shows the result. The *ink* is the text colour of libadwaita: rgba(0,0,6,0.8) in the light palette and white in the dark palette. A percentage means that this fraction of the ink is composed over the dialog background, which is how libadwaita defines these colours in its CSS. The libadwaita window background is almost white, which would make dialogs indistinguishable from the content windows. The dialog background is therefore the sidebar background of libadwaita, which it uses for panels next to the content. There are no gnome_ colour names.

  The portal is accessed over D-Bus using GIO. If XPCE was built without GIO, or there is no portal, the light palette with the default accent colour is used.

- *Other platforms*
  On other platforms, such as Linux running Xfce, the system colours have fixed values that reproduce XPCE's traditional look. They do not depend on the desktop settings.

Name	Derived from	Light	Dark
sys_window_background	view background	#ffffff	#1d1d20
sys_window_foreground	ink	#333338	#ffffff
sys_dialog_background	sidebar background	#ebebed	#2e2e32
sys_dialog_foreground	ink	#2f2f34	#ffffff
sys_button_background	10% ink	#d8d8db	#434347
sys_button_foreground	ink	#2f2f34	#ffffff
sys_button_pressed	30% ink	#b3b3b6	#6d6d70
sys_selection_background	accent	#3584e4	#3584e4
sys_selection_foreground	white or black	#ffffff	#ffffff
sys_text_selection_background	30% accent	#c2daf7	#243c5b
sys_tooltip_background	80% #000006	#2f2f34	#09090f
sys_tooltip_foreground	white	#ffffff	#ffffff
sys_inactive	50% ink	#8d8d91	#979799
sys_link	accent	#3584e4	#3584e4
sys_accent	accent	#3584e4	#3584e4
sys_separator	15% ink	#cfcfd1	#4d4d51
sys_shadow	30% ink	#b3b3b6	#6d6d70

Table 14.3: The sys_ colours on GNOME, using the default accent colour and normal contrast.

The system colours are determined when XPCE looks up a colour name for the first time, normally while it starts up. They are reloaded by 'display_manager → system_colours_changed', which also redraws all windows. Named colour objects for the system colours, such as colour(sys_dialog_background), are updated in place, so all graphicals that use them get the new colour. XPCE sends this message to @display_manager if it is told that the desktop settings changed:

- *All platforms*
  If the user switches between light and dark mode.

- *Windows*
  Also if the user selects another contrast theme or accent colour.

- *MacOS*
  Also if the user changes the accent colour, the highlight colour or the contrast.

Other changes, such as a new accent colour on KDE or GNOME or another KDE colour scheme with the same brightness, are not noticed. The application may send →system_colours_changed itself, or XPCE must be restarted.

After reloading the system colours, the theme colours that are derived from them are resolved again (see section 14.10.1). Next, →system_colours_changed sends 'display_manager ←system_colours_message' if this is not @nil, which allows the application to select another theme. Finally, all windows are redrawn.

Themes

A *theme* defines the colours of the XPCE user interface and the SWI-Prolog development tools, such as the colours for syntax highlighting in PceEmacs. XPCE itself only uses *theme colours*: colours whose name describes their role rather than their value. For example, the background of windows is `ui_window_background` and the colour of a comment in PceEmacs is `syntax_comment`. A theme maps these names to values. Switching themes changes the values of the theme colours in place and redraws all windows, so themes can be switched while XPCE is running.

The library `pce_theme` manages themes. The `light` theme is the default. The other themes are files in `library(theme)` that define the colours for their theme, such as `library(theme/dark)`.

Selecting a theme By default, XPCE follows the light or dark setting of the desktop (see '`display ← theme`') and switches theme if the user changes this setting. This is overruled by, in this order:

- *The Settings/Theme menu*
  The *Settings* menu of the IDE tools, such as `swipl-win`, has a *Theme* submenu to follow the desktop or use one of the available themes. This calls `select_theme/1` and holds for the running session.
- *The Prolog flag* `theme`
  For example, `swipl -Dtheme=dark`.
- *The class variable* `display.theme`
  Setting this in the Defaults file (see section 11) makes the choice permanent:

```
display.theme: dark
```

The theme editor The theme editor creates and changes themes interactively. It is started from *Settings/Theme/Edit theme* ... of the IDE tools or using `theme_editor/0` or `theme_editor/1` from library `pce_theme_editor`. Without argument, it edits the active theme. If this is a theme of the user, a file in the user configuration directory, the editor edits it. Otherwise, e.g., for the `light` or `dark` theme that come with SWI-Prolog, it asks for the name of a new theme that is based on the active theme (see section 14.10.1). *New theme* ... creates a new theme based on any of the available themes.

While editing, the theme exists in memory and is the active theme, so all windows show the effect of every change immediately. The theme only becomes a file when it is saved. The editor shows all *settings* a theme can have in a list on the left:

- *Colours*
  The theme colours (see below), grouped as *User interface*, *Terminal colours*, *Development tools* and *PceEmacs syntax*.
- *Console*
  The colours and style of the messages of the Prolog console by message class, as defined by prolog:console_color/3.

- *Online help*
  The styles of the online help (`help/1`) in the console, as defined by the hook pldoc_style:theme/4.

The menus above the list select a group and the settings that this theme changes (*Changed*) or that have an issue (*Issues*). The field next to them filters the list using a regular expression on the name and description. The status column shows a pencil for a setting that the theme changes and a symbol for the most severe issue of a colour: an error, a warning or information, e.g., the theme defines a colour to the value it would have anyway. The tooltip of a row explains its status.

Selecting a setting shows it on the right. For a colour, these are:

- *Value*
  The *Theme* ... button selects another theme colour, such that the colour follows that colour. *Colour* ... opens the colour editor, which shows a colour wheel, sliders for hue, saturation and value and for red, green and blue, and the named colours closest to the current colour. Clicking a named colour uses it. Every change is shown in all windows immediately. *Ok* keeps the colour and *Cancel* restores the setting as it was. *Use inherited value* removes the setting from the theme, such that it uses the value of its base theme.

- *Derivation*
  How the colour gets its value: the steps from the theme colour through other theme colours to a system colour or an RGB value, and the theme that defines each step. Theme colours in the steps are links that select that colour.

- *Used for*
  Samples of the user interface that use the colour, such as a button, a selected item of a list, a few lines of Prolog code with a syntax colour or a terminal with a hyperlink, and the class variables whose default refers to the colour. Documented class variables link to their description. The samples change with the colour.

The style of a console message class is edited using menus for the foreground and background colour and toggles for bold, italic, underline and faint, or by typing the list of attributes of `ansi_format/3`. *Other* ... in the colour menus opens the colour editor with the 256 colours of the terminal: selecting one of these uses `fg8`(*N*) or `bg8`(*N*), any other colour uses the 24-bit colour. The styles of the online help are edited the same way, but use the CSS properties of pldoc_style:theme/4.

Save writes the theme to `lib/theme/`*Name*`.pl` in the user configuration directory (see the Prolog file search path `user_app_config`), which makes it available as any other theme. The file only defines the settings the theme changes and declares its base using prolog:theme_base/2. If the file exists, only the values that changed are replaced and new settings are added after a similar one, such that comments and the layout of the file are preserved. *Save as* ... saves the theme under another name, *Revert* undoes the changes since it was saved and *Edit theme file* opens the file in the editor. Closing the editor with unsaved changes asks to save them. To use a saved theme in a new session, select it using the *Settings/Theme* menu, `swipl -Dtheme=`*Name* or `display.theme` as described above.

Theme colours A theme colour is an instance of class `theme_colour`, a subclass of `colour`. It is created from a name that describes its role and the colour it is derived from, e.g., `theme_colour(syntax_comment, dark_green)`. This is the name of another colour, which

may be another theme colour or a system colour, or a colour object. The RGB value is computed when it is needed by following 'theme_colour ← derived_from' through other theme colours. Changing what a theme colour is derived from using 'theme_colour →derived_from', or creating it again, makes all theme colours compute their RGB value again on their next use. Theme colours may therefore refer to each other in any order. Theme colours are locked, so they are never garbage collected, and they are never returned when looking up a colour from its RGB values. After changing theme colours, 'display_manager → colours_changed' redraws all windows. As graphicals refer to the colour object, either directly or by name, they show the new value after the windows are redrawn.

The theme colours are named using a prefix that describes where they are used:

- ui_role
  The basic colours of the user interface. For each system colour sys_*role* (see table 14.2) there is a theme colour ui_*role*, e.g., ui_window_background. XPCE only uses the ui_ names, such that a theme can replace the system colours (see below). We call these the *roles*. Other ui_ colours are used for, e.g., incremental search (ui_isearch_background). The ui_ colours and their values in the light theme are in the hash table @theme_colour_defaults.

- ansi_colour
  The 16 ANSI colours of the terminal, from ansi_black to ansi_bright_white.

- syntax_class
  The colours of PceEmacs syntax highlighting. The name is derived from the syntax class of library(prolog_colour): syntax_ followed by the name and the atomic arguments of the class, separated by _. For example, the class goal(built_in,_) uses syntax_goal_built_in. A background colour gets the suffix _bg. The value in the light theme is the colour that is defined for the class.

- debug_, prof_, xref_, ...
  The colours of the graphical debugger, the profiler, the cross referencer and other tools.

Using the system colours The roles, the ui_*role* colours that have a system colour sys_*role*, are the system colours if the theme *matches* the desktop. This preserves the look of the desktop, including colours the theme defines for the role. If the theme does not match, for example when using the dark theme on a light desktop, the theme's own colours replace the system colours. A role the theme does not define remains the system colour. All other theme colours, such as ui_isearch_background, ansi_, syntax_ and the colours of the tools, always come from the theme.

The theme matches the desktop if both are light or both are dark. The theme is dark if its ui_window_background is dark and the desktop is dark if sys_window_background is dark. Only the brightness is compared. A theme that does not define ui_window_background matches any desktop. The light theme provides its own light colours for use on a dark desktop.

Writing a theme A theme is a Prolog file library(theme/*Name*) that defines the values of the theme colours using the multifile predicate pce_theme:colour/3. It does not need to load XPCE/Prolog. The library library(theme/dark) is a complete example. The skeleton is below.

```
:- module(prolog_theme_mytheme, []).
:- multifile
```

```
    prolog:theme/1,
    pce_theme:colour/3.

prolog:theme(mytheme).

pce_theme:colour(mytheme, Name, Value) :-
    colour(Name, Value).

colour(ui_window_background, '#1e1e1e').
colour(ui_window_foreground, white).
...
colour(syntax_comment,       green).
...
```

A theme should define all theme colours. Colours it does not define use their value in the `light` theme, except for the roles: these remain the system colour, also if the theme does not match the desktop. A dark theme that does not define, e.g., `ui_tooltip_background` thus uses the light tooltip colour of a light desktop. The theme colours of the roles are only used if the theme does not match the desktop. The predicate `check_theme/1` verifies a theme against the theme colours of xpce and the development tools. It reports theme colours that are not defined, defined twice or unknown, and values that are not colours. After changing a theme file, use `make/0` and `apply_theme/1` to see the result. A theme file may also define the colours of the Prolog console using prolog:console_color/3, as `library(theme/dark)` does.

Variations of a theme A theme can be a variation of another theme, its *base*, using the hook prolog:theme_base/2. It uses the theme colours, console colours and online help styles of its base that it does not define itself. A variation of the `light` or `dark` theme thus defines only the colours it changes:

```
:- module(prolog_theme_my_dark, []).
:- use_module(library(theme/dark)).
:- multifile
    prolog:theme/1,
    prolog:theme_base/2,
    pce_theme:colour/3.

prolog:theme(my_dark).
prolog:theme_base(my_dark, dark).

pce_theme:colour(my_dark, syntax_comment, '#6a9955').
```

The roles of a theme that is based on another theme always use the value of the theme rather than the system colour. Thus, if `my_dark` defines `ui_window_background`, this colour is also used on a dark desktop. Roles that it inherits from `dark` follow the desktop as described above.

Theme colours of an application An application that wants its colours to follow the theme declares theme colours with their value in the `light` theme and uses the name of the theme colour wherever it needs the colour, for example in a class variable:

```
:- use_module(library(pce_theme)).
:- theme_colours([ myapp_highlight = khaki1 ]).

class_variable(highlight, colour, myapp_highlight).
```

A theme defines the value for the other themes:

```
pce_theme:colour(dark, myapp_highlight, khaki4).
```

Graphicals that refer to a theme colour follow the theme by themselves. Colours that are *copied*, however, do not. This applies to an image into which graphicals are drawn using '`image` → `draw_in`', to a colour computed from another colour and to colours passed to an external program such as Graphviz. For these, a frame or graphical class of the application defines →`colours_changed`. This message is sent by '`display_manager` → `colours_changed`' to all frames and graphicals whose class defines it, before all windows are redrawn:

```
colours_changed(Menu) :->
    "Draw the icons again in the new colours"::
    send(Menu, paint_icons).
```

Colours chosen by the user A colour that is chosen by the user, for example the background of an Epilog profile, is defined using `adaptive_colour/3`. The theme colour is the given colour if this is as dark or as light as a reference, for example `ui_window_background` for a background colour. If not, the lightness of the colour is mirrored, keeping its hue. A light yellow background thus becomes a dark olive one in the dark theme. It is also possible to give a value per theme. The Epilog profile options `background`(*Colour*) and `foreground`(*Colour*) and the colours of `set_epilog/1` use this:

```
epilog:profile(shell, [ background(lightgoldenrodyellow) ]).
epilog:profile(build, [ background([ light = lightgoldenrodyellow,
                                     dark  = '#33301e'
                                   ])
                      ]).
```

Predicates The library `pce_theme` provides the following predicates. See the documentation of the library for details.

apply_theme(+*Theme*)
: Make *Theme* the active theme and redraw all windows. This sets the Prolog flag `theme`, which loads `library(theme/`*Theme*`)` if it exists. Setting the flag directly using `set_prolog_flag/2` has the same effect.

select_theme(+*Theme*)
Select *Theme* or, if *Theme* is `system`, follow the desktop for the running session. This is used by the Settings/Theme menu.

current_theme(-*Theme*)
Theme is the active theme: the value of the Prolog flag `theme`, where `default` is `light`.

theme_colours(+*List*)
Declare the theme colours of an application. *List* is a list of *Name* = *Value*, where *Value* is the value in the `light` theme.

adaptive_colour(+*Name, +Spec, +Reference*)
Define the theme colour *Name* from a colour chosen by the user.

check_theme(+*Theme*)
Report problems with the colours defined by *Theme*.

create_theme(+*Theme, +Base*)
Create *Theme* in memory as a variation of *Base*. Its settings are changed using `set_theme_setting/2` and `reset_theme_setting/2`. This is used by the theme editor.

The console colours of a theme, prolog:console_color/3, follow the active theme. Text that is already written keeps its colours.

14.10.2 Supported Image Formats

Images are loaded using the SDL_image library. The supported formats thus depend on how this library was built. Typically, these include PNG, SVG, JPEG, GIF, WebP, AVIF, TIFF, BMP, ICO and CUR, PNM, TGA, QOI and XPM. The X11 bitmap format (XBM) is not supported. Images can be loaded from a file or any other `source_sink`, e.g., a `resource` (see section 13). The format is determined from the content of the file. The preferred formats for images that are part of an application are PNG and SVG.

An SVG image is rendered when it is loaded. If '`image → initialise`' specifies a width and/or height, it is rendered at this size, where a missing dimension is computed from the aspect ratio. Otherwise it is rendered at the size specified in the SVG file. For example:

```
?- new(I, image('document.svg', 64)).
```

Each image is an array of RGBA pixels. Transparent areas are defined by the alpha channel. The '`image ← kind`' is `pixmap` for a normal image. If all pixels of a loaded image are either opaque black or transparent, the kind is `bitmap`. When drawn, the black pixels of a bitmap are painted in the current foreground colour, while the transparent pixels remain transparent. Such images can thus be used as icons that follow the colour of, e.g., the text around them.

Images can be saved using '`image → save`', which writes the image to a file in PNG format. This is the only format supported for saving. Hot-spots stored in image files, e.g., in XPM or CUR files, are not used: '`image → hot_spot`' must be set explicitly.

bitmap	A bitmap converts an image into a first class graphical object that can be displayed anywhere.
cursor	A cursor may be created from an image and a hot-spot.
'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 → label'	The items of a menu can be an image.
'tab → icon'	An image shown in the label of a tab.
'style → icon'	Allows association of images to lines in a list_browser, as well as marking fragments in an editor.

Table 14.4: GUI classes using image objects

Using Images

Images in any of the formats are recognised by many of XPCE's GUI classes. Table table 14.4 provides a brief list:

Cursors

The mouse pointer of a graphical is set using 'graphical → cursor'. A cursor is either one of the standard cursors of the platform or is created from an image. The standard cursors are named by their role: default, pointer, text, crosshair, move, wait, progress, not_allowed and the resize cursors ew_resize, ns_resize, nwse_resize, nesw_resize, n_resize, e_resize, etc. For compatibility, the most commonly used names of the old X11 cursor font, such as top_left_arrow, watch or xterm, are mapped to the closest standard cursor. The standard cursor none hides the pointer, e.g., while a game that is played with the keyboard has the focus. 'cursor ← kind' is standard, x11 or image and 'cursor ←summary' describes a standard cursor. The cursor demo shows all cursors. A cursor is created from an image using the image and its hot-spot, the point that is the actual pointer location:

```
?- new(C, cursor(@nil, image('my_cursor.png'), point(8,8))).
```

14.11 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.

14.11.1 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 `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))).
```

```
10
11 add_box(P, At:point) :->
12         send(P, display, new(link_box), At).
13
14 :- pce_end_class(link_demo).
15
16 :- pce_begin_class(link_box, box).
17
18 handle(w/2, 0, link, north).
19 handle(w/2, h, link, south).
20 handle(0, h/2, link, west).
21 handle(w, h/2, link, east).
22
23 initialise(B) :->
24         send_super(B, initialise, 100, 50),
25         send_list(B, recogniser,
26                   [ click_gesture(left, '', double,
27                                   message(B, edit)),
28                     new(connect_gesture),
29                     new(move_gesture)
30                   ]).
31
32 colour(red).
33 colour(green).
34 colour(blue).
35 colour(yellow).
36
37 edit(B) :->
38         "Allow changing colour"::
39         new(D, dialog('Select colour')),
40         send(D, append, new(M, menu(colour, choice,
41                                     message(?(D, hypered, box),
42                                             fill,
43                                             @arg1)))),
44         forall(colour(Colour),
45                ( new(I, image(@nil, 32, 16)),
46                  send(I, fill, colour(Colour)),
47                  send(M, append,
48                       menu_item(colour(Colour), @default, I))
49                )),
50         send(D, append, button(done, message(D, destroy))),
51         new(_, partof_hyper(B, D, dialog, box)),
52         get(B, display_position, PosB),
53         get(PosB, plus, point(20,100), PosD),
54         send(D, open, PosD).
55
56 :- pce_end_class(link_box).
```

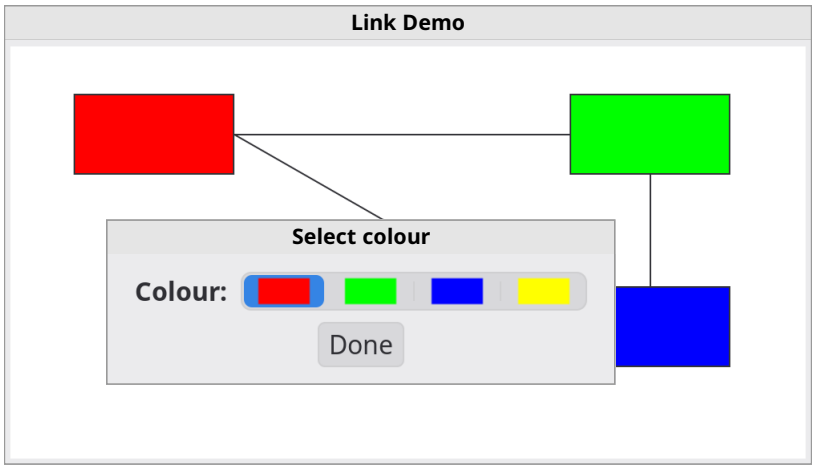


Figure 14.4: Using a hyper to link a window to an object

14.11.2 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

14.12 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 9), 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 9.3.2 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.

14.12.1 (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.

  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`. Clipping regions nest: a →`clip` inside another one

→draw	Paint other graphical
→clip	Clip to area or ←area of graphical
→unclip	Restore the clip before the 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 rectangle with colour or gradient
→draw_image	Paint (part of) image
→draw_line	Draw a line segment
→draw_markup	Draw Pango markup
→draw_path	Draw a path object
→draw_poly	Draw a polygon
→draw_text	Draw string in font
→paint_selected	Paint visual feedback of →selected

Table 14.5: Methods for (re)defining →_redraw_area

clips to the intersection of both. Note that is is **not** necessary to limit the applied paint only inside the given argument *Area*. The graphical infra-structure 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 14.5. Full documentation is available from the online manual.

14.12.2 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,
2                     "Graphical window with optional grid'").'
3
4  variable(grid,      '1..|size*' := 20, get,
```

```
5           "Size of the grid").
6   variable(grid_pen, pen,                 get,
7           "Pen used to draw the grid").
8
9   initialise(P, Lbl:[name], Size:[size], Disp:[display]) :->
10          send(P, send_super, initialise, Lbl, Size, Disp),
11          send(P, slot, grid_pen, pen(1, none, grey90)).
12
13  '_redraw_area'(P, A:area) :->
14          "Draw a grid"::
15          get(P, grid, Grid),
16          (   Grid \== @nil
17          ->  (   integer(Grid)
18              ->  GX = Grid,
19                  GY = Grid
20              ;   object(Grid, size(GX, GY))
21              ),
22              send(P, save_graphics_state),
23              get(P, grid_pen, pen(Pen, Texture, Colour)),
24              send(P, graphics_state, Pen, Texture, Colour),
25              object(A, area(X, Y, W, H)),
26              StartX is (X//GX) * GX,
27              StartY is (Y//GY) * GY,
28              Xlines is ((W + X - StartX)+GX-1)//GX,
29              Ylines is ((H + Y - StartY)+GY-1)//GY,
30              (   between(1, Xlines, Xline),
31                      Xnow is StartX + (Xline-1)*GX,
32                      send(P, draw_line, Xnow, Y, Xnow, Y+H),
33                  fail
34              ;   true
35              ),
36              (   between(1, Ylines, Yline),
37                      Ynow is StartY + (Yline-1)*GY,
38                      send(P, draw_line, X, Ynow, X+W, Ynow),
39                  fail
40              ;   true
41              ),
42              send(P, restore_graphics_state)
43          ;   true
44          ),
45          send(P, send_super, '_redraw_area', A).
46
47
48  grid(P, Grid:'1..|size*') :->
49          send(P, slot, grid, Grid),
50          send(P, redraw).                % changed?
51
52  grid_pen(P, Pen:pen) :->
53          send(P, slot, grid_pen, Pen),
54          send(P, redraw).                % changed?
55
56  :- pce_end_class.
```

14.12.3 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.
- −− 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.

```
:- pce_begin_class(text_shape, graphical,
                   "text with box or ellipse").

variable(string,        char_array,     get,
         "Displayed string").
variable(font,          font,           get,
         "Font used to display string").
variable(shape,         {box,ellipse},  get,
         "Outline shape").

initialise(S, Str:string=char_array, Shape:shape={box,ellipse},
           W:width=int, H:height=int, Font:[font]) :->
        default(Font, normal, TheFont),
        send(S, send_super, initialise, 0, 0, W, H),
        send(S, slot, string, Str),
        send(S, slot, shape, Shape),
        send(S, slot, font, TheFont).

'_redraw_area'(S, A:area) :->
        get(S, area, area(X, Y, W, H)),
        get(S, string, String),
        get(S, font, Font),
        get(S, shape, Shape),
        send(S, clip),                  % text may be bigger
        (   Shape == box
        ->  send(S, draw_box, X, Y, W, H)
        ;   send(S, draw_arc, X, Y, W, H)
        ),
        send(S, draw_text,
             String, Font, X, Y, W, H,
             center, center),
        send(S, unclip),
        send(S, send_super, '_redraw_area', A).

:- pce_end_class.
```

14.13 Printing

XPCE prints graphics by rendering them as PDF. The PDF file may be saved or handed to the printing facilities of the operating system. Rendering uses Cairo, the same library that is used to paint on the screen, and therefore the PDF output looks the same as the screen. Text and line drawings are preserved as vectors, so the output is scalable.

graphical → pdf: File:file, Scale:[num]
: Write the graphical to File as a single page PDF document. The page has the size of the graphical, multiplied by Scale (default 1.0). For a `device`, all graphicals displayed on the device are included.

window → pdf: File:file, Scale:[num]
: Write the full contents of the window as a single page PDF document. Unlike the screen, the output is not restricted to the visible part of the window: the page covers the bounding box of all graphicals displayed on the window.

graphical → svg: File:file, Scale:[num]
: As '`graphical → pdf`', but write an SVG image. The size of the image is in pixels. Text is written as outlines, so the image looks the same without the fonts.

window → svg: File:file, Scale:[num]
: As '`window → pdf`', but write an SVG image.

SVG output depends on the Cairo library XPCE is built with. Test whether it is available using `send(@pce, has_feature, svg)`.

The library `print_graphics` defines the class template `print_graphics` that adds methods for printing and saving as PDF to a window. It is used by, for example, PceDraw. The method →`print` prints the window on the default printer by passing the PDF to the command defined by ←`print_command_template` (`lpr` by default) and →`save_pdf` prompts for a file and saves the contents of the window as PDF.

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

:- pce_begin_class(my_picture, picture).
:- use_class_template(print_graphics).
...
:- pce_end_class(my_picture).
```

15 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.

15.1 **Asking a filename**
This library defines the object @finder, instance of finder. The finder allows for asking filenames.

15.2 **Show help-balloon**
The library help_message registers balloon-text with graphical objects.

15.3 **Dialog utilities**
Toolbars and reporting facilities. Includes example code for an application framework.

15.4 **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.

15.5 **Tabular layout**
Primitives for dealing with tables.

15.6 **Plotting graphs and barcharts**
This section describes a number of libraries providing primitives for drawing graphs and barcharts.

15.7 **Multi-lingual applications**
Discusses support for multi-lingual applications.

15.8 **Drag and Drop Interface**
This library allows for drapping objects within one XPCE application.

15.9 **Document rendering**
Primitives for rendering mixed text/graphics, handling fonts, alignment, tables and other common text-layout primitives.

15.1 Library "find_file"

The 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—{open,save}],
Ext:[name—chain—tuple], Dir:[directory],
Def:[name] → name

Ask the user for a file. If *Exists* is @on or `open`, only existing files can be returned. The value `save` (or @off) prompts for a file to write. *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 class `finder` always uses the portable XPCE dialog defined by the class `find_file_dialog` (library `find_file_dialog`). Applications that prefer the native file dialog of the operating system can use '`frame ← open_file`' and '`frame ← save_file`'. These methods are realised using SDL3 and block until the user has selected a file or cancelled the dialog (in which case they fail). The filters are a `chain` of `tuple` objects holding a label and an extension or chain of extensions. For example:

```
        ...,
        get(Frame, open_file,
            filters := chain(tuple('Prolog files', chain(pl, prolog))),
            File),
        ...,
```

'`frame ← open_file`' accepts `allow_many := @on`, in which case it returns a `chain` of file names.

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.

15.2 Showing help-balloons

The 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 technique is commonly used in modern interfaces to provide feedback on the functions behind icons. See also section 15.3.2.

In addition to registering the global event-handler, the library defines →help_message and ←help_message on the classes visual and dict_item. The classes menu and list_browser redefine ←help_message to return the message associated with the menu_item or dict_item below the pointer. If a graphical has no help message, the message of the graphical it is contained in is used.

visual → help_message: {tag,summary}, string*
: Register *string* as tag (balloon) or extensive help message (summary) for the receiving object. The summary message is displayed if the user presses the help function key. Using @nil deletes the message.

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.

visual → tooltip: string*

visual ← tooltip→ string
: Shorthand for →help_message and ←help_message with tag. These are also defined on dict_item.

A message may be an instance of markup, a string holding Pango markup, which is rendered in the balloon:

```
1          send(B, tooltip, markup('Print the <b>whole</b> document')),
```

Here is a typical usage for this library.

```
1  :- use_module(library(help_message)).
2  resource(print, image, image('tool/print.svg')).
3
4          ...
5          send(X, append, new(B, button(print))),
6          send(B, label, image(resource(print))),
7          send(B, tooltip, 'Print document'),
```

15.3 Dialog support libraries

This section deals with a number of classes from the library to simplify the creation of dialog windows.

15.3.1 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 14.8. The library `pce_report` defines the classes `reporter` and `report_dialog`.

- `reporter`
  This is a refinement of class `label` for displaying messages. It redefines the →`report` message to colour error messages red and keeps errors and warnings visible for a few seconds (see ←`error_delay` and ←`warning_delay`) before they are overwritten by less important messages.

- `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 15.3.2.

15.3.2 Toolbar support

The 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 toolbars 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. If it is a name, this balloon is subject to '`name ← label_name`' (see section 15.7), otherwise it is passed literally. Finally, if *Condition* is present it is evaluated by →`activate` to determine the activation-state of the button.

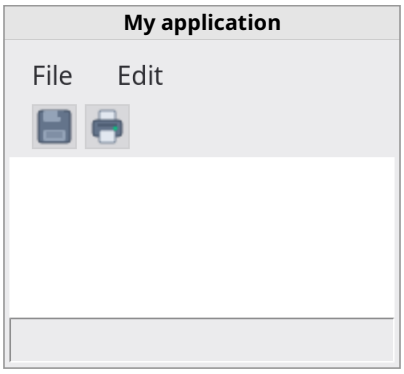


Figure 15.1: Simple application framework

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 14.2.

The library also defines the class `tool_dialog`, a `dialog` that manages a `menu_bar` and `tool_bar` for a client. '`tool_dialog → append`' accepts `popup` objects (added to the menu bar), `menu_item` objects (added to the popup named by the second argument) and `tool_button` objects (added to the tool bar). Selected menu items send their value as a message to the client.

15.3.3 Example

The example below uses these classes as well as class `menu_bar` to arrive at a typical modern application layout.

```
1  %       Pull in the classes
2
3  :- pce_autoload(report_dialog, library(pce_report)).
4  :- pce_autoload(tool_bar, library(toolbar)).
5  :- pce_autoload(finder, library(find_file)).
6  :- pce_global(@finder, new(finder)).
7
8  %       Define icons as program resources
9
10 resource(printer,       image,  image('tool/print.svg')).
11 resource(floppy,        image,  image('tool/save.svg')).
12
13 %       Define the application as a subclass of frame.
14
15 :- pce_begin_class(myapp, frame,
16                     "Frame representing the application").
```

```
17
18 initialise(MyApp) :->
19         send_super(MyApp, initialise, 'My application'),
20         send(MyApp, append, new(D, dialog)),
21         send(D, pen, 0),
22         send(D, gap, size(5, 5)),
23         send(D, append, new(menu_bar)),
24         send(D, append, new(tool_bar(MyApp))),
25         send(MyApp, fill_menu_bar),
26         send(MyApp, fill_tool_bar),
27         send(new(W, myapp_workspace), below, D),
28         send(new(report_dialog), below, W).
29
30 fill_menu_bar(F) :->
31         get(F, member, dialog, D),
32         get(D, member, menu_bar, MB),
33         send_list(MB, append,
34                   [ new(File, popup(file)),
35                     new(_Edit, popup(edit))
36                   ]),
37
38         send_list(File, append,
39                   [ menu_item(load,
40                               message(F, load),
41                               end_group := @on),
42                     menu_item(print,
43                               message(F, print))
44                   ]).
45
46 fill_tool_bar(F) :->
47         get(F, member, dialog, D),
48         get(D, member, tool_bar, TB),
49         send_list(TB, append,
50                   [ tool_button(load,
51                                 resource(floppy),
52                                 load),
53                     gap,                % skip a little
54                     tool_button(print,
55                                 resource(printer),
56                                 print)
57                   ]).
58
59 print(MyApp) :->
60         "Print the document"::
61         send(MyApp, report, progress, 'Printing ...'),
62         get(MyApp, member, myapp_workspace, WS),
63         send(WS, print),
64         send(MyApp, report, progress, done).
65
66 load(MyApp) :->
67         "Ask a file and load it"::
68         get(@finder, file, @on, myp, File),
69         get(MyApp, member, myapp_workspace, WS),
```

```
70          send(WS, load, File).
71
72  :- pce_end_class(myapp).
73
74
75  %       dummy class for the work-area of your application.  The
76  %       print_graphics template defines ->print.
77
78  :- use_module(library(print_graphics)).
79
80  :- pce_begin_class(myapp_workspace, window).
81  :- use_class_template(print_graphics).
82
83  :- pce_end_class(myapp_workspace).
```

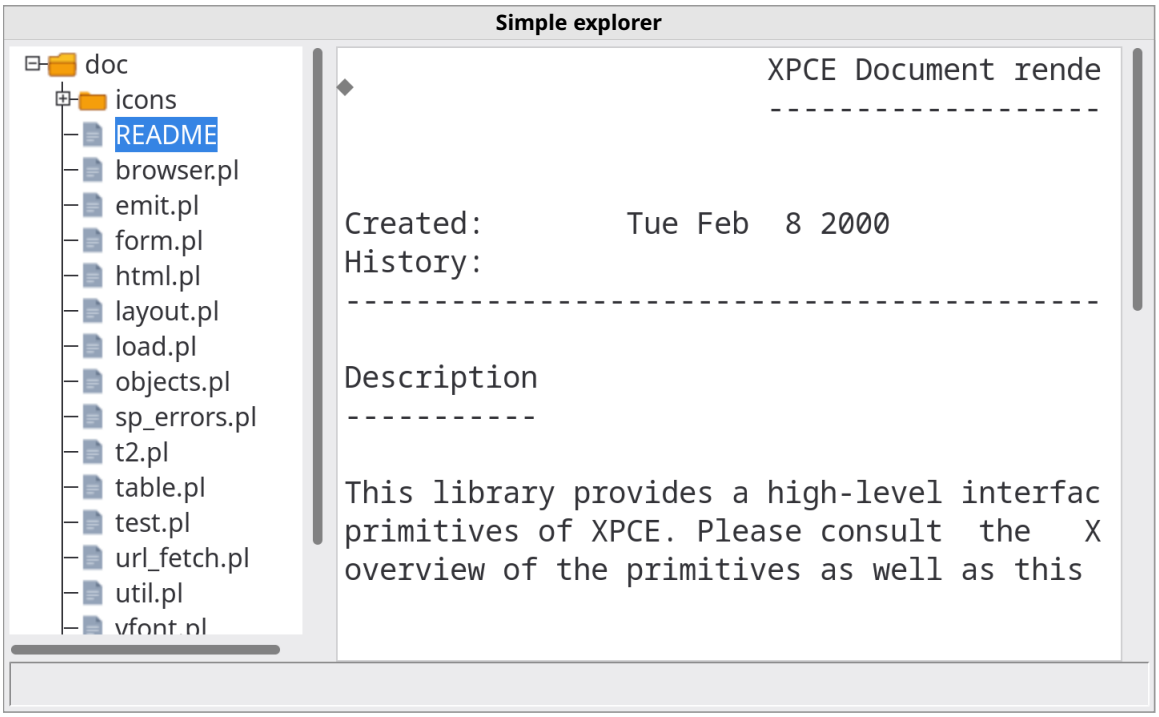


Figure 15.2: Exploring the filesystem

15.4 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 15.2.

```
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,
```

```
                    "Browser for a directory-hierarchy").

initialise(FB, Root:directory) :->
        send(FB, send_super, initialise),
        get(Root, name, Name),
        send(FB, root, toc_folder(Name, Root)).

expand_node(FB, D:directory) :->
        "Called if a node is to be expanded"::
        new(SubDirsNames, chain),
        new(FileNames, chain),
        send(D, scan, FileNames, SubDirsNames),

        get(SubDirsNames, map, ?(D, directory, @arg1), SubDirs),
        send(SubDirs, for_all,
             message(FB, son, D,
                     create(toc_folder, @arg1?name, @arg1))),
        get(FileNames, map, ?(D, file, @arg1), SubFiles),
        send(SubFiles, for_all,
             message(FB, son, D,
                     create(toc_file, @arg1?base_name, @arg1))).

open_node(FB, Node:file) :->
        "Called if a file is double-clicked"::
        send(FB?frame, open_node, Node).

:- 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 above 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 → collapse_node: Id:any
: The user clicked the `[-]` sign of an expanded folder. The default implementation hides the sub-nodes.

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, Relink:[bool]
Assign the hierarchy a (new) root.

toc_window → delete: Id:any
Delete the node representing *Id* and all its sub-nodes.

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 | graphical, 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 → initialise: Label:char_array | graphical, Id:[any], Img:[image]
Create a 'file'-node from its *Label*, *Id* (defaults to the created node object) and *Img* (defaults to a 'document' icon).

Drag-and-drop of nodes is enabled using 'toc_window → drag_and_drop'. The library toc_filesystem refines this library with the classes toc_filesystem and toc_directory for browsing the filesystem.

15.5 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
  An 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 15.5.1.

- *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 tabular provides a user-friendly front-end for most of the functionality of class table.

15.5.1 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 pce_image_browser.

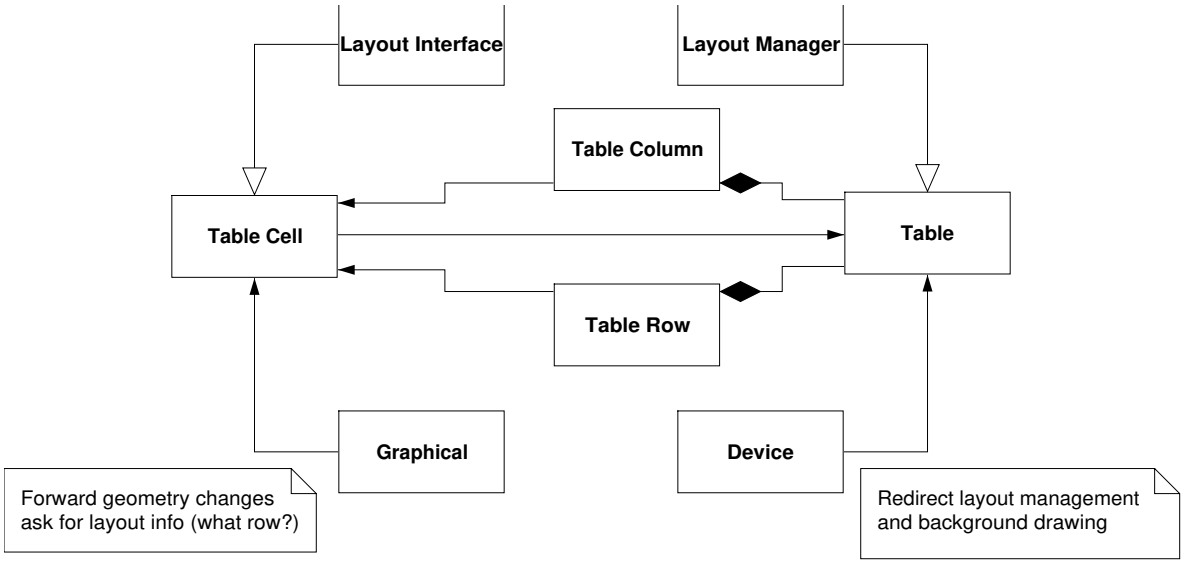


Figure 15.3: Layout manager interface for tables

15.5.2 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 realised using the methods '`device` ⇆ `layout_manager`' and '`graphical` ⇆ `layout_interface`'. Figure 15.3 gives an overview of the classes realising tabular layout.

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: Table:[table]**

Create a device with associated table. If *Table* is omitted, a new table is created.

tabular → **append: Label:name | graphical | table_cell, 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. If this is a `name`, a `text` is created. A graphical is immediately placed in a new cell of the table and a `table_cell` is appended as is.
- 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 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 if it does not 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')
                  ]),
        send(P, open).
```

Table with merged cells

	Length/mm		
	body	**tail**	**ears**
males	31.4	23.7	3.8
females	29.6	20.8	3.4

Figure 15.4: Small table with row/column spanning

15.6 Plotting graphs and barcharts

This section describes three libraries residing in ⟨*pcehome*⟩`/prolog/lib/plot` to deal with plotting graphs and barcharts. The demo `charts_demo/0` from `demo(charts)`, also available from the demo browser of the online manual (see `manpce/0`), is a gallery showing most of the features described in this section.

15.6.1 Painting axis

The 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.

```
?- use_module(library(plot/axis)).
?- use_module(library(autowin)).

?- new(P, auto_sized_picture),
           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))),
           send(P, open).
```

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=num, high=num, step=[num], 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: num → int

Determine the coordinate in the device's coordinate system representing the given value. See also '`plotter ← translate`'.

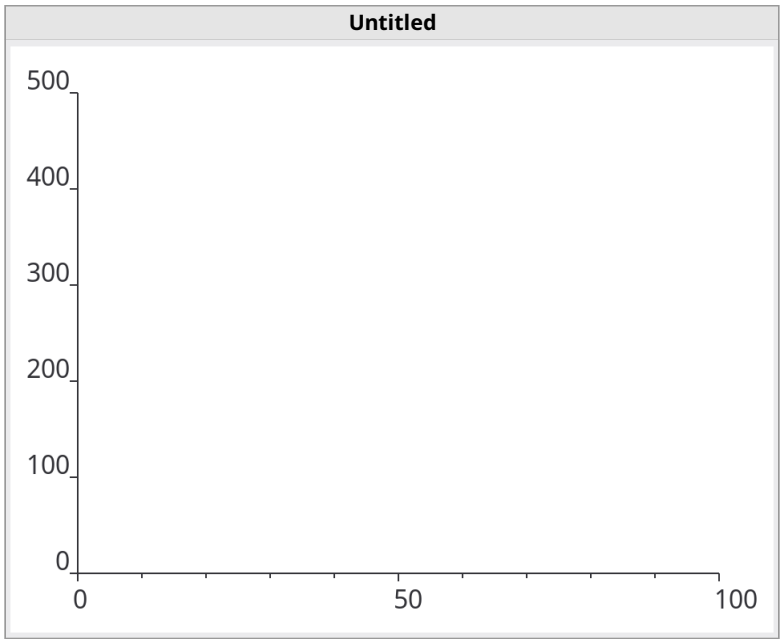


Figure 15.5: A picture showing two axis

plot_axis ← value_from_coordinate: int → num
The inverse of ←`location`, returning the value along the axis from a pixel coordinate.

plot_axis → scale: {linear,log}
Use a linear (default) or logarithmic scale. For a logarithmic scale the low end must be > 0 and ←`step` is the factor between two rule-marks.

plot_axis → lines: {none,normal,all}
Determines whether lines are drawn across the plot at each step (`normal`, default), also at each small step (`all`) or not at all (`none`).

plot_axis ← label_for_value: num → graphical
Compute the graphical displayed next to a rule-mark. The default uses ←`format`. Subclasses may redefine this to print, for example, names of months.

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}.

15.6.2 Plotting graphs

The 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 $Y = sine(X)$

```
1 :- use_module(library('plot/plotter')).
2 :- use_module(library(autowin)).
3
4 plot_function :-
```

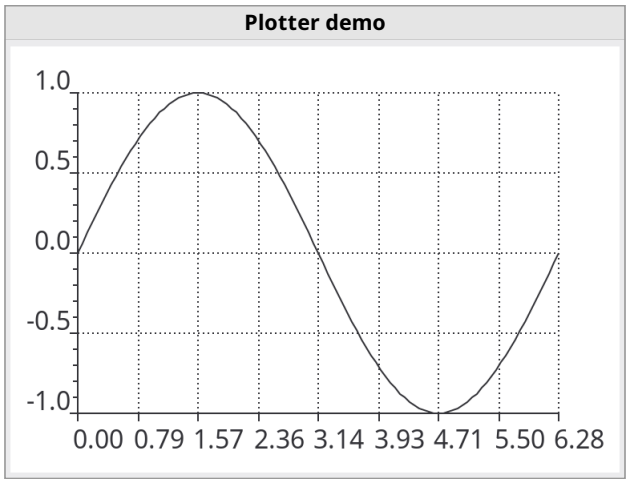


Figure 15.6: Plotter showing sine function

```
5           plot_function(X:sin(X)).
6
7   plot_function(Template) :-
8           To is 2*pi,
9           PlotStep is To/100,
10          Step is pi/4,
11          new(W, auto_sized_picture('Plotter demo')),
12          send(W, display, new(P, plotter)),
13          send(P, axis, new(X, plot_axis(x, 0, To, Step, 300))),
14          send(P, axis, plot_axis(y, -1, 1, @default, 200)),
15          send(X, format, '%.2f'),
16          send(P, graph, new(G, plot_graph)),
17          plot_function(0, To, PlotStep, Template, G),
18          send(W, open).
19
20  plot_function(X, To, _, _, _) :-
21          X >= To, !.
22  plot_function(X, To, Step, Template, G) :-
23          copy_term(Template, X:Func),
24          Y is Func,
25          send(G, append, X, Y),
26          NewX is X + Step,
27          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.

plotter ← translate: X:num, Y:num → point
Translate a coordinate in the value-space to physical coordinates.

plotter ← value_from_x: int → num
Translate an X-coordinate to a value.

plotter ← value_from_y: int → num
Translate an Y-coordinate to a value.

plotter → hover_values: bool
If `@on` (default `@off`), moving the pointer over the plotter shows a thin vertical line at the pointer and, for each graph (any `path` on the plotter), a box in the colour of the graph holding its value at that *X*. The boxes are stacked such that they do not overlap. An application that changes its graphs while the pointer is over the plotter calls →`update_hover` to show the new values.

plotter ← hover_value: path, X:int → num
Value of a graph at pixel *X*. For a `plot_graph` this interpolates the values of the two points on either side of *X*; for other paths it translates the pixel *Y* of the path at *X* using ←`value_from_y`. Fails if the graph has no value at *X*. A subclass may redefine this to return the exact data.

plotter ← hover_label: path, num → char_array
Text for a value of a graph. The default formats the value using '`plot_axis ← label_for_value`' of the *Y* axis.

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=num, y=num
Append a control-point using the coordinate-system of the axis of the plotter.

Individual values can be annotated using a `plot_mark`, a `device` that is created from an *X* and *Y* value and an optional graphical that is centred at the corresponding point. It is displayed on the `plotter` using →`display` and keeps its position if the axes change.

15.6.3 Drawing barcharts using "plot/barchart"

The `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 `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 shown below.

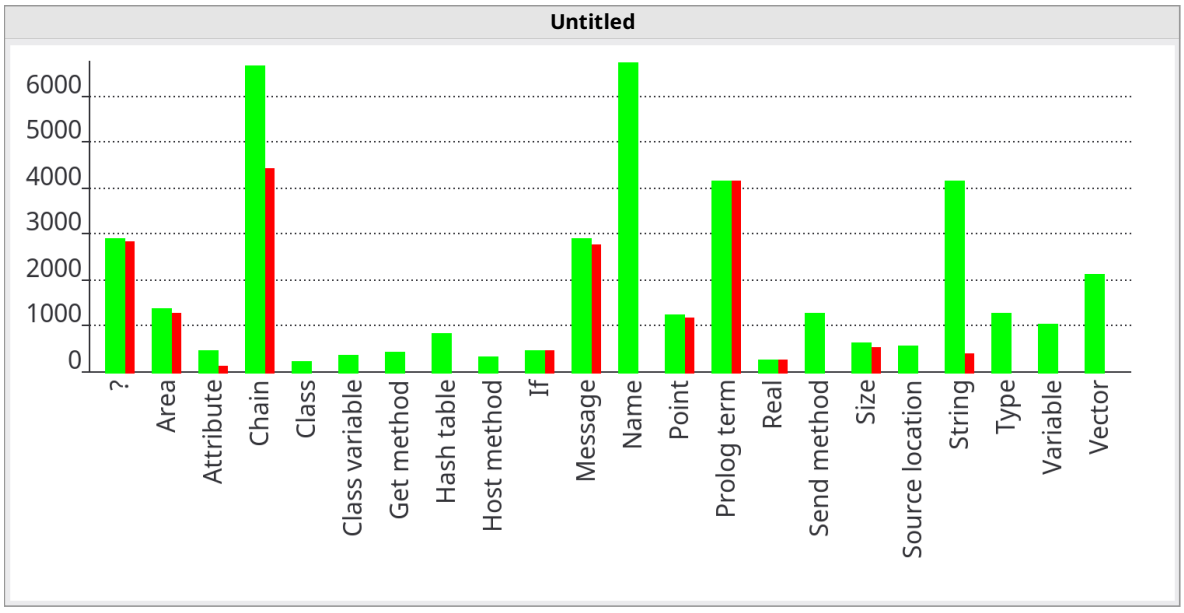


Figure 15.7: Classes of XPCE with $>$ 250 instances created

```
barchart :-
        barchart(vertical).
barchart(HV) :-
        new(W, picture),
        active_classes(Classes),
        length(Classes, N),
        required_scale(Classes, Scale),
        send(W, display, new(BC, bar_chart(HV, 0, Scale, 200, N))),
        forall(member(class(Name, Created, Freed), Classes),
               send(BC, append,
                    bar_group(Name,
                              bar(created, Created, green),
                              bar(freed, Freed, red)))),
        send(W, open).
```

bar_chart → initialise: orientation={horizontal,vertical}, low=real,
high=real, scale_length=[0..], nbars=[0..],
bar_width=[0..], bar_gap=[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 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. Finally, the width of the bars and the distance between them may be specified.

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
num Modifies or requests the value of the named bar. Fails if no bar with this name is on the chart.

bar_chart → sort
Sort the bars by their value.

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: num
Set the value of the bar. This updates the bar-size automatically.

bar → range: low=num*, high=num*
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`.

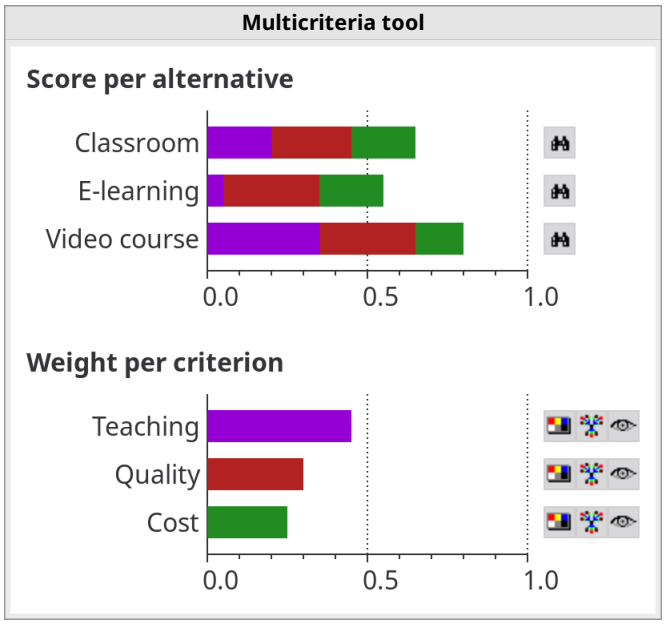


Figure 15.8: Stacked bars with associated buttons

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 15.8 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 to. This behaviour may be used to make the buttons less dependent on the bar they are attached to.

15.6.4 Hierarchical distributions using "plot/hierarchy_chart"

The library `plot/hierarchy_chart` shows how a quantity, such as disk space, time or memory, is distributed over a hierarchy, in the style of GNOME's disk usage analyser `baobab`. The hierarchy

is a tree of `hierarchy_node` objects. There are two views on the same tree: a `rings_chart` shows the *focus* node as a disc surrounded by a ring per level below it, where the angle of a segment is its share of the focus; a `treemap_chart` divides a rectangle into a rectangle per child, nested for the levels below it. Both views colour a node by its position in the focus, so the colours of the two views agree. Moving the pointer over a node highlights it and shows a balloon with its name and value. If the pointer rests on a node of a rings chart, the names of its children are shown around the rings, each connected to its segment by a line (see the class variables `hierarchy_chart.label_delay` and `rings_chart.max_labels`). Clicking a node zooms into it: it becomes the new focus. Clicking the focus zooms out.

```
:- use_module(library(plot/hierarchy_chart)).

show_usage :-
        hierarchy_from_term(node(home, 0,
                                 [ node(src, 0, [ swipl-900, other-300 ]),
                                   photos-1500,
                                   notes-10
                                 ]), Root),
        new(F, frame('Usage')),
        send(F, append, new(W, hierarchy_window(Root))),
        send(W, unit, bytes),
        send(new(D, dialog), below, W),
        send(D, append, label(reporter)),
        send(F, open).
```

hierarchy_from_term(+*Term, -Node*)
: Create a tree of `hierarchy_node` objects from a term `node`(*Label, Value, Children*), `node`(*Label, Value, Children, Options*) or `Label-Value` for a leaf. Options is a list holding `data`(*Data*), `colour`(*Colour*) and `title`(*Title*), a longer name for the balloon.

Class `hierarchy_node`

hierarchy_node → initialise: label=char_array, value=[num], data=[prolog]
: Create a node. *value* is the value of the node itself, excluding the nodes below it, and *data* is application data, available as ←`data`.

hierarchy_node → append: hierarchy_node
: Add a node below this one.

hierarchy_node ← total→ num
: The value of the node and all nodes below it. The total is cached and recomputed if a value below it changes through →`value`, →`append` or →`delete`.

hierarchy_node ← path_label: separator=[char_array] → string
: The labels from the root down to this node.

hierarchy_node → colour: colour*
: Use this colour rather than one computed from the node's position.

Classes `rings_chart` **and** `treemap_chart`

Both are subclasses of the abstract class `hierarchy_chart`, a `device`. The segments of the charts are instances of `hierarchy_segment`, a `path` that accepts events exactly where it is painted (see '`path ⇆ event_area`').

hierarchy_chart → initialise: root=[hierarchy_node]*, extent=[size]
Create a chart of the given size (default 400x400).

hierarchy_chart → root: hierarchy_node*
Show a new hierarchy, with the root as focus. The chart owns the hierarchy: the nodes refer to each other and are thus not reclaimed automatically, so the chart frees the old hierarchy when it gets a new one and when it is freed itself.

hierarchy_chart → focus_node: hierarchy_node*
Zoom to a node. See also →`zoom_out` and →`zoom_to_root`.

hierarchy_chart → extent: size
Resize the chart.

hierarchy_chart → max_depth: 1..
The number of levels shown below the focus. The default is the class variable `hierarchy_chart.max_depth`: 5 for rings and 4 for a treemap.

hierarchy_chart → unit: {bytes,seconds,count,percentage,plain}
How values are shown. Redefine ←`format_value` in a subclass for other units.

hierarchy_chart → reference: num*
The value shown as 100% if ←`unit` is `percentage`. The default, `@nil`, uses the total of the root.

hierarchy_chart → select_message: code*
Called with a node without children that is clicked.

hierarchy_chart → click_message: code*
Called with the node that is clicked, instead of zooming into or out of it and calling ←`select_message`.

hierarchy_chart → focus_message: code*
Called with the new focus.

hierarchy_chart → hover_message: code*
Called with the node below the pointer, or `@nil` if the pointer leaves the chart.

Class `hierarchy_window`

A `picture` that shows a chart filling the window. It reports the path and value of the node below the pointer, and its popup zooms out and switches between the views.

hierarchy_window → initialise: root=[hierarchy_node]*, view=[{rings,treemap}]
Create the window. The default view is the class variable `hierarchy_window.view`.

hierarchy_window → view: {rings,treemap}
Switch the view, keeping the focus. This calls ←`view_message` with the new view.

hierarchy_window → view_message: code*
Called with the new view if the view changes, for example from the popup.

hierarchy_window ← popup_node→ hierarchy_node*
The node below the pointer when the popup opened, for items that act on it. Add such items to '`window ← popup`'.

hierarchy_window ← chart→ hierarchy_chart
The chart shown. →`root`, →`focus_node`, →`unit` and →`reference` are passed to it.

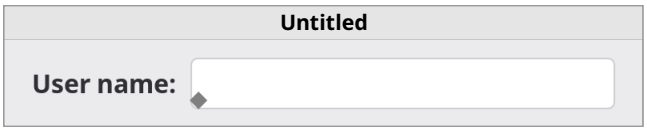
15.7 Multi-lingual applications

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 ← default_label: Value → Label
  Similar to '`dialog_item ← label_name`', computing the label of a `menu_item` that is created without an explicit label from its ←`value`.

  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 14.7, this approach is not very well suited for multi-lingual applications. A better approach is to use `error` objects, as described in section 14.8.

  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 used in the optics simulator to realise an English/Dutch application.

```
:- module(language,
          [ message/2,                  % Id, Message
            current_language/1,         % -Language
            set_language/1              % +Language
          ]).
:- use_module(library(pce)).

:- 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)
            ->  atomic_list_concat(Translation0, Translation)
            ;   Translation = Translation0
            )
        ;   Translation = Translations
        ), !.
message(Term, Translation) :-
        get(Term, label_name, Translation).


                /*******************************
                *    MAP DIALOG IDENTIFIERS    *
                *******************************/

:- pce_extend_class(dialog_item).

label_name(DI, Id:name, Label:'name|image') :<-
        "Multi-lingual label service"::
        message(Id, Label0),
        (   atomic(Label0)
        ->  get(DI, label_suffix, Suffix),
            get(Label0, ensure_suffix, Suffix, Label)
```

```
        ;   Label = Label0
        ).

:- pce_end_class.

:- pce_extend_class(dialog_group).

label_name(_DI, Id:name, Label:name) :<-
        "Multi-lingual label service"::
        (   message(Id, Label),
            atomic(Label)
        ->  true
        ;   get(Id, label_name, Label)
        ).

:- pce_end_class.

:- pce_extend_class(menu_item).

default_label(_MI, Id:name, Label:'name|image') :<-
        "Multilingual label service"::
        message(Id, Label).

:- pce_end_class.

                /*******************************
                *        GENERIC LABELS        *
                *******************************/

%       term(+Term, -Translated)
%
%       Term translates a term. There are three examples
%       here. The first only contains the translation
%       for an English label name into a Dutch one. The
%       second replaces all labels named label' into an'
%       image. The last is for generating a more
%       elaborate message from an identifier.

term(settings,
     [ dutch('Instellingen')
     ]).
term(label,
     image('label.png')).
term(start_named_test(Name),
     [ english(['Click "OK" to start test "', Name, '"']),
       dutch(['Klik op "OK" om aan de toets "', Name,
              '" te beginnen'])
     ]).


                /*******************************
                *             ERRORS           *
                *******************************/
```

```
107
108 %       error(Id, Kind, Translations)
109 %
110 %       Specify and create the required error messages.
111 %       An object that detects there are too many
112 %       instruments directs this information to the user
113 %       by
114 %
115 %               ...
116 %               send(MySelf, error, max_instruments, 5),
117 %               ...
118
119 error(max_instruments, error,
120       [ dutch('%IU kunt niet meer dan %d van deze \c
121               instrumenten gebruiken'),
122         english('%IYou can not use more than %d of \c
123                  these instruments')
124       ]).
125
126 make_errors :-
127         current_language(Lan),
128         T =.. [Lan, Message],
129         error(Id, Kind, Messages),
130         (   memberchk(T, Messages)
131         ->  true
132         ;   Message = Id
133         ),
134         new(_E, error(Id, Message, Kind, report)),
135         fail.
136 make_errors.
137
138 :- initialization make_errors.
```

15.8 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` from the library `dragdict` 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).
```

15.8.1 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`). While dragging, the pointer shows the `move` cursor (see the class-variable `cursor` of `drag_and_drop_gesture`). If this class-variable is @default, a cursor is created from the image of the dragged graphical. In that case *Warp* (default @on) determines whether the pointer is placed in the centre of this cursor. Dragging starts after the pointer has moved a few pixels with the button depressed. *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.

15.8.2 Dropping files from other applications

The interface above handles drag-and-drop between objects of the same XPCE process. Files and text dragged from other applications, such as the file manager of the desktop, are delivered as events. A drop sequence starts with a `drop_begin` event, followed by `drop_position` events while the user moves the pointer over the window, a `drop_file` or `drop_text` event for each dropped item and finally a `drop_complete` event. The dropped file or text is available as '`event ← path`' or '`event ← text`', which the event delegates to its `drop_data` ←`detail` (see section 7.1). The library `pce_drop_target` provides `drop_target_event/4` to handle this sequence from an →`event` method. It shows a hint while the user drags over the target and calls a predicate with the list of dropped files on completion. See also the demo `demo(dragdrop_file)`.

15.9 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 natural 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)),
```

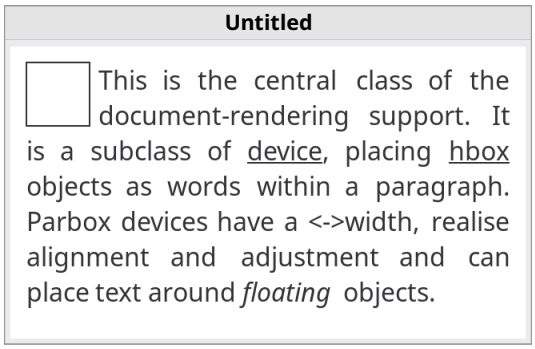


Figure 15.9: A parbox floating text around a box

```
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                    ]).
```

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.

15.9.1 The rendering library

The directory ⟨*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 `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 canonicalised, smashing multiple consecutive blanks 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 15.9.3.
- ⟨*Object*⟩
  XPCE object references 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 15.9.2.

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), !.
```

15.9.2 Predefined objects

The file `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.

15.9.3 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
After trying to pass the event to one of the embedded graphicals, this handles clicking a button or footnote, selecting text by dragging and reporting the balloon text of the button below the pointer as status message (see section 14.7).

pbox ← anchor: Name → tuple(Box, Index)
Return the *Box* and index thereof in the '`parbox ← content`' vector that starts the named anchor (see section 15.9.3).

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. Without a mapping, 'vfont ← font' creates a font from the family, slant and weight, using a size relative to the size of the normal font. This class is defined in 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. The text is placed in a code_block that shows an icon to copy the text to the clipboard if the pointer hovers over it.

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 ⟨*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. Defined options are `bgcolour`(*Colour*) and `text`(*Colour*).

button(*Message, Content, Balloon*)
Add an active area (hyper-link) to the document. When depressed, *Message* is executed. When hovering, *Balloon* is reported as status (see section 14.7). *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 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-right 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 15.5. 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*

15.9.4 Using the "doc/emit" library

In section 15.9.1 and section 15.9.3 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 text-file. 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 `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 ⟨*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 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</var> reflects the attribute-list of the
element and <var>ListOfContent</var> is mixed list of atoms representing
<em>CDATA</em> and <b>element/3</b> terms representing nested elements.
</body>
</html>
```

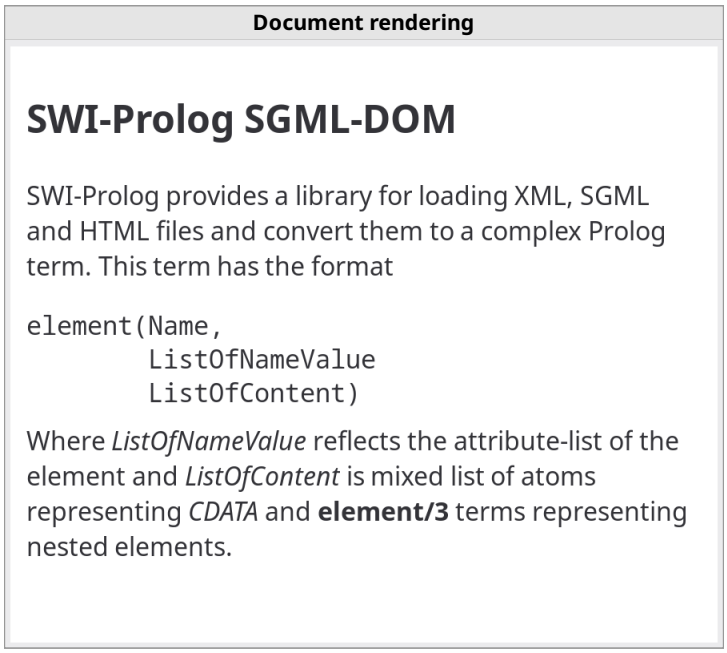


Figure 15.10: Rendering HTML code

In general you do not want to render plain HTML using XPCE/Prolog as it is far less flexible than real browsers dealing with erroneous HTML, the implementation of HTML is incomplete and it supports neither CSS nor JavaScript.

It has proved to be worthwhile using the extensibility of SGML and this layout to render domain-specific documents, often using HTML elements for the basic layout.

Development and debugging tools 16

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.

16.1 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).
B = <pce>(0x7f8a12c5d0,box),
Area = <pce>(freed).
```

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. For each problem, it prints the path from a global object to the faulty object. If problems are found, `checkpce/0` fails.

```
2 ?- checkpce.
% PCE: Checking 281 global objects, 1 Prolog references (1 freed).
[PCE warning: <pce>(0x7f8a12c5d0,box): Freed object in slot
  graphical -area: <pce>(0x7f8a12c400,area) (freed)]
        Path from checked object:
          ...
          @p/picture (refs=7, code_refs=0, locked) <-graphicals
          <pce>(0x7f8a12c600,chain) (refs=1, code_refs=0)[1]
          <pce>(0x7f8a12c5d0,box) (refs=3, code_refs=0) <-area
[PCE: Checked 6749 objects]
false.
```

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 14.4 and section 14.11 for techniques to avoid 'dangling' object references.

16.2 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)).
Tracing Send 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 an infix operator with priority 250, so no extra brackets are needed:

```
2 ?- tracepce(graphical<-area).
Tracing Get method: graphical <-area
```

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.

16.3 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.

nospypce(+*Spec*)
: Remove a spy point set with `spypce/1`.

16.4 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 Tools menu of the manual (see `manpce/0`) or using GUI/Inspect GUI hierarchy from the Epilog console or the other windows of the development environment. Figure 16.1 shows this tool examining the structure of the "PceDraw" demo program.

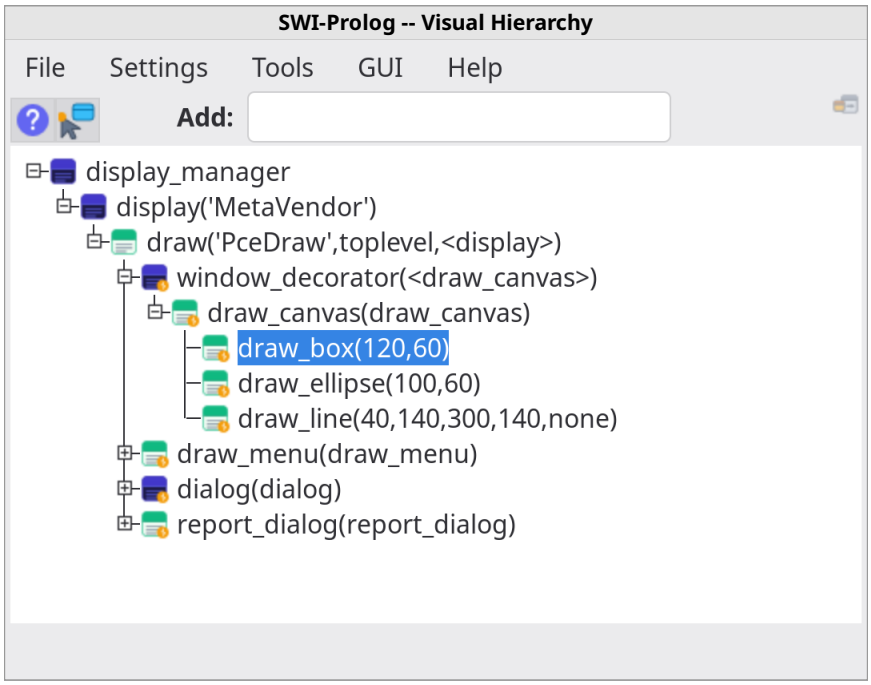


Figure 16.1: Visual Hierarchy Tool showing "PceDraw"

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 16.5) 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 "Add" text_item and press RETURN. The most comfortable way is to click the hand button on the tool bar and select the target object on the screen. The popup menu on a node of the tree allows for showing the source of a user-defined class, examining the class in the class browser and opening the inspector on the object.

16.5 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 argument reference in the Prolog window. The inspector is started from the Tools menu of the manual tools, using GUI/Inspect XPCE object or from the popup menu of the visual hierarchy tool. An object can be added by typing its reference or by selecting it on the screen using the hand button of the tool bar.

Each inspected object is shown as a card. The title of the card shows the object reference, its flags (e.g., `L` if the object is locked) and its reference count. The rows show the slots of the object and their values. Clicking a value that is an object flashes it if it is displayed, and double-clicking it adds a card for this object, linked to the slot by an arrow. Figure 16.2 shows a chain holding a point, an area and a name, where the point and area have been expanded.

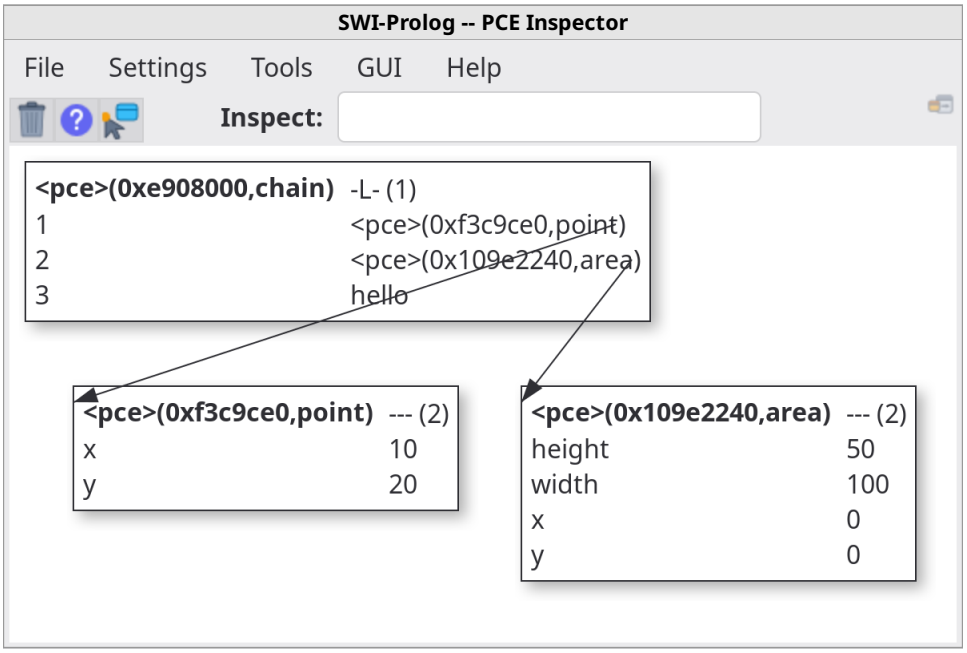


Figure 16.2: Inspector Tool

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-prolog.org.

[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-prolog.org.

[Wielemaker, 1994] J. Wielemaker. *XPCE/Prolog Course Notes*. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1994. E-mail: jan@swi-prolog.org.

[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-prolog.org.

SDL video drivers

A

XPCE relies on SDL to access host windows, the keyboard, mouse and other devices. SDL is primarily a library for implementing portable games. It is a good match for XPCE because XPCE GUI widgets are implemented in XPCE itself, providing maximal portability.

SDL provides multiple *drivers*, depending on how it is built and the hosting operating system. The default driver also depends on how the library is built as well as the OS. Details on window behaviour and event handling may be affected by the driver. The current driver can be retrieved using ‘`pce ← window_system_driver`‘, e.g.,

```
?- get(@pce, window_system_driver, Driver).
Driver = wayland.
```

Typical drivers are `windows` on MS-Windows, `cocoa` on MacOS and `x11` or `wayland` on Unix-like systems. The driver in use is also shown by ‘`pce → banner`’.

If multiple drivers are supported, a specific driver may be selected using the environment variable `SDL_VIDEODRIVER` or using the SWI-Prolog flag `'SDL_VIDEODRIVER'`. The latter allows for e.g.,

```
swipl -DSDL_VIDEODRIVER=x11 ...
```

All SDL implementations support the driver `dummy`. This allows for using XPCE *headless*. This is typically useful for executing build steps that involve XPCE, both on systems that always provide a GUI context (Windows, MacOS) to avoid opening windows and on systems that may not provide a graphical context such as almost all Unix-derived systems.

Today’s Linux systems are migrating from X11 to Wayland. The default used on Linux depends on how SDL was built, more and more favouring Wayland as default. In general, Wayland is faster and more secure. Security on Wayland is better than on X11 because Wayland restricts control of absolute positions of windows, does not allow accessing the display beyond windows owned by the application and restricts grabbing the mouse and keyboard focus. While XPCE works with both drivers, SDL may not honour various requests on Wayland.

SDL also uses drivers for audio and cameras (see chapter 7). These are selected using the environment variables `SDL_AUDIO_DRIVER` and `SDL_CAMERA_DRIVER`. The `dummy` drivers provide no devices. The `disk` audio driver plays to and records from files, which is useful for testing. Note that SDL copies the environment when it is initialised, i.e., when `pce` is loaded, so these variables must be set before that.

Notes on XPCE for MS-Windows

B

XPCE runs on Windows 10 and later. Its functionality is very close to the non-Windows versions, making applications source-code compatible between the two platforms.

XPCE does not build on top of the hosting window-systems GUI library. Instead, it uses SDL for windows and events, Cairo for graphics and Pango for text 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 SDL video driver on Windows is named `windows` (see appendix A).

The look-and-feel of XPCE may be tailored using the user's *Defaults* file. See section 11. The system colours (`sys_*`) follow the Windows light or dark mode and the accent colour.

B.1 Currently unsupported features in the Windows 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.
- *Cut/Paste*
  Supported for exchanging text.
- *Drag-and-drop*
  Files and text dropped from other applications are supported (see `pce_drop_target`).

B.3 Accessing Windows Graphics Resources

XPCE on Windows defines the same cursor, colour and font-names as the other versions to guarantee portability. Cursors are mapped to the native system cursors by SDL and fonts are selected by

Pango from the fonts installed on the system. It is desirable to have access to all the native Windows colours. This allows the application to maintain better look-and-feel compatibility to other Windows applications. Therefore class colour provides access to the Windows system colours.

It is NOT advised to use these colours in your application code directly as this will stop the application to run on the other versions of XPCE. We advice using these colours in the XPCE *Defaults* file only, or use conditional code using the Prolog flag `windows`.

B.4 Accessing Windows Colours

Colours may be created from their X11 names. 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 a *Defaults* file 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. These names are only defined on Windows. Portable applications should use the `sys_` colour names, which XPCE maps to the corresponding Windows colours as described in section 14.10.1.

B.5 Unicode text on Windows

XPCE represents characters internally as 32-bit integers on all platforms. On most platforms this is the C type `wchar_t`. On Windows however `wchar_t` is a 16 bit unsigned integer and Windows uses UTF-16 encoding to represent Unicode code points above 0xffff. XPCE converts between UTF-16 and its internal representation when exchanging text with the operating system. As a result, code points above 0xffff are supported on Windows.

win_3ddkshadow	Dark shadow for three-dimensional display elements.
win_3dface	
win_btnface	Face color for three-dimensional display elements.
win_3dhilight	
win_3dhighlight	
win_btnhilight	
win_btnhighlight	Highlight color for three-dimensional display elements
win_3dlight	Light color for three-dimensional display elements
win_3dshadow	
win_btnshadow	Shadow color for three-dimensional display elements
win_activeborder	Active window border.
win_activecaption	Active window title bar.
win_appworkspace	Background color of MDI applications.
win_background	
win_desktop	Desktop.
win_btntext	Text on push buttons.
win_captiontext	Text in caption, size box, and scroll bar arrow box.
win_gradientactivecaption	Right side color of an active window's title bar.
win_activecaption	specifies the left side color.
win_gradientinactivecaption	Right side color of an inactive window's title bar.
win_inactivecaption	specifies the left side color.
win_graytext	Grayed (disabled) text.
win_highlight	Item(s) selected in a control.
win_highlighttext	Text of item(s) selected in a control.
win_hotlight	Color for a hot-tracked item.
win_inactiveborder	Inactive window border.
win_inactivecaption	Inactive window caption.
win_inactivecaptiontext	Color of text in an inactive caption.
win_infobk	Background color for tooltip controls.
win_infotext	Text color for tooltip controls.
win_menu	Menu background.
win_menutext	Text in menus.
win_scrollbar	Scroll bar gray area.
win_window	Window background.
win_windowframe	Window frame.
win_windowtext	Text in windows

Table B.1: Windows colour 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.

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

[1] We will mix the terms *instance* and *object* freely in this document. They are considered synonyms.

[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)

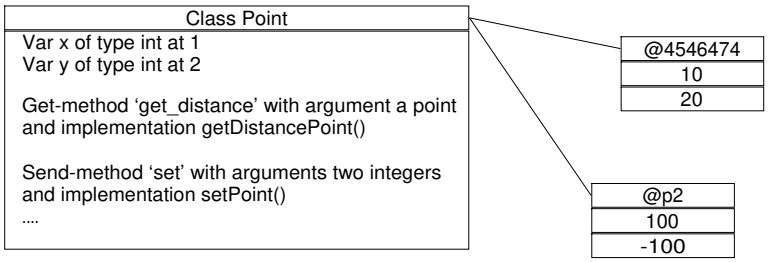


Figure C.1: Classes and Objects in XPCE

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/[2-8]` and `get/[3-8]` 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 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

[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 9.5.2.

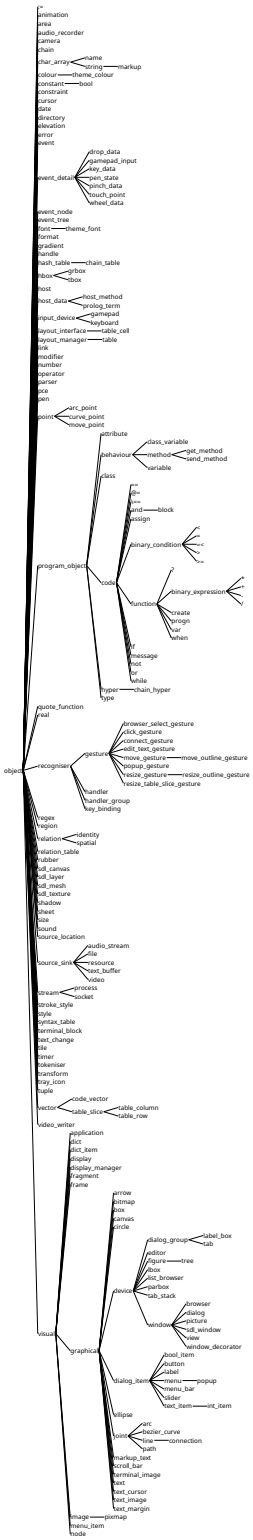


Figure C.2: XPCE's Class hierarchy

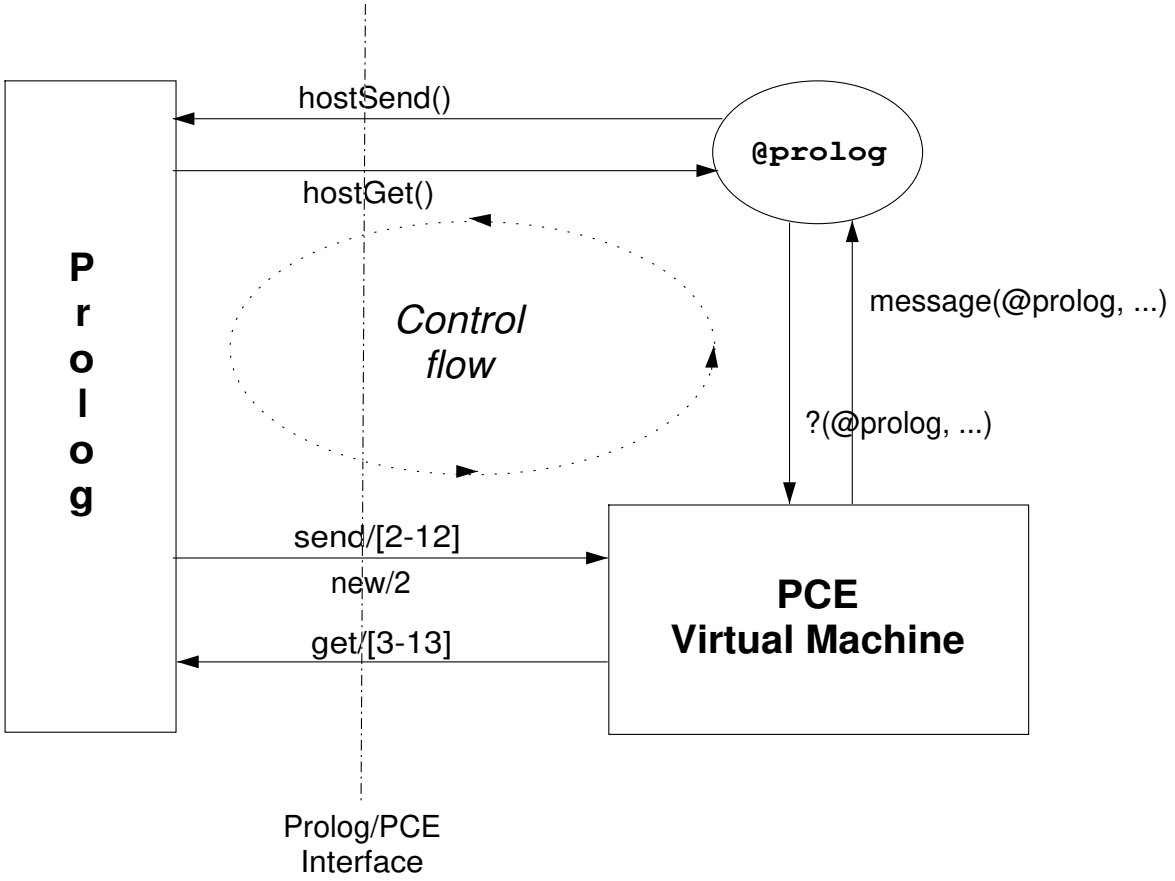


Figure C.3: Data and Control flow in XPCE/Prolog

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 8. 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.

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 14.2 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 XPCE. 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 using `:- use_module(library(pce))` or autoloaded by SWI-Prolog if one of its predicates is used.

new(*?Reference, +TermDescription*)
Create a XPCE object from *TermDescription* and either unify an anonymous reference (e.g. ⟨*pce*⟩`(0x7f8a12c400,box)`) with *Reference* or give the new object the provided named reference (e.g. @my_diagram). An anonymous reference is an opaque handle: it can be passed to `send/2` and `get/3`, compared with ==/2 and used as a key, but it is not a term to take apart. See section E.2. 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 if they are in the range $\pm 2^{51}$ (see '`pce ← max_integer`'). Larger integers are translated into floating point numbers.
- *float*
  Prolog floats (and rational numbers) are translated into floating point numbers. Like integers, these are not objects. XPCE uses the least significant bit of the double as a tag, which implies that some precision is lost.
- `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/[2-8]` 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/[2-8]`. 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 = <pce>(0x7f8a12c400,size)
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/[2-8]` and `get_super/[3-8]`.

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 9 for further discussion on defining classes and methods.

object(+*Reference*)

Succeeds if *Reference* is an object reference and denotes an existing object. Fails silently otherwise. A reference Prolog holds cannot go stale: the object may be destroyed with `free/1` or →`free`, but the handle then reports that it is gone rather than coming to denote whatever was created in its place.

```
1 ?- new(P, point(100,100)).
P = <pce>(0x7f8a12c400,point).
2 ?- new(P, point(100,100)), free(P), object(P).
false.
3 ?- new(P, point(100,100)), free(P).
P = <pce>(freed).
```

This was not so as long as an anonymous reference was the object's address written as `@Integer`: after the object was freed and its memory reused, `object/1` would succeed and denote a different object. See also section E.2, section 16 and section 14.3.3.

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 8 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 implemented for class `text_buffer` (reading and writing) and `char_array` (reading only). Classes that delegate to a `text_buffer`, such as `view` and `editor`, can be used as well. 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(Stream),
...
```

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 predicate allows the application to handle errors occurring 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 occurs 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 14.8.

D.1.1 Declaration of required library predicates

The `require/1` directive declares that a module needs the named library predicates without specifying the library that defines them. It was introduced when XPCE was connected to several Prolog systems with different library structures. With SWI-Prolog, library predicates are normally autoloaded and modern code often uses `use_module/2` or `autoload/2`. The `require/1` directive remains supported and is used by many of the XPCE libraries. 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 in SWI-Prolog.

  Note the command `pce_insert_require_directive` (use `M-x`) 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. As SWI-Prolog autoloads library predicates, this is the same as `call/1`. It is provided for compatibility.

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/[2-8]`. Each of the arguments is either as accepted by `send/[2-8]` 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(colour(green))
                  ]),
```

get_object(*+Receiver, +Selector, +Argument..., -Result*)
Equivalent to `get/[3-8]`, but instead of unifying a variable with a reference the variable is unified with the term-description. The arguments are unified as in `get/[3-8]`. 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. When *Chain* is given, *List* holds its elements at the moment of the call. This is safe if other threads use or modify the chain.

member_chain(*?Element, +Chain*)
True when *Element* is an element of *Chain*. On backtracking it enumerates the elements in order from a snapshot of the chain taken at the first call, so changes to the chain while enumerating do not affect the result, except that elements that are freed before they are reached are skipped. Unlike walking the chain using chain←`next`, this is safe if other threads use or modify the chain. Fails if *Chain* is @nil.

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 16 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, standard Prolog requires additional brackets:

```
1 ?- tracepce((graphical ->selected)).
```

SWI-Prolog accepts the term without the additional brackets. In SWI-Prolog (<-) is a predefined infix operator, so get-methods can be specified the same way.

notracepce(*+Class* <-|->|- *Selector*)
Disables trace-point set with `tracepce/1`.

checkpce

Collect all global (named-) objects and run '`object → _check`' on them. This performs various consistency checks on the objects and prints diagnostic messages if something is wrong. '`object → _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                <pce>(0x7f8a12c400,modifier)
        condition               @nil/constant
        status                  inactive
        cursor                  @default/constant
        drag_scroll             @nil/constant
        drag_scroll_timer       @nil/constant
        drag_scroll_event       @nil/constant
        offset                  <pce>(0x7f8a12c5d0,point)
```

A graphical tool for inspecting instance variables is described in section 16.5.

D.2.3 Accessing the XPCE manual

manpce

Start the XPCE online manual tools. This opens a small GUI console providing access to the manual, demo programs and tools described in this manual. In the SWI-Prolog console window these tools are also available from the XPCE menu. See chapter 3.

manpce(+*Spec*)

As `manpce/0`, but immediately opens 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 XPCE.

E.1 Lifetime of an object

Object lifetime management is a difficult issue in XPCE/Prolog as objects may be referenced from other XPCE objects as well as from 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 XPCE 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.

XPCE distinguishes between *global-*, *top level-*, *support-*, *argument-* and *answer-* objects. *Global* objects are created and exist for the entire XPCE session: @prolog, class objects, etc. *Top-level* objects are the principal objects of the application. They should exist even if no other XPCE 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.

XPCE maintains the following information on objects to support garbage collection. This information may be requested using the XPCE inspector (see section 16.5).

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_manager, 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 XPCE's garbage collector. The lock flag is automatically set on any object that has a named reference. Locking is not needed to keep an object alive while Prolog holds a reference to it, also not if this reference is stored in the Prolog database (see section E.2). See also section 8.

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

XPCE 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 XPCE assumes the object is garbage and removes the object from the object base. A reference held by Prolog counts: see section E.2.

E.2 References held by Prolog

An anonymous object reference passed to Prolog is a *blob*: an opaque handle written as ⟨*pce*⟩`(0xADDR,ClassName)`. Holding such a handle is a reference to the object, exactly like a slot of another object referring to it, and it is counted as one. Consequently an object that Prolog can still reach is never garbage collected, and an object that Prolog can no longer reach is reclaimed once SWI-Prolog's atom garbage collector discovers the handle has become unreachable.

Two properties follow, and both are relied upon:

- A reference can not dangle. `→free` on an object Prolog still refers to marks the object destroyed but keeps its administration alive, so a later use reports a clean error rather than silently denoting whatever object was allocated in its place.
- Objects created and dropped in a loop do not accumulate.

Named references (`@name`) are unchanged: they are Prolog atoms inside the term `@/1`, and a named object is locked, so it lives until it is explicitly freed.

Do not take a reference apart with `Obj = @Ref`: that only matches a named reference. Use `is_object_reference/1` to test a term, and `object_reference/2` and `object_from_reference/2` for the name of a named object. An anonymous object has no identity besides its handle: keep the handle, compare it with ==/2 and index on it.

XPCE identifies an anonymous object internally by its address, which `object\get{object_reference}` reports, but that number is not a reference and cannot be turned back into an object. It was one until handles replaced it, written `@`*`Integer`*, and it was the reason a reference could go stale.

E.2.1 Porting code that inspects references

An anonymous reference used to be `@Integer`, a compound term, so it answered the Prolog type tests the same way a named reference does. A handle does not, and the difference is easy to overlook because the two forms now disagree:

[1] XPCE 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 lose their 'answer' status.

	`@name`	anonymous
`atom/1`	no	no
`atomic/1`	no	**yes**
`compound/1`	yes	**no**

The trap is `atomic/1`. Code that asks "is this a name or an object?" by testing `atomic/1` now answers "name" for an anonymous object. Where the question is really "is this a name", use `atom/1`, which is false for both kinds of reference. Where it is "is this a plain value rather than an object", use `atomic/1` together with `is_object_reference/1`:

```
(   atomic(X),
    \+ is_object_reference(X)
->  ...                          % a name, number or string
;   ...                          % an object
)
```

The same applies to matching a reference by its shape. A clause head such as `emit([@Ref|T], ...)` no longer sees anonymous references; test with `is_object_reference/1` instead.

Finally, an object reference has never been convertible to text by writing it, and that has not changed: `format('~w', [Obj])` yields the reference, not the object's contents. Ask the object, for example with `get(Obj, value, Text)` for a `string`. Code that wrote a reference into text and read it back used to produce a term that parsed but meant nothing; it now produces a syntax error.

E.3 Practical considerations

The principal predicates `new/2`, `send/[2-8]` and `get/[3-8]` 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/[3-8]`.

An object created by `new/2` with an anonymous object reference needs no attention: Prolog holds a reference on it, so it survives as long as it is reachable and is reclaimed when it is not. Neither locking it nor '`object → done`' is required, and '`object → done`' has no effect while Prolog can still reach the object. Use `free/1` to destroy an object at a known moment rather than leaving it to the garbage collector.

> Before the introduction of blob references this was not so: an object returned to Prolog stayed on the answer stack forever unless it was attached, locked or explicitly finished with '`object → done`'. Code written against that model still works, but the precautions are no longer needed.

E.4 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 24 bytes on a 64-bit machine. Each instance variable consumes an additional 8 bytes. For example a point object has 'x' and 'y' instance variables and thus consumes $24 + 2 * 8 = 40$ 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).
S = 184
```

But a box has an ←area instance variable consuming an additional 56 bytes.

XPCE status and plans

F

XPCE has been used by many projects and has been sold as a commercial product. In addition it has been used to implement the SWI-Prolog development tools, the PceEmacs editor, a source level debugger, profiler front end and many more tools. Over the years, interest for using XPCE for building GUIs faded away because the system was barely maintained and claimed as *deprecated*.

XPCE 7 is a major rewrite of the lower layers of XPCE. It makes XPCE future proof by building on widely supported portable libraries rather than only providing bindings for X11 and Windows. XPCE 8 and 9 modernised the appearance of the system and its development tools. Some major changes compared to XPCE 6 are:

- Fonts, colours, cursors and images have been updated, removing many of the old legacy X11 features. Fonts are managed by Pango. Colours are RGBA values and no longer accept images (pixmaps).

- Colours may follow a *theme*. The themes switch at runtime and, by default, follow the light or dark appearance of the desktop. The `sys_*` colours provide the colours of the desktop.

- The dialog items have a flat, themed look and dialogs, menus and popups can be operated from the keyboard.

- Images are loaded using SDL_image, which supports the common image formats as well as SVG. Most bundled images are SVG or PNG files. '`image` → `save`' only saves in PNG format. While images represent pixels using the 32-bit RGBA model, images can still be assigned the type `bitmap`. Such images should only have black, white and transparent pixels. Black pixels are rendered using the current foreground, white pixels using the current background and transparent pixels have no effect.

- While XPCE 6 supported primarily integers, XPCE 7 generalizes non-object numbers to floating point numbers. In many, but not all, classes floats are properly passed to the Cairo library which expresses all coordinates as doubles. The current system supports a type `num` to represent such numbers. Currently, `int` is an alias for `num`. Passing floats to methods that require a strict integer causes silent rounding to an integer. Most methods and libraries that used to accept a `real` now use `num`.

- Host file dialogs are supported through SDL using the '`frame` ← `open_file`' and '`frame` ← `save_file`' methods. The native xpce *finder* still works, but does not use the host functionality.

- The '→`fill`' method used to fill graphicals accepts a solid colour, a `gradient` or an `image`, which is repeated to fill the area.

- Where XPCE 6 provided access to multiple X11 displays, each of which could be divided into one or more monitors, XPCE 7 removed class `monitor`. The system exposes one or more `display` instance as members of `display_manager`. Unlike XPCE 6, XPCE 7 deals with hot plugging, removing and modifying the display configuration. @display refers to the current display.
- Generating PostScript from graphicals has been removed. The methods '`graphical` → `pdf`' and '`window` → `pdf`' use Cairo to write PDF to a file. The implementation is rather minimal. Notably producing multi-page documents is not yet supported.
- There is no support for printing. The above mentioned PDF methods allow for generating PDF from graphicals that can then be printed.
- Files and text dropped from external applications are delivered as events (see `pce_drop_target`). Dragging objects from XPCE to external applications is not supported.
- Support for Windows meta-files has been removed.
- The C and C++ embedding API (classes `c` and `c_pointer`) has been removed.
- The development tools are *panes* that can be placed in tabs or beside each other in the windows of the development environment (see `pane_frame`).
- The reference manual sources are markdown files. The online manual renders them as HTML.

F.1 The future for XPCE

The underlying libraries allow for many extensions. Notably SDL provides APIs for dealing with video, audio and many input devices used for gaming. Cairo supports a *transformation matrix*, which is available for scaling, rotating and skewing the contents of a `figure` using a `transform`. Partial transparency (*alpha channel*) is supported in `colour` and through '`graphical` → `opacity`'. Transformations are not (yet) supported for other graphicals.

Glossary



Attribute
: An `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 XPCE *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 11.

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 mathematical 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/[3-8]`, when an obtainer is executed or from XPCE's built-in functionality.

GUI
: Abbreviation for Graphical User Interface.

Inheritance
: The sharing of definition from a super-class. When a XPCE *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 used 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. XPCE defines both get- and send-methods. If a *send-operation* is invoked on an object, XPCE 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 a named reference is `@Atom` and an anonymous one is an opaque handle, written ⟨*pce*⟩`(Address,Class)`.

Object
: An *object* is an entity in XPCE'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 a *send-method* on some object. Started by the Prolog predicate `send/[2-8]`, when a *message* is executed or from XPCE's built-in functionality.

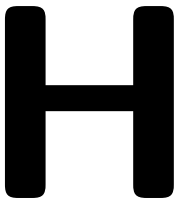
Slot
: Equivalent to *instance_variable*.

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 of a message), @event (currently processed event object).

Class summary descriptions



This appendix provides an overview of the 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, value*)

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, right*)
\==(*left, right*)

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, selector, argument ...*)

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, object*)

Class @= assigns a symbolic reference name to the argument object. It is used to define global objects in the class-variable pce.initialise. The following example from its built-in value creates the objects @_dialog_bg and @_win_pen.

```
pce.initialise: \
        and(_dialog_bg @= colour(ui_dialog_background), \
            _win_pen   @= number(0), \
            ...)
```

object <program_object−code−and chain

and(*test ...*)

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−animation

animation(*file*)

An animation holds the frames of an animated image file (PNG, GIF, WebP, AVIF) as `images`, with the time each frame is shown. A `bitmap` shows and plays it:

```
?- new(B, bitmap(animation('spinner.png'))),
   send(B, play).
```

object−visual−application

application(*name*)
An application object is a `visual` object used to combine multiple `frames`. See section 14.5 for a discussion on its usage.

object ← visual−graphical−joint−arc
layout_interface

arc(*radius, start_angle, size_angle*)
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−point−arc_point

arc_point(*x, y, rx, ry, rotation, large, sweep*)
A `point` of a `path` that is reached by an elliptical arc, as the 'A' command of an SVG path.

object−area

area(*x, y, width, height*)
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, wing*)
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 directly to create fancy arrows.

```
?- new(L, line(0, 0, 100, 50, second)).
```

object−program_object−code−assign

assign(*var, value, scope*)
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, value*)

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−audio_recorder

audio_recorder(*level_message*)

An audio_recorder records from the default audio input, normally the microphone, into its ←sound or to a file or other source_sink, or into the sound of a video_writer. While recording, its ←level_message is executed regularly with the input level, e.g., to show a level meter.

object−source_sink−audio_stream

audio_stream(*format, frequency, channels*)

An audio_stream plays audio while it is written, for example using pce_open/3: raw samples or a sound file such as MP3 or FLAC that is decoded on the fly. Its ←low_water_message asks for more data.

object−program_object−behaviour

behaviour(*name, context*)

Super class of method and variable, representing the two types of objects that can realise behaviour in classes. Not useful for the application programmer.

object ← visual−graphical−joint−bezier_curve / layout_interface

bezier_curve(*start, end, control1, control2*)

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.

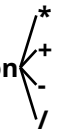
object−program_object−code−binary_condition ← <, =, =<, >, >=

binary_condition(*left, right*)
<(*left, right*)
=(*left, right*)
=<(*left, right*)
>(*left, right*)
>=(*left, right*)

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),
...,
```

---

object−program_object−code−function−binary_expression 

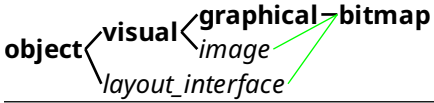
---

binary_expression(*left, right*)
⋆(*left, right*)
+(*left, right*)
−(*left, right*)
/(*left, right*)

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),
...
```

---

object 

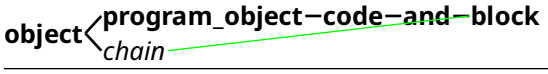
---

bitmap(*image*)

A bitmap turns an `image` into a graphical object that can be displayed on a `device`. Transparency is defined by the alpha channel of the image. A bitmap can also show and play an `animation`.

```
?- new(I, image('pce.svg')),
   new(B, bitmap(I)).
```

---

object 

---

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).
```

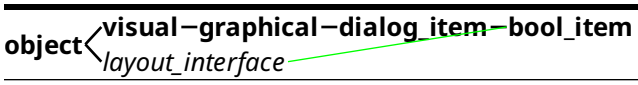
```
hello world
```

object−constant−bool

bool(*self*)

Class `bool` defines two instances: @on and @off, representing 'true' and 'false'. The user cannot create instances of this class.

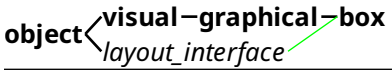
```
...,
send(Text, transparent, @on)
...
```



bool_item(*label, selection, message*)

Controller that edits a boolean value using a *switch*: a track with a knob that slides to the other side when clicked. The message is called with @on or @off. For a set of boolean options, a `menu` with `marked` items and '`menu → multiple_selection`' is an alternative.

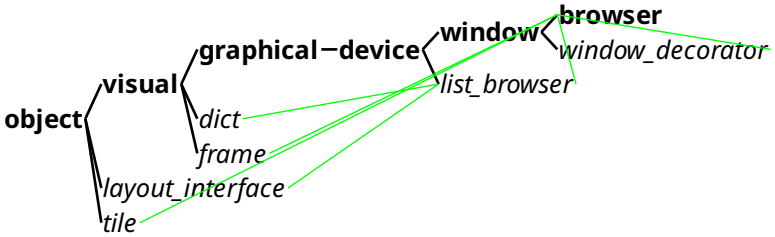
```
...,
send(Dialog, append,
     bool_item(dark_mode, @off,
               message(@prolog, set_dark_mode, @arg1))),
...
```



box(*width, height*)

Graphical representing a rectangle. Corners can be rounded, the interior can be filled with a `colour` or `gradient`, the texture and thickness of the line can be controlled and a drop `shadow` can be defined.

```
?- new(B, box(100, 50)),
   send(B, radius, 10),
   send(B, shadow, 5).
```



browser(*name*)

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(*label, message, accelerator*)
A button is a push-button controller. It has an associated message that is executed if the button is activated using the mouse or the keyboard. Inside a dialog, one button can be assigned as 'default' button. 'button → danger' marks a button that starts a destructive action.

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

object−camera

camera(*name*)
A camera delivers video from a camera as an image: each frame is copied into its ←image, so a bitmap that shows this image shows the video, and its ←message is executed. If its ←texture is an sdl_texture, the frames go there instead, without Cairo, and its ←writer, a video_writer, records them. 'display ← cameras' lists the available cameras.

object ← visual−graphical−canvas
layout_interface

canvas(*width, height*)
A canvas is a graphical to draw on using operations modelled after Cairo: build a path using 'canvas → move_to', 'canvas → line_to', 'canvas → curve_to', 'canvas → arc', etc. and draw it using 'canvas →stroke' or 'canvas → fill' with the current source, line width and transformation. The drawing is recorded as vectors, so the canvas redraws itself and remains sharp when scaled or written as PDF or SVG.

object−chain

chain(*member ...*)
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 and member_chain/2 enumerates the elements of a chain. Class chain 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()
Link two objects with a 'chain'. If either dies, the other will die with it. See also the library hyper and section 14.11.

object−hash_table−chain_table

chain_table(*buckets*)
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(*value*)
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*)
Equivalent to an ellipse with the same ←width and ←height. Not used frequently.

object−program_object−class

class(*name, super_class_name*)
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 9.

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(*context, name, default, type, summary*)

A `class_variable` can be used to describe class properties as well as to provide access to the XPCE `Defaults` database. See chapter 11. 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, modifier, multiclick, execute_message, preview_message, cancel_message*)

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 14.2.

```
?- send(message(@prolog, format, 'Hello ~w.', @arg1),
        forward, world).
Hello world.
```

object−vector−code_vector

code_vector(*element ...*)

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 9.5.2.

object−colour

colour(*name*)

A `colour` represents an 'RGBA' value: the intensities of the primary colours 'red', 'green' and 'blue' and the opacity ('alpha'). Colours are created from a name, which is a CSS or X11 colour name, a system colour (see section 14.10.1) or a hexadecimal specification such as `#ff000080`, or from the numeric components using the `rgb` or `hsv` model. Colours are used as attributes to graphicals, windows and `styles`. See also `theme_colour` and `gradient`.

```
?- new(C, colour(@default, 255, 0, 0, 128)),
   get(C, name, Name).
Name = '#ff000080'.
```

object−recogniser−gesture−connect_gesture

connect_gesture(*button, modifier, 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 with the graphical at which the connection should start. The demo program PceDraw exploits connections and connect_gestures.

object ← visual−graphical−joint−line−connection / layout_interface

connection(*from, to, link, from_handle, to_handle*)

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(*self*)

A `constant` is a unique handle. XPCE predefines the following constants: @nil, @default, and from the subclass `bool`, @on and @off. The user 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, to, relation*)

A constraint is a relation between 2 objects that has to be maintained if either of the 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, argument ...*)

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*)

A cursor defines the shape that indicates the position of the pointer (also called mouse). The system provides the standard cursors of the platform using CSS-like names such as default, pointer, text, move, wait and ew_resize. For compatibility, the most commonly used old X11 cursor names are mapped to these. Cursors can also be created from an image and a hot spot. 'cursor ←kind' tells whether the cursor is a standard, x11 or image cursor. 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, pointer),
...
```

---

object−point−curve_point

---

curve_point(*x, y, control1, control2*)
A `point` of a `path` that is reached by a quadratic or cubic Bezier curve.

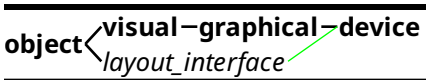
---

object−date

---

date(*second, minute, hour, day, month, year*)
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.

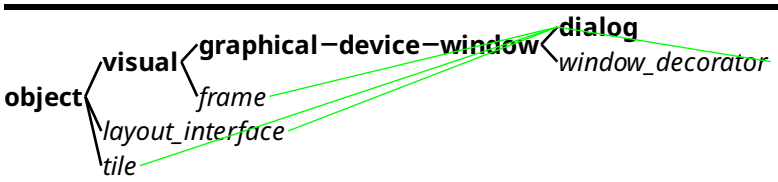
```
?- send(@pce, format, 'It is now "%s"\n',
        new(date)?string).
It is now "Tue Jan 30 14:07:05 1996"
```



---

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 14.12. 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(*name*)
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 →appending 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 14.7) and keyboard accelerators, handling the default button and moving the keyboard focus between the controllers.

```
:- pce_autoload(file_item, library(file_item)).

edit_file_dialog :-
        new(D, dialog('Edit File')),
        send(D, append,
             new(F, file_item(edit_file, ''))),
        send(D, append,
             button(edit, message(@prolog, edit, F?selection))),
        send(D, append,
             button(cancel, message(D, destroy))),
        send(D, open).
```

---

object ← **visual−graphical−device−dialog_group**
layout_interface

dialog_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(*label*)

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()

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, label, object*)

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(*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(*name*)

A `display` represents a monitor attached to the computer. Display objects are created by the system for each monitor that is found; their ←`area` and ←`work_area` describe the position and size of the monitor in the desktop. @display is a function that evaluates to the current display: the display of the last event or, if there is none, the primary display. The display implements a number of global operations: getting the screen ←`size`, showing modal message boxes using →`inform` and →`confirm`, accessing the clipboard and the selection, finding the light or dark ←`system_theme` of the desktop, etc.

```
?- get(@display, size, size(W, H)).
W = 1920, H = 1080.
```

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. After the theme or system colours change, '`display_manager → colours_changed`' notifies all frames and graphicals.

object−event_detail−drop_data

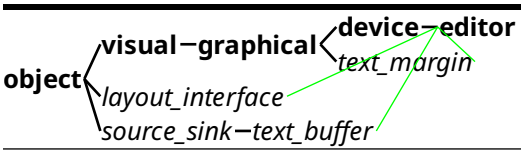
drop_data(*path, text, source*)

The '`event ← detail`' of the drop events that are generated when files or text are dropped on a window from another application. The event delegates to it, so '`event ← path`' and '`event ← text`' give the dropped file or text.

object−recogniser−gesture−edit_text_gesture

edit_text_gesture(*button, modifier*)

Used internally to handle selection inside a `text` object. See also the library `pce_editable_text`.



editor(*text_buffer, width, height, margin_width*)

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.

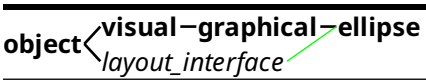
Editors can handle sensitive regions, different fonts, colours and attributes using `fragment` objects. All text windows in XPCE's demo programs (PceEmacs, cards from the online help, application help, etc.) either use class view or class editor to display the text.

object−elevation

elevation(*name, height, colour, relief, shadow*)

An `elevation` object describes an elevated region on the screen. Elevations come in two flavours ('`elevation ← kind`'): `3d`, using light and dark edges, and `shadow`. The elevation object itself just contains the colour definitions. The actual painting is left to the graphical object the elevation is attached to.

Some controllers handle elevations, although most have a flat, themed look. The only general purpose graphical supporting an elevation is `figure`. For a drop shadow around a graphical, see class `shadow`.



ellipse(*width, height*)

Elliptical shape. Class `ellipse` defines similar attributes as `box`: pen, texture, fill and shadow. See also `circle`.

object−error

error(*id, format*)

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 14.7. 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 16.

object ← event
event_detail

event(*receiver, name, position*)

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 SDL, the library that connects XPCE to the window system. 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_detail

event_detail()

Super class of the objects in '`event ← detail`', which hold data that is specific to the device that generated an event, such as the pressure of a pen.

object−event_node

event_node(*value, parent*)

An `event_node` is a node in the event 'is_a' hierarchy. See the demo program EventHierarchy. 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 '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, an `elevation` or `shadow` and a clipping region. A figure may have a `transform`, which rotates, scales or shears its contents (see '`figure → rotate`', '`figure → scale`' and the demo Transform). 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(*name*)

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.plx'), size, Size).
```

object−font

font(*family, style, points*)

A font is a reusable object that describes the typeface of text. Fonts are rendered using the Pango library. Section 14.9 documents the specification of physical and logical fonts.

```
...,
send(Text, font, bold),
...
```

object−format

format(*direction, width, columns*)

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_buffer, start, length, style*)

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, kind, display*)

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 14.6 describes the layout of windows inside a frame. 'frame ← open_file' and 'frame ← save_file' prompt for a file using the dialog of the operating system. The libraries tab_frame and pane_frame build windows that hold tabs and panes on top of class frame; see section 12.

```
...,
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 14.2.2.

object−input_device−gamepad

gamepad(*kind, id, name*)

A `gamepad` is the `input_device` of a game controller. '`display ← gamepads`' enables gamepads and returns the connected ones. Their buttons and axes generate gamepad_* events for the window with the keyboard focus. A gamepad can also be polled using ←`button` and ←`axis`, and may rumble (→`rumble`) and show a colour (→`led`).

object−event_detail−gamepad_input

gamepad_input(*control, value*)

The '`event ← detail`' of gamepad events: the name of the button or axis and its value. The event delegates to it, so '`event ← control`' gives the name of the button.

object−recogniser−gesture

gesture(*button, 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−gradient

gradient(*kind*)

A `gradient` is a linear or radial colour gradient that can be used instead of a `colour` to fill a graphical such as a `box`, `ellipse` or `path` ('`box → fill`') or as the background of a `figure`. A linear gradient runs from point *p0* to *p1*; a radial gradient from a circle around *p0* with radius *r0* to a circle around *p1* with radius *r1*. The points use the coordinate system of the device on which the graphical is displayed or, if *units* is `bounding_box`, are fractions of the box around the shape, so the gradient moves and scales with it. The colours are given as a `chain` of `tuple(Fraction, Colour)` stops, where *Fraction* is between 0 and 1.

```
?- new(B, box(100, 50)),
   send(B, fill,
        gradient(linear, point(0, 0), point(100, 0),
                 stops := chain(tuple(0, white),
                                tuple(1, steel_blue)))).
```

object−program_object−behaviour−method−get_method

get_method(*name, return_type, types, summary*)
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 9.

object ← visual−graphical, layout_interface

graphical(*x, y, width, height*)
The most generic graphical object. This class defines generic geometry management, display, update, event-handling, opacity (see section 5.3), etc. 'graphical → pdf' and 'graphical → svg' write the graphical as a PDF or SVG file. This class can be sub-classed to define specialised graphics. See section 14.12.

object ← hbox−grbox, visual−graphical, layout_interface

grbox(*graphical*)
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 15.9.

object−handle

handle(*x_position, y_position, kind, 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(w, h/2, link, east))).
:- pce_global(@west_handle,
              new(handle(0, h/2, link, west))).
```

object−recogniser−handler

handler(*event, message, 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 ...*)

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*)

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. The optional second argument determines whether the table protects its keys and/or values from being garbage collected.

```
?- new(@ht, hash_table),
   send(@ht, append, gnu, 'GNU is Not Unix').

?- get(@ht, member, gnu, Expansion).
Expansion = 'GNU is Not Unix'.
```

object−hbox

hbox(*width, ascent, descent, rubber*)

Superclass of `tbox` and `grbox` dealing with document-rendering. Instances of `hbox` itself can be used to define 'rubber'. See section 15.9 for details.

object−host

host(*system*)

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 8.

```
?- send(@prolog, write, hello).
hello
```

object−host_data

host_data(*handle*)

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 8.2.

object−program_object−hyper

hyper(*from, to, 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 14.11.

object−relation−identity

identity(*from, to*)

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, then, else*)

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*)

An `image` is a two-dimensional array of pixels. Images come in two flavours ('`image ← kind`'): `pixmap`, the default, where each pixel is a colour with an alpha channel, and `bitmap`, where each pixel represents a boolean that is painted in the current foreground colour. Images are loaded using the SDL_image library, which supports the common formats such as PNG, JPEG, GIF and SVG. '`image → save`' saves an image to a PNG file. Images are displayed on a graphical device using a `bitmap`. They are also used to specify `cursor` objects, the icons of `menu_items` and the icon associated with a 'frame'. See section 14.10.

object−input_device

input_device(*kind, id, name*)

An `input_device` describes the device that generated an event, see '`event ← device`'. Events from the mouse refer to @mouse_device and keyboard events to @keyboard_device.

Pens, touch screens and gamepads are represented by their own `input_device`. 'display ←`input_devices`' lists the known devices.

---

object ← **visual−graphical−dialog_item−text_item−int_item**
layout_interface

int_item(*name, default, message*)
Subclass of `text_item` for entering integer values. Has stepper buttons for incrementing and decrementing the value.

---

object ← **visual−graphical−joint**
layout_interface

joint(*arrows*)
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*)
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−event_detail−key_data

key_data(*key, repeat*)
The '`event` ← `detail`' of keyboard events and of the `key_up` event generated when a key is released. '`event` ← `key`' of a key_up is the key of the key-down it releases and '`event` ← `repeat`' tells whether a key-down is auto-repeated.

---

object−input_device−keyboard

keyboard(*kind, id, name*)
The `input_device` @keyboard_device. Keyboard events refer to it and '`keyboard` ← `key_down`' tells whether a key is held down, which games use to poll the keyboard.

---

object ← **visual−graphical−dialog_item−label**
layout_interface

label(*name, selection, font*)
A `label` is a controller used to display read-only text or `image`. Labels can handle →`report` messages. See section 14.7. 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()
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(*image*)

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()
Class of the document-rendering system to render a list environment, a sequence of labels and text. See section 15.9 for details.

object ⟨ **visual−graphical−joint−line**
layout_interface

line(*start_x, start_y, end_x, end_y, arrows*)
A `line` is a straight line-segment with optional arrows, thickness and texture. Class `path` implements a 'multi-line'.

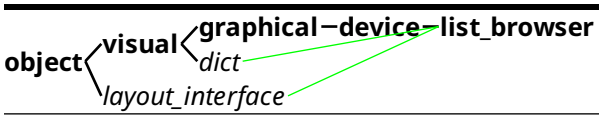
object ⟨ **link**
visual−graphical−joint−line
layout_interface

link(*from, to, line*)
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, width, height*)

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. A list_browser can be operated using the keyboard; typing searches the items incrementally. The hook '`list_browser` → `selection_changed`' is called after the selection changed and may be redefined in a subclass.

```
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−char_array−string−markup

---

markup(*value*)

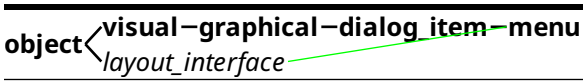
A `markup` is a `string` holding Pango markup. Formatting escapes the arguments. A `label` renders a markup selection, which also applies to messages from '`object` → `report`'.



---

markup_text(*markup, width*)

A `markup_text` displays read-only text formatted using Pango markup, mixing fonts, sizes, styles and colours, for example "`Some ⟨b⟩bold</b> text`". Pango wraps, aligns, justifies and ellipsizes the lines. See also `text` for editable text.



---

menu(*label, kind, message*)

Class `menu` realises various different styles of menus and is the super-class for `popup`. Basically, a menu presents multiple values and allows the user to choose one or more items.

'menu → kind' defines only the presentation: cycle shows the selection in a combo box, marked shows the items with a radio button or check box, choice shows a segmented control and popup is used for the items of a popup menu. Whether the user may select one or more items is defined by 'menu → multiple_selection'. The kind toggle is an old name for a marked menu with multiple selection. 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]),
new(S, menu(style, marked)),
send(S, multiple_selection, @on),
send_list(S, append, [bold, italic, underline]),
...,
```

---

object ← **visual** ← **graphical** ← **dialog_item** ← **menu_bar**
layout_interface

---

menu_bar(*label*)

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. On MacOS, 'menu_bar ← native' tells whether the menu bar is shown in the native menu bar at the top of the screen.

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

initialise(F) :->
        send_super(F, initialise, 'My Application'),
        send(F, append, new(MBD, dialog)),
        send(new(B, browser), below, MBD),
        send(new(_V, view), right, B),
        send(MBD, append, new(MB, menu_bar)),
        send(MB, append, new(File, popup(file))),
        send(MB, append, new(_Edit, popup(edit))),
        send_list(File, append,
                  [ menu_item(load,
                              message(F, load)),
                    ...
```

---

object ← **visual** ← **menu_item**

---

menu_item(*value, message, label, end_group, condition, accelerator, mnemonic*)

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, selector, argument ...*)

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 14.2 and many examples in this chapter.

object−program_object−behaviour−method

method(*name, types, message, summary, source*)

Class `method` is the super-class of `send_method` and `get_method`. Instances of this class itself are useless.

object−modifier

modifier(*shift, control, meta, gui*)

A `modifier` is a reusable object that defines a condition on the status of the four 'modifier keys' shift, control, meta/alt and gui, the Windows key or the Apple command key. 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`, `m` and `g` into a modifier that requires the shift, control, meta and/or gui key to be down and 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, 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, recogniser, new(move_gesture)),
...
```

object−recogniser−gesture−move_gesture−move_outline_gesture

move_outline_gesture(*button, 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−point−move_point

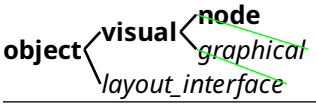
move_point(*x, y, closed*)

A `point` of a `path` that is reached without drawing, starting a new sub-path.

object−char_array−name

name(*value*)

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.



node(*image*)

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(*argument*)

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*)

A `number` is the object version of an integer (int). It may be used as a storage bin. To compute the widest graphical of a device:

```
widest_graphical(Device, Width) :-
        new(N, number(0)),
        send(Device?graphicals, for_all,
             message(N, maximum, @arg1?width)),
        get(N, value, Width),
        send(N, done).
```

object

object()

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, priority, kind*)
Part of XPCE's object parser. Not (yet) available to the application programmer.

object ← program_object−code−or (chain)

or(*test ...*)
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, alignment*)
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 15.9.

object ← parser (tokeniser)

parser(*tokeniser*)
Part of XPCE's object parser. Not (yet) available to the application programmer.

object ← visual−graphical−joint−path (layout_interface)

path(*kind, radius_or_interval*)
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. A `poly` path may also contain a `curve_point`, `arc_point` or `move_point`, making it a Bezier curve, elliptical arc or new sub-path to that point. This allows for the same shapes as SVG paths, including shapes with holes (see '`path → fill_rule`').

```
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(*version*)
Class `pce` defines a single instance called @pce. Actions that cannot sensibly be related to a particular object are often defined on class `pce`.

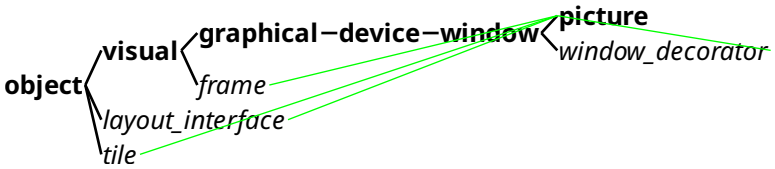
```
?- get(@pce, user, User).
User = jan
```

object−pen

pen(*thickness, texture, colour*)
Reserved for future usage.

object−event_detail−pen_state

pen_state(*pressure, xtilt, ytilt, hover_distance, barrel_rotation, slider, tangential_pressure, eraser, barrel_buttons*)
The ‘`event ← detail`’ of events from a pen on a tablet or touch screen, holding the pressure, tilt and other axes of the pen and its barrel buttons. The event delegates to it, so ‘`event ← pressure`’ gives the pressure of the tip.



picture(*name*)
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−event_detail−pinch_data

pinch_data(*scale*)
The ‘`event ← detail`’ of the pinch_begin, pinch_update and pinch_end events of a track-pad zoom gesture. The event delegates to it, so ‘`event ← scale`’ gives the change of scale since the previous update.

object−visual−image−pixmap

pixmap(*source*)
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. As images are colour images (←`kind pixmap`) by default, this class is rarely needed.

object−point

point(*x, y*)
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),
...
```

object ← visual−graphical−dialog_item−menu−popup
layout_interface

popup(*name, message*)

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, button, 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(*name*)

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 grep and similar commands of *PceEmacs*, the ispell program and the chess front-end. Interactive programs such as a shell are better run in a terminal_image. See also socket.

object−program_object−code−function−progn

progn(*statement ...*)

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 `spypce/1`.

object ← quote_function, program_object−code−function

quote_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*)

A `real` is XPCE's notion of a floating-point number. Reals are represented using a C double-precision 'double'. Reals define the same operation as class `number`, its integer equivalent. Most methods that accept a floating point number use the type `num`, which accepts both integers and floats without creating a `real` object.

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*)

A `regex` is XPCE's encapsulation of Henry Spencer's Regular Expression library as also used by Tcl. Regular expression form a powerful mechanism for the analysis of text. By default, the *advanced* syntax is used; '`regex ← syntax`' also supports the POSIX `basic` and `extended` syntax. 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, y, width, height*)

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()

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, 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, 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, button, 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 15.5.

object−source_sink−resource

resource(*name, class*)

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.png')).

show_splash_screen :-
        new(I, image(resource(splash))),
        get(I, size, Size),
        new(W, window(splash, Size)),
        send(W, display, bitmap(I)),
        send(W, open_centered),
        send(timer(2), delay),
        send(W, destroy).
```

object−rubber

rubber(*stretch, shrink*)

Defines how elastic objects in the document-rendering system such as `hbox` and its subclasses are. Used by `parbox` to realise layout. See section 15.9 for a full description of the rendering primitives.

object ← visual−graphical−scroll_bar, layout_interface

scroll_bar(*object, orientation, message*)

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−sdl_canvas

sdl_canvas(*size*)

An `sdl_canvas` holds `sdl_layer` objects as an `sdl_window` does, but draws them into a texture of its own rather than on the screen. A layer shows the canvas as it shows an `sdl_texture`, e.g., to move many sprites as one, for a minimap or for trails.

object−sdl_layer

sdl_layer(*texture, area, scale*)

An `sdl_layer` shows an `sdl_texture` in an `sdl_window`, scaled to an area or as a *sprite* that is moved, rotated and scaled using a `transform`. The renderer draws it without Cairo, below the graphicals of the window.

object−sdl_mesh

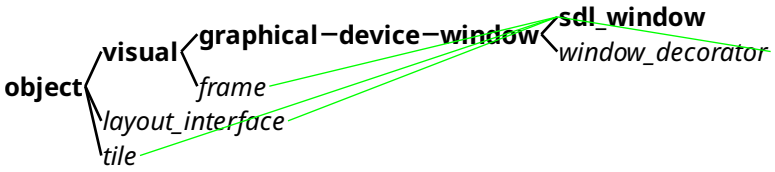
sdl_mesh()

An `sdl_mesh` holds triangles with coloured corners that an `sdl_layer` draws, possibly mapping its texture onto them. It is used for gradients, warping a texture and many particles in a single layer. `pce_sdl_mesh_set/3` replaces all triangles in one call.

object−sdl_texture

sdl_texture()

An `sdl_texture` holds pixels for the SDL renderer in their own format, e.g., the frames of a `camera` or `video`. It can be made from an `image` or drawn from a graphical, e.g., to make a sprite.



sdl_window(*name*)

A `window` that shows `sdl_layer` objects below its graphicals. It is the place to show video and fast moving sprites.

object−program_object−behaviour−method−send_method

send_method(*name, types, message, summary, source*)

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 preprocessor as described in chapter 9.

object−shadow

shadow(*x_offset, y_offset, blur, colour, spread*)

A `shadow` describes a CSS-like drop shadow that is painted outside a `box`, `ellipse`, `circle` or `figure` using '`graphical → shadow`'. It defines the offset of the shadow, the width of the blurred edge, the colour (normally with an alpha value) and how much larger than the shape the shadow is. An integer *N* is converted into a soft shadow at distance *N* to the bottom-right. Popup menus use a shadow as defined by the class variable `popup.shadow`.

```
?- new(B, box(100, 50)),
   send(B, radius, 8),
   send(B, shadow,
        shadow(0, 4, 12, colour(@default, 0, 0, 0, 70))).
```

object−sheet

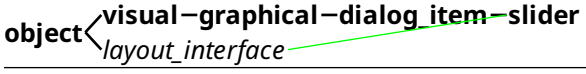
sheet(*member ...*)

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, height*)

Combination of ←`width` and ←`height` used to communicate with graphical objects about dimension. See also `point` and `area`.



slider(*label, low, high, selection, message*)

Controller for a numeric value inside a range that does not require exact values. Specifying

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*)
Communication end-point for a TCP/IP or Unix-domain interprocess communication stream. XPCE supports both 'server' and 'client' sockets. On Windows, only TCP/IP sockets are provided. The `pce_server` provides a good starting point for defining server sockets. See also PceEmacs server mode as defined in `'emacs/server'`. Note that SWI-Prolog's own socket and HTTP libraries are often a better choice.

object−sound

sound(*file*)
A `sound` holds audio, loaded from a WAV file or recorded using an `audio_recorder`. '`sound` → `play`' plays it on the default audio output; its ←`message` is executed when the play is finished.

object−source_location

source_location(*file_name, line_no*)
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 loading image-data. Notably a `resource` can be used for creating images (see section 13) and a `text_buffer` can be used to hold image data in memory. Images can only be saved to a `file`.

object−relation−spatial

spatial(*x_from, y_from, x_to, y_to, w_to, h_to*)
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/2, yref=y+h,
                                w2=w+10, h2=h+10)))
```

object−stream

stream()

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−stroke_style

stroke_style(*cap, join, dash, paint***)**

A stroke_style describes how lines are drawn: the shape of their ends and corners, an arbitrary dash pattern and a colour or gradient to paint them with. It is used as the texture of a graphical ('graphical → texture').

object−char_array−string

string(*value***)**

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, font***)**

A style defines attributes for a text fragment as handled by a editor/text_buffer or a dict_item as handled by a list_browser/dict. It defines the font, fore- and background colours as well as underlining (optionally in a colour), strike-through, 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*)

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_table → keywords`' registers the keywords of a language. Syntax tables are introduced to support the implementation of modes in PceEmacs. See also the `emacs_begin_mode/5` directive as defined in `emacs_extend`.

object < visual−graphical−device−dialog_group−tab
layout_interface

tab()

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 ...*)

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(*image*)
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()
table_row()
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, style*)
Add text using a defined `style` to a `parbox`. Part of the document-rendering infra-structure. See section 15.9.

object−relation_table

relation_table(*names, keys*)
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−terminal_block

terminal_block(*terminal*)
A `terminal_block` represents the prompt, command and output of a single command in a `terminal_image`. The blocks are created by the terminal from the OSC 133 *semantic prompt* marks that are emitted by the program that runs in the terminal, such as Prolog or a shell. They are available through '`terminal_image ← blocks`' and '`terminal_image ← block_at`'. Blocks can be selected, copied, folded to hide their output and removed.

object<**visual**−**graphical**−**terminal_image**
layout_interface

terminal_image(*width, height*)
A `graphical` that represents a terminal (console). It is capable of providing I/O for Prolog threads or external processes. On Windows systems I/O is provided by Windows named pipes opened in *overlapped* mode or a pseudo console. On POSIX systems *pseudo terminals* (pty) are used as interface between tasks and the terminal image. The `terminal_image` input handles a rich set of the ANSI escape sequences and various extensions, including mouse reporting, hyperlinks, the window title and the working directory of the client. The terminal supports incremental search ('terminal_image ←`find`'), selecting and copying text, and splits its contents into `terminal_blocks`, one for each command, that can be folded. This class is used to build the *Epilog* Prolog console provided by `swipl-win` and `epilog`.

object<**visual**−**graphical**−**text**
layout_interface

text(*string, format, font*)
Graphical representing a string in a specified font. Class text defines various multi-line and wrapping/scrolling options. 'text →`style`' associates a `style`, for example to underline or strike through the text. It also implements methods for selecting and editing text. Class `editable_text` as defined in `pce_editable_text` exploits these methods to arrive at a flexible editable text object.

object−**source_sink**−**text_buffer**

text_buffer()
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(*style*)
Cursor (caret) as displayed by an `editor`. Its style is one of `bar`, `block`, `underline`, `xpce` or `image` and it may blink. 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, width, height*)
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(*label, value, message*)

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. '`text_item → placeholder`' defines a hint that is shown while the item is empty. 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, width, height*)

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. PceEmacs uses this to show, e.g., bookmarks and diagnostics. 'text_margin →`icon_size`' defines the size of the icons.

---

object−colour−theme_colour

---

theme_colour(*name, derived_from*)

A `theme_colour` is a subclass of `colour` whose value follows the active theme. It is created from a name that describes its role and the colour it is derived from: the name of another colour, a system colour, another theme colour, or a colour object. The RGB value is computed when needed by following '`theme_colour ← derived_from`'. XPCE itself uses the `ui_`*role* theme colours such as `ui_window_background` and `ui_accent`. Applications should use theme colours for colours that must adapt to a light or dark theme. See section 14.10.1 and `pce_theme`.

```
?- new(_, theme_colour(myapp_highlight, khaki1)),
   new(T, text(hello)),
   send(T, colour, myapp_highlight).
```

---

object−tile

---

tile(*object*)

Tiles are used to realise the 'tile-layout' of windows in a `frame`. Section 14.6 explains this in detail. Tiles can also be used to realise tabular layout of other re-sizable graphical objects.

object−timer

timer(*interval, message*)

Timers are used to initiate messages at regular intervals, schedule a single message or a fixed number of messages 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))),
send(T, start),
...
```

object−tokeniser

tokeniser(*source, syntax*)

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.

object−event_detail−touch_point

touch_point(*finger, pressure, dx, dy*)

The '`event ← detail`' of events from a touch screen. The first finger that touches the screen generates the ms_left_* events, the others the touch_* events. The event delegates to it, so '`event ← finger`' identifies the finger.

object−transform

transform(*xx, xy, yx, yy, tx, ty*)

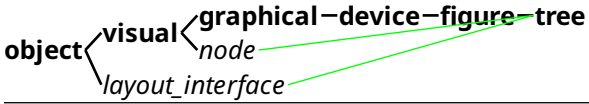
A `transform` is a 2D affine transformation (rotation, scaling, shearing and translation). It is used by '`figure ⇆ transform`' to transform the contents of a `figure`. Events are mapped through the inverse transform, so the transformed graphicals remain sensitive to the mouse. Usually the transform is manipulated using 'figure →`translate`', '`figure → rotate`', '`figure → scale`' and '`figure → shear`'. See the demo Transform.

```
?- new(F, figure),
   send(F, display, box(100, 50)),
   send(F, rotate, 30).
```

object−tray_icon

tray_icon(*image, tooltip, popup*)

A `tray_icon` shows an icon in the system tray (notification area), with an optional tooltip and a menu built from a `popup`. Selecting an entry executes the popup with the `tray_icon` as context.



tree(*root*)

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, second*)

Anonymous tuple of two objects. Commonly used by get-methods that have to return two values.

object−program_object−type

type(*name, kind, context, supers*)

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 9.5.1.

```
?- get(type(int), check, '42', X). X = 42
```

object−program_object−code−function−var

var(*_value*)

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, type, access*)

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 9.

object−vector

vector(*element ...*)

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.

object−source_sink−video

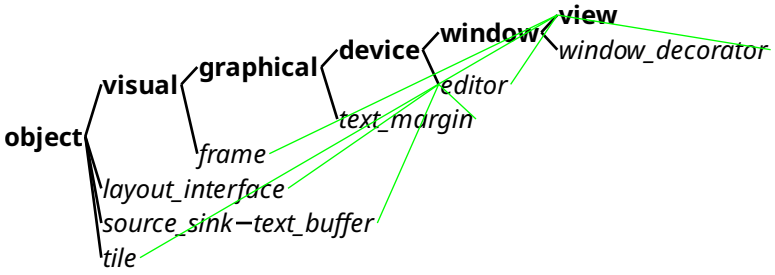
video(*source*)

A video plays a video file, a network stream or encoded data written to it, with its sound. Its frames go into an sdl_texture, shown using an sdl_layer.

object−video_writer

video_writer(*target, size*)

A `video_writer` encodes images, windows, a `camera` and the microphone into a video file, streams them live to a URL or gives the encoded data to code, e.g., to write it to a Prolog stream.



view(*name*)

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()

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 14.7 and defines '`visual → destroy`' to ensure destruction of a sub-tree of the visual consists-of hierarchy.

object−event_detail−wheel_data

wheel_data(*dx, dy, flipped*)

The '`event ← detail`' of the wheel events, holding the precise scroll distance in both directions as reported by trackpads. The event delegates to it, so '`event ← dx`' and '`event ← dy`' give the distance.

object−program_object−code−function−when

when(*condition, then, else*)

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.

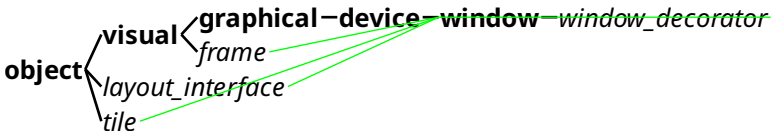
```
frame.geometry: \
        when(@display?width > 2000, \
             '1200x800', \
             '800x600')
```

object−program_object−code−while

while(*condition, body*)

Code statement executing *body* as long as *condition* executes successfully. Not used frequently.

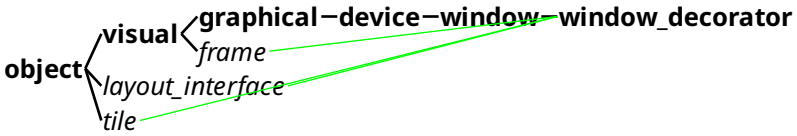
Most iteration in XPCE uses the →for_all, →for_some, ←find and ←find_all methods defines on most collection classes.



window(*name*)

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 14.6. 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 → pdf' and 'window → svg' write the entire contents of the window as a PDF or SVG file.



window_decorator(*window*)

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*, 266
@= *class*, 233
@display
  →dispatch, 102
  →inform, 124
@dragbox_recogniser, 188
@nil, 187
@on, 33
@pce
  ←application_data, 86
  ←home, 86
  ←last_error, 128, 219
  →exception_handlers, 129
  →exception, 129
  →format, 123
@prolog, 31, 32
@receiver, 32
+*class*, 68
?*class*, 13, 14, 106, 232

absolute_file_name/[2
  3], 97
adaptive_colour/3, 143
adaptive_colour/3, 142
alpha, 45
alt, 257
and *class*, 106, 236, 273
animation *class*, 236
annotations, 246
ansi_format/3, 139
append_style/3, 33
application
  ←member, 117
  →append, 117
  →delete, 117
application *class*, 117, 118
apply_theme/1, 142
apply_theme/1, 141
arc, 43
arc *class*, 38, 234, 253
arc_point *class*, 43, 259
area
  ←size, 223
  →set, 9
area *class*, 260, 265
argument
  named, 9
  optional, 9
  type, 17
arguments
  typed, 10
arithmetic, 107
arrangements_file/1, 94
arrow *class*, 38, 234, 253
ask questions, 30
ask_style/3, 33
assert/1, 104
assert_employee/5, 26
assoc
  redefined, 129
  undefined, 129
atom, 8, 258
atom/1, 225
atomic/1, 225
attribute
  editing, 33
attribute *class*, 229
audio, 57
  driver, 205
audio_recorder *class*, 53, 57, 235, 266
audio_stream *class*, 235
auto_call/1, 219
autoload/2, 219
autoloading, 219
axis, 173

balloon, 157
bar
  →drag_message, 179
  →initialise, 178
  →message, 178
  →range, 178
  →value, 178
bar *class*, 177–179

bar_button_group
←bar, 179
→initialise, 179
bar_button_group *class*, 179
bar_chart
⇆value, 178
→append, 177
→clear, 178
→delete, 177
→event, 178
→initialise, 177
→selection, 178
→select, 178
→sort, 178
bar_chart *class*, 177, 178
bar_group
→initialise, 179
bar_group *class*, 177
bar_stack
←selection, 178
→initialise, 179
bar_stack *class*, 177, 179
barchart, 173, 176
behaviour
⇆group, 73
bezier_curve *class*, 38
binary relation, 252
bitmap *class*, 20, 38, 58, 144, 233, 238, 252
blend, 48
blinking, 272
block *class*, 236, 273
BMP
file format, 143
bool *class*, 237, 241
bool_item *class*, 27
box
→colour, 11
→fill_offset, 46
→fill, 249
box *class*, 22, 38, 39, 43, 45, 80, 108, 152, 178, 192, 246, 249, 265
branch, 252
break, 201
browser *class*, 13, 22, 27, 237, 242, 244, 255, 275
bug
reporting, 2
bullet_list *class*, 195
button
→danger, 27, 238
button *class*, 13, 27, 28, 101, 105
button-bar, 159

C++, 9
Cairo, 50
call-back, 105
call/1, 219
camera, 58
⇆message, 58
⇆status_message, 58
⇆writer, 61
←image, 58
←status, 58
→sdl_test_frame, 65
→close, 58
→open, 58
driver, 205
camera *class*, 53, 58, 133, 238, 265, 274
canonical *class*, 194
canvas, 50, 60
→arc, 50, 238
→clear, 51
→clip, 50
→curve_to, 50, 238
→draw_in, 50
→fill_rule, 50
→fill, 50, 238
→image, 50
→line_to, 50, 238
→line_width, 50
→move_to, 50, 238
→paint, 50
→path, 50
→rectangle, 50
→restore_state, 50
→rotate, 50
→save_state, 50
→scale, 50
→source, 50
→stroke_style, 50
→stroke, 50
→text, 50
→translate, 50
canvas *class*, 43, 50, 238
catch

errors, 128
catch-all, 21
cell
⇆note_mark, 169
chain
→for_all, 105, 242, 252
→initialise, 81
→sort, 262
chain *class*, 13, 68, 105, 108, 155, 165, 214, 235, 238, 244, 249, 273
chain_list/2, 220
chain_list/2, 221, 238
char_array *class*, 18, 218, 239, 263, 267
chart, 173
check_theme/1, 143
check_theme/1, 141
checkpce/0, 76, 200, 201, 222
chess *demo*, 261
circle *class*, 38, 43, 80, 246, 265
class
←class_variable, 86
←instance_size, 226
←instance, 215
←un_answer, 224
→get_method, 250
hierarchy, 15
structure, 15
undefined, 129
class *class*, 72, 83, 239
class variable
editing, 16
class_variable
⇆value, 87
class_variable *class*, 86, 240
class_variable/[3
4], 72
class_variable
←type, 87
class_variable_editor/[0
1], 86
ClassBrowser *tool*, 86
click_gesture *class*, 42, 55, 240
clip, 48
clipboard, 63
clipping, 247
close/1, 218
code
→forward, 105, 106, 240, 273
code *class*, 67, 70, 178, 179, 224, 229, 240, 249, 257, 259, 262
code object, 105
code_block *class*, 194
code_vector *class*, 68, 241
colour
←rgba, 133
images, 133
system, 133
colour *class*, 20, 39, 45, 133, 139, 228, 237, 241, 249, 267, 271
combo-box, 27
completion, 19, 271
compound graphical, 243
compound/1, 225
connect_gesture *class*, 39, 40, 42, 241, 254
connection *class*, 39–41, 145, 241, 253, 254
consists-of, 16
constant *class*, 72, 87, 241
constraint
←relation, 242
vs. method, 78
constraint *class*, 78, 252, 263
constructor, 9, 20
control, 257
structure, 101
controller, 244
controllers, 25, 244
built-in, 26
copy, 63
create *class*, 108
create_theme/2, 143
current_theme/1, 143
cursor, 144
←kind, 144
cursor *class*, 20, 144, 242, 252
Cursors *demo*, 242
curve, 43
curve_point *class*, 43, 259
curves, 259
cut-and-paste, 206

dark theme, 138
dash, 46
date *class*, 243
DDE, 206

default arguments, 9
default/3, 80
default_arrangement/1, 93
define_style/1, 32
definition_list *class*, 195
delegate_to/1, 72
delegate_to/1, 81
delegation, 210
demo programs, 16
destructor, 20
device
  ⇆layout_manager, 168
  ←graphicals, 239
  ←member, 113, 117
  ←pointed, 77
  ←selection, 105
  →layout_dialog, 167
  →append_dialog_item, 28
  →event, 149, 169
  →format, 38, 167
  →geometry, 149
  →initialise, 149
  →layout_dialog, 29
  →layout_dialog, 29
device *class*, 38, 39, 52, 77, 108, 149, 150, 152, 153, 167, 168, 174, 176, 181, 191, 196, 236, 243, 246–248, 254, 268, 275
device_added, 53
device_removed, 53
diagram, 173
dialog
  ←gap, 29
  →_compute_desired_size, 29
  →append, 28, 29
  →apply, 34
  →gap, 28
  →layout, 29, 30
  →restore, 34
  layout in, 28
  resizing, 30
dialog *class*, 13, 25, 26, 29, 123, 159, 160, 167, 238, 243, 244, 253, 255, 268, 275
dialog_group
  ←label_name, 183
dialog_group *class*, 27, 29, 30, 167, 179, 183, 244, 254, 268
dialog_item
  ⇆label_format, 29
  ⇆label_width, 29
  ⇆reference, 28
  ⇆value_width, 29
  ←label_name, 183
  →above, 28
  →alignment, 28
  →apply, 34
  →below, 28
  →default, 34
  →left, 28
  →restore, 34
  →right, 28
dialog_item *class*, 13, 25, 26, 77, 144, 183, 244, 268
dialog_item
  ←alignment, 29
  ←label_name, 183
  →alignment, 30
  →label, 144
  →value_width, 29
  built-in types, 26
dict *class*, 13, 238, 244, 255, 267
dict_item *class*, 13, 157, 187, 238, 242, 244, 255, 267
dict_item
  ⇆object, 245
  ⇆style, 245
dictionary, 244
direct-manipulation, 3
directory *class*, 13, 164, 245, 248
disk usage, 179
display
  ←battery_percent, 64
  ←cameras, 58, 238
  ←gamepads, 56, 249
  ←power_state, 64
  ←sdl_test_gamepad, 65
  ←selection_targets, 63
  ←selection, 63
  ←sound_file_extensions, 57
  ←theme, 138
  ←video_decoders, 62
  ←video_encoders, 62
  →sdl_test_gamepad, 64
  →copy, 52, 63
  →dispatch, 103, 247

→report, 124
display *class*, 117, 123, 228, 245
display_manager *class*, 228
display_manager
→colours_changed, 140, 142, 245
→system_colours_changed, 137
doc/emit *library*, 192
doc/html *library*, 197
doc/objects *library*, 193
doc/vfont *library*, 194
doc:action/3, 194
doc_mode
→colour, 194
→initialise, 193
→set_font, 194
→underline, 194
doc_mode *class*, 192, 195
doc_table *class*, 196
doc_mode
←link_colour, 196
←space_mode, 192
double, 262
drag-and-drop, 206
files, 189
drag_and_drop_dict_item_gesture *class*, 187
drag_and_drop_gesture
→initialise, 188
drag_and_drop_gesture *class*, 187–189
dragbox, 188
→event, 188
dragdict *library*, 187
dragdropdemo/0, 188
drop shadow, 265
drop_data *class*, 54, 189
drop_picture, 187
→drop, 187
→preview_drop, 187

edit/1, 92
editable_text *class*, 270
editable_graphical
→event, 82
editor
→margin_width, 271
→style, 248
editor *class*, 18, 19, 27, 79, 144, 210, 218, 238, 246, 248, 253, 263, 264, 267, 270, 271, 274
elevation
←kind, 246
elevation *class*, 246, 247
ellipse *class*, 38, 45, 246, 249, 265
emacs_extend *library*, 268
emacs_pane *class*, 90
emacs_begin_mode/5, 268
emit/3, 192–195, 197, 198
enum_list *class*, 195
epilog *library*, 270
error
←feedback, 128
description of, 127
error *class*, 127, 183, 246
Error Browser *tool*, 246
errors
browsing, 16
catching, 128
reporting, 159
event
←control, 56, 249
←detail, 245, 247, 249, 253, 260, 272, 274
←device, 53, 252
←dx, 54, 274
←dy, 54, 274
←finger, 55, 272
←key, 53, 253
←path, 189, 245
←position, 54
←pressure, 53, 260
←receiver, 78, 123
←repeat, 54, 253
←rotation, 54
←scale, 56, 260
←text, 189, 245
←value, 56
←x, 54
←y, 54
←detail, 54
→is_a, 54, 247
→post, 77, 78, 247
device, 53
processing, 41
event *class*, 53, 247
event-driven, 101

event_detail *class*, 53
event_node *class*, 247
EventHierarchy *demo*, 247
exception, 129
exception-handling, 110
executable
    object, 105

figure
    ⇆transform, 272
    →blend, 49
    →clip_path, 49
    →rotate, 49, 247, 272
    →scale, 49, 247, 272
    →shear, 49, 272
    →status, 248
    →transform, 48, 59
    →translate, 49
figure *class*, 38, 39, 45, 48, 52, 56, 228, 243, 246, 247, 249, 265, 272, 273
file
    ←object, 129
    OS file dialog, 155
    prompting for, 155
file *class*, 13, 97, 164, 189, 248, 266
file_search_path/2, 98
fill_menu/1, 113
fill_rule, 44
find_file *library*, 155
find_file_dialog *class*, 155
find_file_dialog *library*, 155
finder
    ←file, 155
finder *class*, 154, 155
finding
    graphical, 9
    graphicals, 113
finger, 55
floating point, 8
folder, 245
font *class*, 10, 20, 131, 132, 168, 191, 194, 248, 256
forget_arrangements/0, 93
format *class*, 38, 167, 168, 248
fragment *class*, 144, 246, 248, 267, 270
frame
    ←confirm_centered, 33, 118
    ←confirm, 30, 102, 103
    ←image, 65
    ←name, 117
    ←open_file, 125, 155, 227, 248
    ←save_file, 155, 227, 248
    ←sdl_test_image, 65
    →sdl_test_event, 64
    →application, 117
    →delete, 121
    →fit, 29, 121
    →label, 195
    →modal, 117, 118
    →open, 30
    →report, 123
    →return, 30, 33
    →shortcut, 91
    →status, 117
    →transient_for, 117
    →transient_for, 33
    layout, 119
frame *class*, 11, 90, 117, 119, 121, 123, 234, 244, 248, 253, 271, 275
free/1, 9, 14, 110, 216, 217, 223, 225
function, 68
function *class*, 17, 34, 68, 106, 240, 249, 262, 273

game controller, 56
gamepad, 56
    ←axis, 56
    ←button, 56
    →led, 57
    →rumble, 56
gamepad *class*, 56, 249
gamepad_input *class*, 54, 56
garbage collection, 223
gesture *class*, 41, 242, 249, 250, 257, 262
get/3, 8–11, 75, 214, 216
get/[3-13], 239
get/[3-8], 10, 14, 86, 210, 220, 225, 229
get/[4-8], 75
get_chain/3, 221
get_class/3, 216
get_class/4, 75
get_implementation/4, 83
get_method *class*, 257, 273
get_object/[3-8], 220
get_super/3, 75, 217

get_class/4, 75, 217
get_super/3, 75
get_super/[3-8], 75, 217
GIF
    file format, 143
global objects, 16
    with reconsult, 110
GNU-Emacs, 246
goal_expansion/2, 11
GPU, 59
gradient, 45
    ←extend, 45
    ←units, 45
gradient *class*, 45, 133, 227, 237, 241, 249, 267
graph, 39
graphical
    ⇆layout_interface, 168
    ←colour, 46, 47
    ←hor_stretch, 28
    ←pen, 46
    ←texture, 46
    →_redraw_area, 79, 149
    →_redraw_area, 149
    →alert, 124
    →compute, 79, 84
    →cursor, 144
    →draw_markup, 52
    →draw_path, 52
    →draw, 150
    →drop, 189
    →event, 77, 78, 110, 250
    →geometry, 78
    →opacity, 49, 228
    →pdf, 52, 153, 228, 250
    →preview_drop, 189
    →preview_drop, 187
    →recogniser, 110, 247
    →request_geometry, 78
    →shadow, 265
    →svg, 52, 153, 250
    →texture, 267
    finding, 9, 113
    name, 113
    window, 36
graphical *class*, 10, 20, 22, 39, 63, 76, 77, 144, 157, 187, 191, 234, 243, 244, 257, 258, 270, 272
graphics tablet, 54
grbox *class*, 191, 250, 251
gui, 257
GUI *menu*, 15
GUI/Inspect GUI hierarchy *menu*, 201
GUI/Inspect XPCE object *menu*, 202

handle
    ←kind, 241
handle *class*, 39, 40, 74, 241, 250, 254
handle/[3
    4], 72
handler *class*, 41, 42, 250, 262, 263
handler_group *class*, 42, 251
hash_table *class*, 239, 244, 251
hbox *class*, 191–193, 251, 264
headless, 205
help
    on manual, 16
help/1, 139
help_message *library*, 154, 157
hierarchy
    of UI components, 201
hierarchy_chart
    →click_message, 181
    →extent, 181
    →focus_message, 181
    →focus_node, 181
    →hover_message, 181
    →initialise, 181
    →max_depth, 181
    →reference, 181
    →root, 181
    →select_message, 181
    →unit, 181
hierarchy_chart *class*, 181
hierarchy_from_term/2, 180
hierarchy_node
    ←path_label, 180
    ←total, 180
    →append, 180
    →colour, 180
    →initialise, 180
hierarchy_node *class*, 180
hierarchy_segment *class*, 181
hierarchy_window
    ←chart, 182

←popup_node, 182
→initialise, 181
→view_message, 182
→view, 182
hierarchy_window *class*, 181
home
web, 2
host
←messages, 102
←message, 102
→call_back, 102
→catch_all, 102
host *class*, 14, 102, 251
host-language, 251
host_data *class*, 70
hyper
→initialise, 147
→unlink_from, 147
→unlink_to, 147
hyper *class*, 79, 145, 242, 252
hyper *library*, 145, 239

ICO
file format, 143
icon *class*, 167
identity *class*, 242, 252, 263
if *class*, 106, 232, 235, 252, 274
image
←kind, 143, 252
→copy, 59
→draw_in, 59, 142
→hot_spot, 143
→initialise, 143
→save, 143, 227, 252
file formats, 143
image *class*, 10, 20, 38, 45, 58, 59, 63, 133, 144, 159, 183, 227, 233, 236, 238, 242, 252, 253, 256, 260, 265
in_pce_thread/1, 85
in_pce_thread_sync/1, 85
inheritance
multiple, 210
of classes, 209
initialisation
failed, 129
input device, 53
input_device *class*, 53, 249, 252, 253
inspecting instances, 16
inspector, 202
Inspector *tool*, 248
int, 8, 258
int_item *class*, 27
interpolated, 259
is_object_reference/1, 224, 225
ispell *demo*, 261

joint
→arrows, 234
joint *class*, 234, 253
JPEG
file format, 143

key_binding *class*, 42, 246, 250, 253, 262
key_data *class*, 54
key_up, 53
keyboard
←key_down, 253
keyboard-accelerators, 253

label
→report, 81, 124
→selection, 144
label *class*, 27, 123, 144, 159, 253, 255
lambda, 240
lambda functions, 105
layout
in dialog, 28
manager, 168
window, 119
layout of graphicals, 248
layout_interface *class*, 269
layout_manager *class*, 254
lbox *class*, 191, 195
line
cap, 46
join, 46
line *class*, 38–40, 234, 253, 254
line-wrapping, 270
link *class*, 40, 241, 254
Linux
Wayland, 205
X11, 205
list_browser *class*, 22, 27, 144, 157, 187, 237, 238, 244, 253, 255, 263, 264, 267, 270
list_browser

→selection_changed, 255
live video, 62
look-and-feel, 2

make/0, 141
make_dragbox_recogniser/1, 188
make_verbose_dialog/0, 69
manpce/0, 1, 2, 15, 201, 222
manpce/1, 21, 222
markup, 47
markup *class*, 157, 255
markup_text *class*, 47, 48, 255
markup_text
←alignment, 47
←ellipsize, 48
←width, 47
member_chain/2, 221
member_chain/2, 238
memory usage, 225
menu
→kind, 27, 256
→multiple_selection, 27, 237, 256
menu *class*, 27, 29, 157, 237, 255, 256
menu_bar *class*, 27, 160, 261
menu_item
←default_label, 183
menu_item *class*, 157, 160, 183, 252, 256, 261
menu_bar
←native, 256
menu_item
⇆accelerator, 253
→label, 144
mesh, 59
message
→execute, 108
message *class*, 13, 14, 26, 42, 102, 105, 106, 229, 238, 257
messages
reporting, 159
meta, 257
method *class*, 230, 235, 257, 266
modal, 103, 117, 118
dialog, 30
modes, 268
modifier *class*, 188, 257
monitor, 245
mouse, 242
move_gesture *class*, 42, 55, 257, 263
move_outline_gesture *class*, 42, 257
move_point *class*, 43–45, 259
moving graphicals, 41
MP3, 57
MP4, 61
MS-Windows, 2
multi-segment line, 259
multiple inheritance, 210

name
←label_name, 159
name *class*, 8, 10, 18, 20, 66, 72, 168, 214, 239, 245, 256–258, 263
named argument, 9
named pipes, 206
new
failed, 129
new/2, 6, 7, 9, 14, 20, 25, 109, 128, 210, 214, 215, 218, 224, 225, 239
node *class*, 165, 210, 273
nospypce/1, 201
notation
in this manual, 11
notification area, 64
notracepce/1, 201, 221
number *class*, 258, 262

object
⇆attribute, 235
⇆slot, 73
←catch_all, 21
←class_variable_value, 86
←class_variable_value, 80
←convert, 21, 76
←get_super, 75
←hypered, 147
←lookup, 20, 77
←slot, 75
→_check, 222
→catch_all, 21
→done, 103, 224, 225
→error, 128, 246
→free, 223
→initialise, 20, 76
→protect, 216, 223
→report, 123, 246, 255

→save_in_file, 72
→send_hyper, 147
→send_super, 75
→slot, 75
→unlink, 20, 76
inspecting, 16, 202
management, 225
memory usage, 225
reference, 6
remove, 9, 20
removing, 223
object *class*, 17, 81, 123, 147, 258, 262
object/1, 217
object/2, 72, 74, 217
object_from_reference/2, 224
object_reference/2, 224
objects
creating, 20
obtainer, 14, 106
once/1, 85, 128
open
icon, 41
object as stream, 218
open_resource/3, 97
open_pane_frame/2, 94
open_resource/3, 97
optional arguments, 9
or *class*, 259

pan_zoom *library*, 56
pane
→handles_prefix, 92
pane *class*, 90
pane_frame
focus_on_enter., 95
inactive_opacity., 95
label_format., 95
prompt_style., 95
pane_frame *class*, 90
pane_frame *library*, 90, 228, 248
pane_handle
handle_size., 96
pane_layouts *library*, 90, 93, 94
pane_menu_bar
menu_order., 95
pane_menu_bar *class*, 90
pane_popup
accelerator_font., 95
pane_stack
pane_side., 95
pane_stack *class*, 90
pane_status_dialog *class*, 90
pane_tab
closable., 95
editable_label., 95
pane_tab *class*, 90
pane_frame
←current_pane, 90
←pane_group, 90
←pane_term, 94
→append_pane, 91
→split_beside, 91
→split, 91
pane_menu_bar
←menu_order, 91
pane_placement/3, 94
Pango, 47
paragraph, 248
parbox
←content, 193
→cdata, 192
parbox *class*, 191, 192, 250, 259, 264, 269
paste, 63
path, 43
⇆event_area, 181
←svg_data, 44
→append, 43
→closed, 45
→fill_rule, 44, 259
→inside, 45
→radius, 45
→svg_data, 44
path *class*, 38, 43, 45, 50, 52, 150, 176, 181, 234, 235, 243, 249, 253, 254, 257, 259
pbox
←anchor, 193, 196
→event, 193
→show, 193
pbox *class*, 192, 193, 195, 196
pce
←exception_handlers, 129
←max_integer, 214
←window_system_driver, 205
→banner, 205

→exception, 129
pce *class*, 259
pce *library*, 65, 86, 205, 214
pce_begin_class/[2
3], 70
pce_catch_error/2, 128, 218
pce_class_directive/1, 72, 83
pce_class_variable_editor *library*, 86
pce_drop_target *library*, 189, 206, 228
pce_editable_text *library*, 245, 270
pce_end_class/0, 73
pce_end_class/1, 73
pce_global/2, 217
pce_group/1, 73
pce_image_browser *library*, 167
pce_open/3, 218
pce_portray *library*, 8
pce_renew *library*, 129
pce_report *library*, 159
pce_sdl_mesh_set/3, 60
pce_server *library*, 266
pce_theme *library*, 138, 142, 271
pce_theme_editor *library*, 138
pce_autoload/2, 129
pce_begin_class/[2
3], 70, 73, 88
pce_catch_error/2, 127, 128, 246
pce_class_directive/1, 72, 83
pce_end_class/0, 70, 73, 88
pce_end_class/1, 73
pce_global/2, 109, 110, 129, 217
pce_group/1, 70, 75
pce_image_directory/1, 87, 98
pce_open/3, 62, 218, 235
pce_sdl_mesh_set/3, 264
pce_thread/1, 103
pcedraw *library*, 2
PceEmacs *tool*, 255, 261, 274
PDF, 52
pen, 54
pen_state *class*, 54, 55
percent-done, 266
picture *class*, 36, 181, 260, 275
pie-chart, 234
pie-slice, 38
pinch, 56
pinch_data *class*, 54
pixmap *class*, 260
plot, 173
plot/axis *library*, 173
plot/barchart *library*, 176
plot/demo *library*, 176
plot/hierarchy_chart *library*, 179
plot/plotter *library*, 174
plot_axis
←label_for_value, 174
←location, 173
←value_from_coordinate, 174
→format, 173
→initialise, 173
→label, 173
→lines, 174
→scale, 174
plot_axis *class*, 173, 175, 176
plot_graph
→append, 176
→initialise, 176
plot_graph *class*, 174, 176
plot_mark *class*, 176
plot_point *class*, 176
plot_axis
←label_for_value, 176
plotter
←hover_label, 176
←hover_value, 176
←translate, 173, 176
←value_from_x, 176
←value_from_y, 176
→axis, 175
→clear, 175
→graph, 175
→hover_values, 176
plotter *class*, 173, 174, 176, 177
PNG
file format, 143
PNM
file format, 143
point *class*, 6, 43, 80, 176, 234, 243, 257, 265
pointer, 242
popup *class*, 27, 64, 160, 165, 255, 256, 261, 272
popup_gesture *class*, 261
portray/1, 8
preferences, 16
print_graphics *class*, 153

print_graphics *library*, 153
process *class*, 261, 267
progn *class*, 262
program
    as object, 105
program_object *class*, 262
prolog-term
    life-time, 69
prolog_ide
    `tool_placement.`, 94
prolog_term *class*, 67, 69, 252
prolog_ide
    ←`tool_placement`, 92
prompt, 30
pulldown, 261
push-button, 27

qsave_program/2, 83
quit/0, 103
quote_function *class*, 262

radio-button, 27
re-usability, 113
read
    text from object, 218
real *class*, 8, 10, 66, 227, 262
recogniser
    →`event`, 78
Recogniser *class*, 41
recogniser *class*, 41, 77, 240, 247, 249–251, 262, 271
recorda/2, 104
redefined
    object reference, 129
reference
    object, 6
    Prolog, 224
regex
    ←`syntax`, 263
regex *class*, 263, 267
region *class*, 263
Regular Expression, 263
relation *class*, 242, 252, 263, 266
relation_table *class*, 269
remove
    objects, 9
repeat
    key, 53
report_dialog *class*, 159
reporter *class*, 159
reporting, 159
require/1, 219
reset_theme_setting/2, 143
resize_gesture *class*, 42, 263
resize_outline_gesture *class*, 42, 263
resource *class*, 97, 98, 143, 263, 266
resource/3, 97
reusability, 20
rings chart, 179
rings_chart *class*, 180, 181
rubber *class*, 191, 192
runtime generation, 83
runtime-system, 127

scroll_bar *class*, 264
sdl_canvas *class*, 60, 264
sdl_layer *class*, 59, 61, 264, 265, 273
sdl_mesh *class*, 59, 60, 264
sdl_texture *class*, 59–61, 238, 264, 265, 273
sdl_window *class*, 59, 60, 65, 264
sdl_layer
    ⇆`blend`, 59
    ⇆`mesh`, 59
    ⇆`offset`, 59
    ⇆`slices`, 59
    ⇆`tint`, 59
    ⇆`transform`, 59
    →`hide`, 59
sdl_mesh
    →`append`, 60
    →`triangle`, 60
sdl_texture, 59
    ⇆`hot_spot`, 59
    ←`image`, 59
    →`draw_in`, 59
sdl_window, 59
    ←`layers`, 59
select_theme/1, 143
select_theme/1, 138
send/2, 7, 8, 10, 11, 75, 214, 215, 220
send/3, 68, 69
send/[2-12], 239
send/[2-8], 10, 14, 25, 72, 80, 87, 88, 210, 215, 216, 219, 225, 230

send/[3-8], 75
send_class/3, 75, 216
send_implementation/3, 83
send_list/2
3, 219
send_method *class*, 242, 257, 265, 273
send_super/2, 75, 217
send_class/3, 75, 217
send_implementation/3, 83
send_list/2, 219
send_super/2, 75
send_super/[2-8], 75, 217
sensitive, 41
set_epilog/1, 142
set_prolog_flag/2, 142
set_style/2, 32
set_theme_setting/2, 143
set_verbose/0, 68
Settings/Edit GUI preferences *menu*, 3
shadow *class*, 237, 246, 247, 265
shared editing, 246
sheet *class*, 64, 265
shift, 257
shift-click, 257
show_slots/1, 222
show_key_bindings/1, 274
show_slots/1, 202
size *class*, 170, 260
slider *class*, 27, 58
smooth, 38
smooth scrolling, 54
socket *class*, 261, 267
sound, 57
⇆message, 57
←peaks, 58
→play, 57, 266
→stop, 57
sound *class*, 53, 57, 266
source_sink *class*, 97, 143, 235, 266
spatial *class*, 41, 236, 242, 263, 266
split_handle
grip_image., 96
split_handle *class*, 91
split_handle_gesture
cursor_border., 96
cursor_size., 96
cursor., 96
sprite, 59
spy, 201
spy/1, 201
spypce/1, 201, 262
statement
as object, 105
stdarg, 81
stream
open Object as, 218
stream *class*, 267
streaming, 62
string
→format, 81, 123
string *class*, 10, 18, 107, 108, 225, 239, 255, 263, 267
stroke_style *class*, 46, 47, 267
stroke_style, 46
style
←icon, 271
→icon, 144
style *class*, 9, 238, 241, 248, 267, 269–271
stylus, 54
sub-path, 44
sub-pixel, 54
subpixel_events, 54
summary line, 16
sunburst, 179
SVG, 52
file format, 143
path, 43
svg_data, 44
syntax_table *class*, 270
syntax_table
→keywords, 268
system colours, 133

tab
edit_width., 95
→icon, 144
tab *class*, 27, 144, 183, 244, 268
tab-stops, 270
tab_frame
horizontal_resize_cursor., 96
split_bias., 96
tile_border_root., 96
vertical_resize_cursor., 96
tab_frame *class*, 90

tab_frame *library*, 90, 248
tab_stack *class*, 27, 268
tab_frame
←split_bias, 91
table
→border, 169
→cell_padding, 170
→cell_spacing, 170
→frame, 170
→next_row, 169
→rules, 170
table *class*, 38, 167–169, 196, 254, 263, 268, 269
table_cell *class*, 168, 197
table_column *class*, 168
table_row *class*, 169, 197
table_row
→end_group, 197
tabular, 248
←table, 169
→append_label_button, 169
→append, 168
→event, 169
→initialise, 168
→sort_rows, 169
→table_width, 169
tabular *class*, 168, 169
tabular *library*, 167
tbox *class*, 191, 192, 197, 251
TCP/IP, 266
template *class*, 81, 231
term_expansion/2, 83
terminal_block *class*, 269, 270
terminal_image *class*, 261, 269, 270
terminal_image
←block_at, 269
←blocks, 269
testing
input devices, 64
text
rich, 47
text *class*, 38, 39, 43, 48, 87, 152, 168, 239, 245, 253, 255
text-entry field, 27
text-entry-field, 271
text_buffer *class*, 124, 218, 246, 248, 263, 266–268, 270, 271
text_cursor *class*, 246
text_image *class*, 79, 238, 246, 270, 271
text_item *class*, 27, 28, 253, 271
text_margin *class*, 246, 271
text_buffer
←read_as_file, 218
←size_as_file, 218
→format, 218
→write_as_file, 218
text_item
→placeholder, 271
theme, 138
theme colour, 138, 139
theme editor, 138
theme_colour *class*, 139, 241, 271
theme_colours/1, 143
theme_colour
←derived_from, 140, 271
theme_editor/0, 138
theme_editor/1, 138
thread, 84
tick-box, 27
tile
gap_colour., 95
separator_colour., 95
separator_pen., 96
→hor_shrink, 120
→hor_stretch, 120
→ideal_height, 120
→ideal_width, 120
→ver_shrink, 120
→ver_stretch, 120
tile *class*, 90, 119, 120, 248
timer *class*, 56
toc_directory *class*, 165
toc_file
→initialise, 165
toc_file *class*, 164, 165
toc_filesystem *class*, 165
toc_filesystem *library*, 165
toc_folder
→initialise, 165
toc_folder *class*, 164, 165
toc_window
←node, 165
←popup, 165
←selection, 165
→collapse_node, 164

→delete, 165
→expand_node, 164
→expand_root, 165
→open_node, 164
→root, 165
→select_node, 164
→son, 165
toc_window *class*, 163, 164
toc_window
→drag_and_drop, 165
→son, 164
tokeniser *class*, 272
tool_bar
→activate, 159
→append, 159
→initialise, 159
tool_bar *class*, 159, 160
tool_button
→activate, 160
→active, 160
→initialise, 159
tool_button *class*, 159, 160
tool_dialog *class*, 160
tool_pane *class*, 90
tool_status_button *class*, 159, 160
tool_bar
←client, 159
→append, 159
tool_dialog
→append, 160
toolbar *library*, 159
Tools *menu*, 201, 202
touch screen, 55
touch_point *class*, 54, 55
trace/0, 201
tracepce/1, 21, 201, 221, 222, 262
trackpad, 54
transform, 48
transform *class*, 59, 228, 247, 264, 272
Transform *demo*, 247, 272
transient
frame, 117
transparency, 45
tray, 64
tray_icon *class*, 53, 64, 272
tray_icon
→update, 64
tree *class*, 154, 163, 210, 258, 272
treemap, 179
treemap_chart *class*, 180, 181
triangle, 259
tuple *class*, 155
type, 10, 17
←convert, 17
←value_set, 271
conversion, 10, 21
type *class*, 17, 80, 271, 273

UI
structure of, 16
undefined
class, 129
object reference, 129
undo, 246
use_class_template/1, 72
use_class_template/1, 81, 82, 231
use_module/2, 219
user-defined graphics, 243

var *class*, 66, 83, 106, 234, 240, 273
vararg, 81
variable
→clone_style, 83
variable *class*, 230, 235, 273
variable number of arguments, 81
variable/[3
4], 72
vector *class*, 47, 68, 241
vfont
←font, 194
vfont *class*, 194, 195
video, 58, 61
⇆position, 61
←image, 61
←live, 62
←status, 62
←texture, 61
→option, 62
→pause, 61
→play, 61
→stop, 61
video *class*, 61, 62, 265, 273
video_writer *class*, 61, 62, 235, 238, 274
video_writer

→append, 61
→close, 61
view
→load, 13
view *class*, 13, 27, 210, 218, 246, 274, 275
view/1, 13
view/2, 13
visual
←contained_in, 274
←contains, 274
←help_message, 157
←tooltip, 157
→destroy, 274
→help_message, 157
→report, 123
→tooltip, 157
visual *class*, 117, 123, 157, 234, 245
Visual Hierarchy *tool*, 245, 274

WAIS, 22
WAV, 57
webcam, 58
wheel, 54
wheel_data *class*, 54
when *class*, 274
while *class*, 232, 235
window
⇆smooth_scroll, 54
←height, 11
←popup, 182
→_redraw_area, 150
→above, 119
→create, 119
→flush, 37
→open, 84, 119
→pdf, 52, 153, 228, 275
→resize_message, 30
→svg, 52, 153, 275
coordinates, 36
layout in frame, 119
size specification, 120
window *class*, 11, 25, 36, 38, 119, 163, 168, 210, 243, 254, 260, 264, 265, 274, 275
window_decorator *class*, 275
Windows, 206
WinSock, 206
write
text to object, 218
WWW
addresses, 2
wysiwyg/1, 31

X11, 242
XPM
file format, 143

zoom, 56