A Tutorial to the Clean Object I/O Library - version 1.0.1

Draft version

Peter Achten

September 8, 1998

Contents

Chapter 1

Preface

The functional programming language Clean has an extensive library to build graphical user interface applications, the *object I/O library*. In this tutorial the basic concepts of the object I/O library are explained by means of examples. In this report all Clean code will be printed in `type writer` style. All examples are also available as Clean sources in the corresponding 'Tutorial Examples' folder.

This tutorial is not a technical reference manual. It is assumed that the reader is familiar with functional programming and Clean.

In Section 2 a very broad overview of the object I/O library is given, just to give the reader a taste of what the object I/O system is all about. Section 3 presents the global structure of the object I/O system.

The remaining part of this document explains the individual components of the library. But before we can explain the graphical user interface elements, we first talk about object identification (Section 4), font and text handling (Section 5), and drawing (Section 6). In Section 7 we discuss clipboard handling, a simple user driven mechanism to transfer data between interactive applications.

To users of graphical user interfaces *the* interface elements are ofcourse the *windows* and *dialogues*. These are discussed in Section 8. Windows and dialogues can contain *controls*. Because there are many aspects about control handling their treatment deserves a separate section (9). In all graphical user interface systems, the set of available commands is presented by means of *menus*, see Section 10. To support timing features, *timers* can be used, see Section 11. Flexible communication of arbitrary expressions between components can be achieved by using *receivers* and *message passing*, see Section 12.

All of the above objects are element of one interactive process. The object I/O library enables the programmer to split up a large interactive program into several interactive processes that can be created and closed dynamically. This is presented in Section 13.

Appendix A contains the definition modules of the Clean Object I/O library, version 1.0.1. in alphabetic order.

Chapter 2

Introduction

In this chapter we give a brief overview of the main features of the object I/O library. We first discuss what the basic components are and how they can be used to construct more complex components (Section 2.1). When these elements have been constructed, they must be opened to create an actual working image on the underlying platform. Elements can be opened and closed dynamically, but it is ofcourse also possible to change them dynamically (Section 2.2). Once we know how to construct graphical user interfaces we can start an interactive program. This is explained in Section 2.3. Finally, to wrap things up Section 2.4 presents the first complete interactive Clean object I/O program of this tutorial, the ubiquitous "Hello world!".

2.1 What are interactive objects

One way of looking at the object I/O library is to regard it as a collection of building blocks, the *interactive objects*, that the programmer can use to construct graphical user interfaces. For instance, Figure 2.1 summarises the standard set of *control* objects that can be placed in a *window* object.

All interactive objects are defined by means of algebraic data types. For instance, to define the button control element in the table one would write:

```
button = ButtonControl "Button" []
```

The constituents of this expressions are the data constructor `ButtonControl`, applied to the string `"Button"`, and an empty list `[]` of control attributes. This is defined more concisely by the library type definition of a button control element:

```
:: ButtonControl ls ps
=  ButtonControl String [ControlAttribute (ls,ps)]
```

Note that the names of the type constructor and the data constructor are identical (`ButtonControl`). This convention is used throughout the object I/O library.

The type definition of the button control is parameterised with two type variables: *ls* and *ps*. These correspond to another fundamental characteristic of interactive objects: an interactive object can have *local state*, and also have an effect on a *public state*. The type of the local state is identified by the *ls* type parameter, while the type of the public state is identified by the *ps* type parameter. The *effect* of an

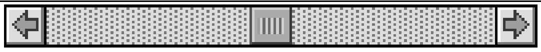
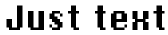
Control object:	**What does it look like:**
`RadioControl`	
`CheckControl`	
`PopUpControl`	
`SliderControl`	
`TextControl`	
`EditControl`	
`ButtonControl`	
`CustomButtonControl`	Program defined button
`CustomControl`	Program defined button
`CompoundControl`	Program defined combination of controls

Figure 2.1: The standard set of controls.

interactive object that has a local state of type ls and a public state of type ps is defined by means of a function of type $(ls, ps) \rightarrow (ls, ps)$. Such a function is called a *callback function*. For most interactive objects, the callback function is an *attribute* of the object. Attributes are also defined by means of algebraic data types. For instance, among many other control attributes, one can find the callback function attribute of controls:

```
:: ControlAttribute state
=  ... | ControlFunction (state->state) | ...
```

Note that the pair of local state and public state constitute the *state* of an element. The *meaning* of attributing a control element with a callback function f is that when that element is selected by the user, and the current state is the value (l, p), then the new state will be $(f(l, p))$. In other words, a callback function defines a *state transition*.

Besides having a bag of interactive objects the object I/O library provides programmers *glue* to construct user interfaces. This glue serves two purposes: *(a)* from primitive objects one can construct new composite objects, and *(b)* it puts restrictions on what components are 'glue compatible'.

The object I/O library has one universal glue *:+:* that can be used to connect two interactive objects that operate on the same local state of type ls and public state of type ps. Its type definition is as follows:

```
:: :+:  t1 t2 ls ps
= (:+:) infixr 9 (t1 ls ps) (t2 ls ps)
```

In order to define what components are compatible to be glued *type constructor classes* are applied. The type constructor class `Controls` contains all control elements (see table above) but also defines that only `Controls` members can be glued:

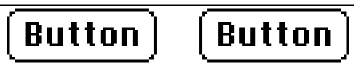
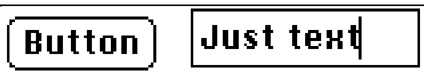
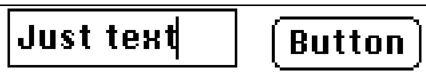
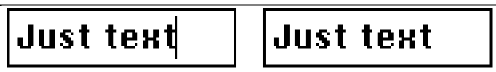
```
instance Controls RadioControl,
```

```
                 CheckControl,
                 PopUpControl,
                 SliderControl,
                 TextControl,
                 EditControl,
                 ButtonControl,
                 CustomButtonControl,
                 CustomControl,
                 :+: t1 t2 | Controls t1 & Controls t2
```

The last line of the instance declaration list states that if `e1` and `e2` are `Controls` instances, then the expression `e1:+:e2` is also a `Controls` instance. Let `button` and `edit` below define a button control and an edit control respectively:

```
button = ButtonControl "Button"    []
edit   = EditControl   "Just text" []
```

then the following expressions are all legal `Controls` instances:

Control composition:	**What does it look like:**
`button :+: button`	
`button :+: edit`	
`edit   :+: button`	
`edit   :+: edit`	

The collection of interactive objects that is supported by the object I/O library is ordered in four categories, called *abstract devices*.

Menus: Menus provide the set of commands that are available to the user of an interactive program. A program can have an arbitrary number of menus. Menus can be hierarchical, i.e. they can contain menus (*sub menus*) which can contain submenus as well. Menu items correspond with the menu commands of the program.

Windows: Windows provide the primary interface element to the user of a program. Windows can either be *dialogues* or general purpose *windows*. Windows and dialogues can contain arbitrary collections of controls. Analogous to menus, these control collections can be hierarchical, i.e. they can contain collections of controls (*compound controls*), and so on.

Timers: Timers are used by a program to be able to *softly* synchronise actions. A timer basically triggers a callback function every passing of a given time interval.

Receivers: Receivers are the basic components that interactive objects can use to communicate messages in a flexible way.

2.2 How to manage running interactive objects

In the previous section we have had a glimpse of how to define (compositions of) interactive objects. In the object I/O library every interactive object can be *created* and *destroyed* dynamically, but we prefer to call this *opening* and *closing* which sounds more peacefully. Once an interactive object is running the program will need to *modify* it in several ways. Examples are to enable and disable menu elements depending on the state of the program, change the content of a window to reflect the state of the program, and so on. Closing an element is the ultimate modification of a running interactive object. So interactive objects have a *life-cycle* which consists of three consecutive phases: *opening*, *modification*, and *closing*. Below we discuss these phases.

2.2.1 Opening of interactive objects

For each abstract device a function is defined that will open a definition of such an abstract device instance. Again type constructor classes are used to control what elements are proper instances of each abstract device. For instance, dialogues are defined by the following type definition:

```
:: Dialog c ls ps
=  Dialog Title (c ls ps) [WindowAttribute (ls,ps)]
```

Because callback functions are state transition functions of type $(ls,ps) \rightarrow (ls,ps)$, the attribute type constructor is parameterised with (ls,ps).

The type constructor class `Dialogs` fixes the instances of dialogues. A (`Dialog c`) is a proper instance of this class, provided that `c` is a proper instance of the `Controls` type constructor class.

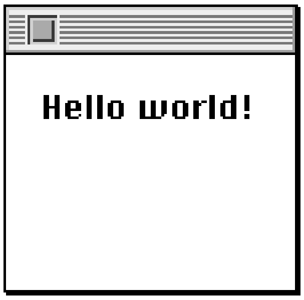
```
class Dialogs ddef where
  openDialog :: .ls (ddef .ls (PSt .l .p)) (PSt .l .p)
             -> (ErrorReport,PSt .l .p)

instance Dialogs (Dialog c) | Controls c
```

In Subsection 2.2.2 we will look more closely at the public state argument `PSt`.

Of each abstract device, the open function maps the definition of an abstract device instance (a value parameterised with callback functions) to a concrete 'physical' graphical user interface element that can be modified by the user (in case of windows, dialogues, and menus) or by the program (in case of all abstract device). As an example we can open a very small and not very useful dialogue by the following expression:

```
(error,new_public_state)
   = openDialog
        my_local_state
        (Dialog "" (TextControl "Hello world!" []) [])
        the_public_state
```

which will have the following effect (in case no error occurs):



2.2.2 Modification of interactive objects

Once an interactive object has been opened and is in its running phase, it can be modified by the user and the program. For this purpose a running interactive object must be *stored* somewhere, and it must be *identified* by the program. Every running interactive object is stored in the *I/O state* of a program. In Section 2.1 we explained that every interactive object has access to a *local* state and a *public* state. The public state, also called *process state*, is a structured value defined by means of a record:

```
:: *PSt l p             // The process state record type
=  {  ls :: l           // The local  process state
   ,  ps :: p           // The public process state
   ,  io :: *IOSt l p   // The I/O state
   }
:: *IOSt l p            // IOSt is an abstract data type
```

The `IOSt` is an abstract value of unique type created specifically for each interactive program by the object I/O system. In this special value the state of running interactive objects is stored. (The purpose of the local process state and public process state record fields ls and ps will be discussed in Chapter 13.)

The modication operations require the `IOSt` value (although some functions such as the abstract device open functions require the `PSt` record). But this is not sufficient because in general the `IOSt` will contain an arbitrary number of windows, dialogues, menus, timers, and receivers. So one has to specify which particular interactive object one intends to modify. For this purpose interactive objects (and their component structures) can be identified by means of *Ids*. An `Id` is an abstract type that can be generated by the programmer but also by the object I/O system. An interactive object is identified by means of a specific `Id` by adding this `Id` to the *attribute list* of the corresponding object definition (see Section 2.1). As an example, for controls the corresponding attribute is:

```
:: ControlAttribute state
=  ... | ControlId Id | ControlFunction (state->state) | ...
```

The major part of the object I/O library defines the modification functions that the programmer can use to modify a running interactive object.

As an example suppose we want to change the content of the text control of the "Hello world!" example to "Goodbye world!". To do this, we need to identify the text control. A control is identified uniquely by the `Id` of its window or dialogue and its personal `Id`. So the definition of both the "Hello world" dialogue as the text control need to be parameterised with an Id attribute. Let dialog_id be the Id of the dialogue, and text_id the Id of the text control. Then the adapted definition of opening the "Hello world!" dialogue is:

```
(error,new_public_state)
   = openDialog
        my_local_state
        (Dialog ""
            (TextControl "Hello world!" [ControlId text_id])
            [WindowId dialog_id]
        )
        the_public_state

changeText public_state
   = setWindow dialog_id
        [setControlTexts [(text_id,"Goodbye world!")]]
        public_state
```

2.2.3 Closing of interactive objects

As explained in the previous two subsections, once an interactive object has been opened it is in a running state and it will remain in that state until it is explicitly closed by the program. Note that although it may seem to the user that he is able to close a window, it is actually the program that responds to a *user request* to really close a window. For all interactive objects there are close operations. A close operation will remove the 'physical' graphical user interface element and free system resources that were required to operate the interactive object properly.

2.3 How to start an interactive program

The starting point of every Clean program (interactive or not) is the Start function. The essence of an interactive program is that it is a function that can change the world. So for interactive programs the Start function must have type *World → *World. The typical appearance of an interactive program looks something like this:

```
Start :: *World -> *World
Start world
   = ... world
```

In the previous sections we have seen how to define interactive objects and get them running. The abstract device open functions require a PSt value, containing an IOSt value. These are created by the object I/O system using the function startIO.

```
startIO :: !.l !.p !(ProcessInit      (PSt .l .p))
                   ![ProcessAttribute (PSt .l .p)]
                   !*World
                  -> *World
:: ProcessInit ps :== [ps->ps]
```

startIO will create an *initial* process state given the initial local and public process state components of type l and p. In particular this initial process state will contain a tailor-made `IOSt` value. In this way one switches from the world environment to the process state environment, and the interactive program can be initialised. This is done by the initialisation functions (of type `ProcessInit`).

After initialising the interactive program `startIO` will evaluate the interactive program until it becomes *closed*. The only way for an interactive program to be closed is by means of the process modification function `closeProcess`:

```
closeProcess :: (PSt .l .p) -> PSt .l .p
```

This function will close all currently running interactive objects of the interactive program and return a process state that contains an *empty* `IOSt` value. All modification operations have no effect when applied to an empty `IOSt` value. Note that if an interactive program does not close itself it will run on forever.

2.4 My first Clean object I/O program

To complete the introduction we present the Clean object I/O version of the well known “Hello world!” program. Here it is:

```
module hello

//  ****************************************************************************
//  Clean tutorial example program.
//
//  This program creates a dialog that displays "Hello world!" text.
//  ****************************************************************************

import StdEnv, StdIO

::  NoState
    =   NoState

Start :: *World -> *World
Start world
    =   startIO NoState NoState [initialise] [] world
where
    initialise process
        #   (error,process) = openDialog NoState hello process
        |   error<>NoError
            =   closeProcess process
        |   otherwise
            =   process

    hello   = Dialog ""
                (   TextControl "Hello world!" []
                )
                [   WindowClose (noLS closeProcess)
                ]
```

This program will create an interactive program which opens the same dialogue as shown earlier in Section 2.2.1. The singleton type `NoState` is applied to state that this program has no interesting local and public process state components (the first two arguments of `startIO`), and also no interesting local dialogue state (the first argument of `openDialog`). The dialogue has one attribute, the callback function that should be applied in case the user wants to close the dialogue. In this case closing the dialogue will close the “Hello world!” program. So the callback function can simply be `closeProcess`. However, the type of a callback function of an interactive object also operates on a local state (which is `NoState` in this case). To conveniently transform a function of type $(a \to b)$ into a function of type $(c, a) \to (c, b)$, the library function `noLS :: ` $(a \to b)\ (c, a) \to (c, b)$ is used.

Chapter 3

Global structure of the object I/O library

The Clean object I/O library currently consists of *thirty two* modules. All corresponding definition modules can be found in Appendix A. These modules contain everything you need to create interactive Clean programs with. *No other modules and no other symbols should be imported from the object I/O library.* Violation of this rule can result in error-prone applications at worst and non portability at least.

In this chapter the module structure of the object I/O library is discussed. This will help you to find your way quickly in this application programmer's interface. As a global naming convention, all definition modules start with `Std`. The module `StdIO` is a convenience module that collects all other modules of the object I/O library. The thirty two modules can be divided roughly into four major categories: *the abstract devices*, *interactive processes*, *drawing*, and *general*. Below they will be handled in the same order.

3.1 Abstract devices

Abstract devices have been introduced in Section 2.1 (page 11). These are the *menu*, *window*, *timer*, and *receiver* device. These devices occupy most of the modules and type definitions of the object I/O library. The following naming conventions have been employed for these modules:

- The names of the modules that contain type definitions to define abstract device instances, end with `Def`. So *menu* definitions can be found in the module `Std`*Menu*`Def`, *receiver* definitions can be found in the module `Std`*Receiver*`Def`, and so on. Although *controls* are not an abstract device, a definition module also exists for *controls*, namely `Std`*Control*`Def`.

- Abstract device instances that consist of elements use type constructor classes to enumerate their elements. The corresponding type constructor classes and standard instances are defined in the modules which names end with `Class`. So *menu elements* can be found in the module `Std`*MenuElement*`Class`. The *controls* can be found in `Std`*Control*`Class`.

- Receivers are non standard elements of some abstract device instances. Given the name *Object* of the parent device instance, you can find the type constructor class instance declarations in the modules which names are formed like `Std`*Object*`Receiver`. So, receiver instances of timers can be found in `Std`*Timer*`Receiver`.

- The operations on an object *Object* can be found in the module named `Std`-*Object*. So *window* operations can be found in the module `Std`*Window*, *menu element* operations can be found in the module `Std`*MenuElement*, and so on.

Below we discuss briefly the module structure of each of the abstract devices.

3.1.1 Menus

The API for menus and menu elements consists of six modules that are related as schematised in Figure 3.1.

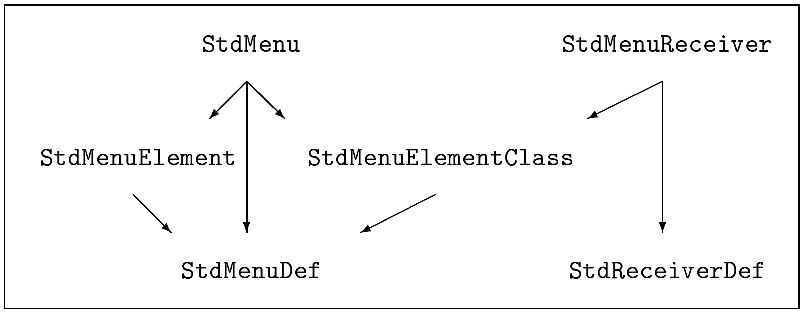


Figure 3.1: The module structure of menus.

3.1.2 Windows

The API for windows, dialogues, and controls consists of eight modules that are related as schematised in Figure 3.2.

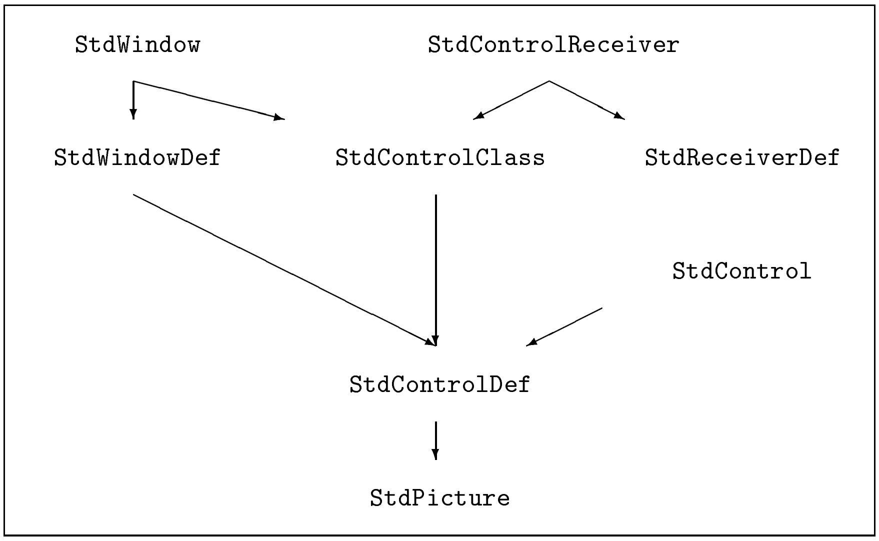


Figure 3.2: The module structure of windows.

3.1.3 Timers

The API for timers consists of six modules that are related as schematised in Figure 3.3.

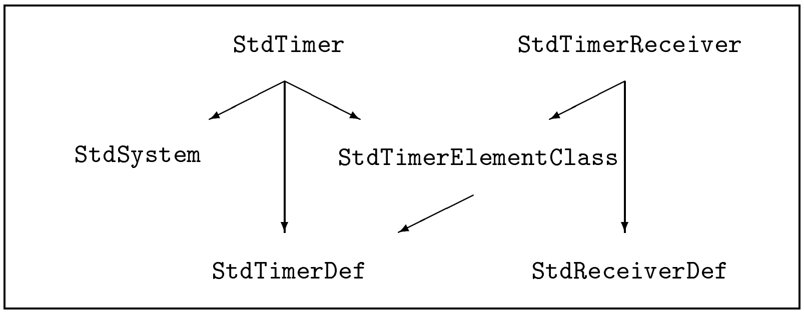


Figure 3.3: The module structure of timers.

3.1.4 Receivers

The API for receivers consists of two modules that are related as schematised in Figure 3.4.

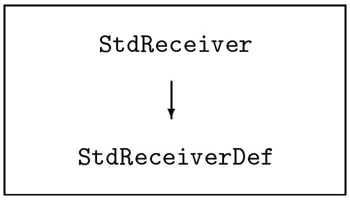


Figure 3.4: The module structure of receivers.

3.2 Interactive processes

As stated in Section 2.3, every interactive program has to be opened as an interactive process. Interactive processes are handled in more detail in Chapter 13. The API for interactive processes consists of three modules that are related as schematised in Figure 3.5.

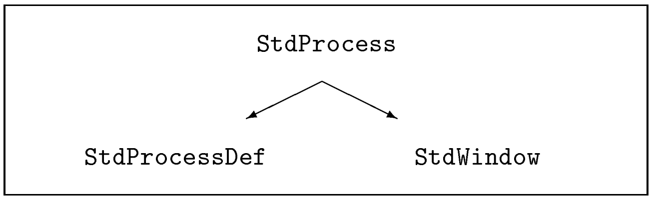


Figure 3.5: The module structure of interactive processes.

3.3 Drawing

In graphical user interface applications graphics play an important role. Virtually every interface object has a visual representation that is drawn by the underlying platform. Drawing operations will be required by most applications to give the user visual feedback on the current documents that are being manipulated or the status of controls. The manipulation of text is also an issue in drawing information. Text can be presented in very different fonts, sizes, and variations (see Chapter 5). Drawing is discussed in detail in Chapter 6.

The API for drawing consists of five modules that are related as schematised in Figure 3.6.

```
 StdBitmap
     ↓
 StdPicture
     ↓
StdPictureDef
     ↓
  StdFont
     ↓
 StdFontDef
```

Figure 3.6: The module structure of drawing operations.

3.4 General

Finally, there are a number of modules that are less easily categorised. These are the following seven modules:

`StdClipboard`: In this module *clipboard* operations are defined. Clipboard operations are defined in more detail in Chapter 7.

`StdFileSelect`: In this module two functions are defined by which a user can open platform dependent directory browsing dialogues.

`StdId`: In this module the *identification* value generating functions are defined. This is discussed in more detail in Chapter 4.

`StdIOCommon`: In this module a lot of type definitions and functions are provided that are needed by many of the abstract device modules introduced in Section 3.1.

`StdMaybe`: In this module a type is introduced that is very useful for providing optional results and optional arguments. It is used by many operations in the abstract device modules.

`StdPSt`: In this module operations are collected on the process state that are not related to any abstract device. It contains several type constructor class instance declarations for file, font, and time access. Other frequently used functions are the 'lifting' functions defined on the process state. With these lifting functions one can easily transform for instance an `IOSt` transition function to a `PSt` transition function.

`StdTime`: In this module some time access operations can be found that can be used independently of the timer device.

Chapter 4

Object identification

Before we can really delve into the details of the object I/O library we first need to learn how to identify running interactive objects. As we have briefly discussed in Section 2.2.2, all objects can have an *identification attribute*. An identification attribute is a value of type `Id`. This value is an attribute, so the programmer is not forced to provide a value for this attribute. Objects without identification attribute can not be modified at run-time. If the program needs to modify an interactive object, it must have been provided with an identification attribute. In the remainder of this chapter we will explain about the identification of the other devices.

The type `Id` is an abstract data type, and you can import it via the module `StdId`. *All* `Id`s are generated by the system. The type constructor class `Ids` defines the creation functions. `Ids` can be created from the `World` environment and the `IOSt` environment. Every new `Id` taken from these environments is guaranteed to be fresh with respect to the other `Ids` generated by any of these functions.

```
::  Id

class Ids env where
    openId   ::      !*env -> (!Id,        !*env)
    openIds  :: !Int !*env -> (![Id],      !*env)

    openRId  ::      !*env -> (! RId m,    !*env)
    openRIds :: !Int !*env -> (![RId m],   !*env)

    openR2Id ::      !*env -> (! R2Id m r, !*env)
    openR2Ids:: !Int !*env -> (![R2Id m r],!*env)

instance Ids World
instance Ids (IOSt .l .p)
```

As the `Ids` class definition shows, `Id` values are not the only identification values that are used in the object I/O system. Two special kinds of identification values of type `RId m` and `R2Id m r` are used to identify receivers that respond to messages of type `m` in the first case, and in addition respond with a message of type `r` in the second case. Receivers are discussed in Chapter 12.

The purpose of having `Ids` is to unambiguously identify running interactive objects. When assigning `Ids` to interactive objects, a program must comply to the following

`Id` *assignment rules*:

- Within one abstract device element instance, the `Ids` assigned to its elements must be unique.
- Within one abstract device, the `Ids` assigned to the abstract device element instances must be unique.

For instance, the `Ids` assigned to windows must all be unique, and inside a window the `Ids` assigned to its controls must be unique. But for two different windows, the `Ids` assigned to the controls may overlap. Also the `Ids` assigned to other abstract device element instances such as menus and timers may overlap.

For `RIds` and `R2Ids` the assignment rules are even more strict: at all times when a program is running, the `R(2)Ids` assigned to open receivers must be unique. The reason for this is explained in Chapter 12.

The abstract device open functions check whether the interactive object definition argument is valid with respect to the `Id` assignment rules. If this is not the case, the `ErrorReport` alternative `ErrorIdsInUse` is returned, and the interactive object will not be opened. If the `Id` assignment rules were not violated, and no other error occured, then the alternative `NoError` is returned.

Chapter 5

Font and text handling

When working with text you frequently will want to know the dimensions of the text for layout purposes or simply to calculate the size of an element containing that text. The dimensions of a piece of text depend on two parameters:

- The *font* is an abstract value that describes the shape of a text. The usual way to identify a font is by its name, point size, and style variations.
- The *drawing environment* determines the actual size in terms of a resolution dependent unit. The resolution of the screen is usually a lot smaller than the resolution of a laser writer.

Because there is a great variance of available fonts and drawing environments per machine writing a program that handles fonts properly requires some experience.

The font data type definitions and operations can be found in the definition modules `StdFontDef` and `StdFont`. Except for one operation, all font operations are overloaded in the drawing environment argument and are part of the `FontEnv` type constructor class (see `StdFont`). The operations are ordered in three groups.

The first group of font operations return information about the currently installed fonts. The function `getFontNames` returns a list of the names of all available fonts. Given an element of this list, the functions `getFontStyles` and `getFontSizes` return for that particular font the available style variations and sizes. Because in modern font management systems there is no restriction anymore on the size of a font, the function `getFontSizes` is also parameterised with two bounding size arguments. Their type definitions are:

```
getFontNames ::                     !*env->(![FontName], !*env)
getFontStyles::           !FontName !*env->(![FontStyle],!*env)
getFontSizes ::!Int !Int !FontName !*env->(![FontSize], !*env)
```

The second group of font operations opens fonts. The function `openFont` creates a value of type `Font` given a font definition. A font definition is a record of type `FontDef` (see `StdFontDef`) and is defined as follows:

```
:: FontDef
   =   { fName   :: !FontName
       , fStyles :: ![FontStyle]
       , fSize   :: !FontSize
       }
```

Because there are so many different font systems and style variations both font names and style variations are of type `String`. If you want to be sure that you are selecting an existing font use the functions of the first group. In any case, if the font definition argument of `openFont` does not correspond with an installed font, then it returns a `False` Boolean and a dummy font value. If it succeeds to find a matching font it will return a `True` Boolean and that font. The type of `openFont` is:

```
openFont :: !FontDef !*env -> (!(!Bool,!Font),!*env)
```

There are two functions that open the font that is used by default in an application window (`openDefaultFont`) and the font that is used by the system for controls and window titles and so on (`openDialogFont`). These fonts are of course always available. Their types are:

```
openDefaultFont :: !*env -> (!Font,!*env)
openDialogFont  :: !*env -> (!Font,!*env)
```

The last group of font operations determine the metrics of fonts and text. The metrics of a font consists of three values related to the height of characters of that font (*leading*, *ascent*, and *descent*) and one value related to the width of characters of that font (*max. width*). Figure 5.1 below explains the height characteristics.

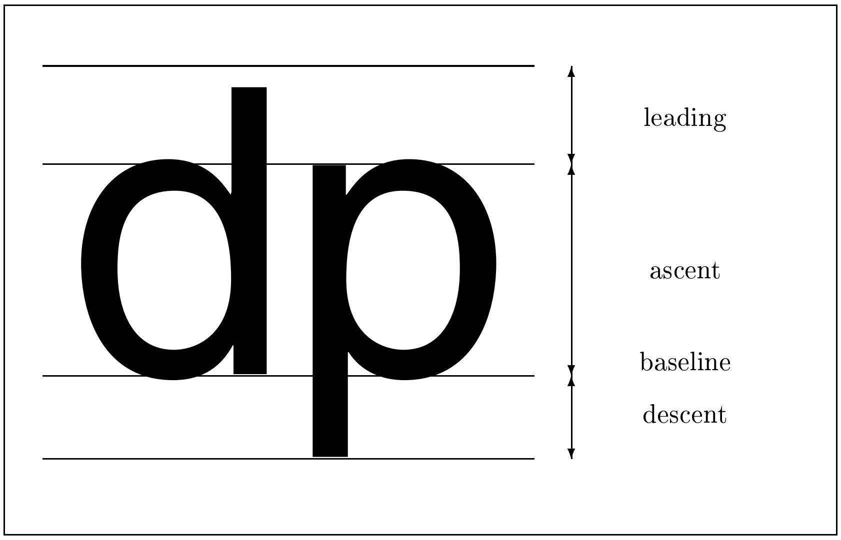


Figure 5.1: Font metrics.

The max. width characteristic is the maximum width of all characters in that particular font. For *non-proportional* fonts such as `Courier` this implies that the width of all characters is identical, so "`iii`" is just as wide as "`mmm`". For *proportional* fonts such as this text the width of characters can vary a lot. Compare for instance the width of the text "iii" with "mmm". The metrics of a font are collected in a record of type `FontMetrics`:

```
:: FontMetrics
   =   { fAscent  :: !Int  // The ascent  of the font
       , fDescent :: !Int  // The descent of the font
```

```
, fLeading :: !Int  // The leading of the font
, fMaxWidth:: !Int  // The max. width of the font
}
```

There are four other functions in the last group of font operations. With these functions one can calculate the width of a (list of) character(s), and a (list of) string(s). Actually the list versions (singleton versions) are redundant and can be expressed in terms of the singleton version (list version) of the functions. They have been added for reasons of efficiency because on some systems calculating the size of a piece of text can be expensive.

There is a subtle difference in calculating the width of *one character* versus the width of *one string*. The width of a character is determined by the character only. The width of a string can depend on the order of the characters it contains. A font system can take advantage of the fact that some adjacent characters can be placed more closely together to obtain a better looking result when drawing the string. This is called *kerning*. In the object I/O system, the programmer can rely on the fact that if a piece of text is drawn character by character then the character width function returns the correct width of the drawn character. If a piece of text is drawn by using a string, then the string width function returns the correct width of the drawn string.

In the `StdFont` module `World` is declared to be an instance. `World` itself is not a drawing environment. When applied to a `World` value, the metrics functions will be defined in terms of pixels. One final function in the `StdFont` module, `getFontDef`, returns the `FontDef` of a given `Font` value.

Because there is wide variety of fonts available the `StdFontDef` module provides a number of macros that help you make a program less dependent on the set of available fonts. The following macros provide a number of font definitions that are guaranteed to be available on the platform:

Font macros:	**Example:**
`SerifFontDef`	Garamond, Times
`SansSerifFontDef`	Helvetica
`SmallFontDef`	"This is a small text"
`NonProportionalFontDef`	`Courier`
`SymbolFontDef`	$\forall\ \exists\ \alpha\ \beta \Leftarrow \Rightarrow$

The following macros provide a number of standard font variations that are guaranteed to be available on the platform:

Style macros:	**Example:**
`ItalicsStyle`	*Madam, I'm Adam*
`BoldStyle`	**Madam, I'm Adam**
`UnderlinedStyle`	Madam, I'm Adam

Chapter 6

Drawing

All drawing functions require an environment of type `*Picture`. A `Picture` is created for each interactive object that can be drawn in. The life-time of a `Picture` environment is equal to the life-time of its parent object.

A `Picture` defines a coordinate system for drawing operations. Increasing x-axis coordinates run from left to right. Increasing y-axis coordinates run from top to bottom. The range for both axes is the full Clean `Integer` range.

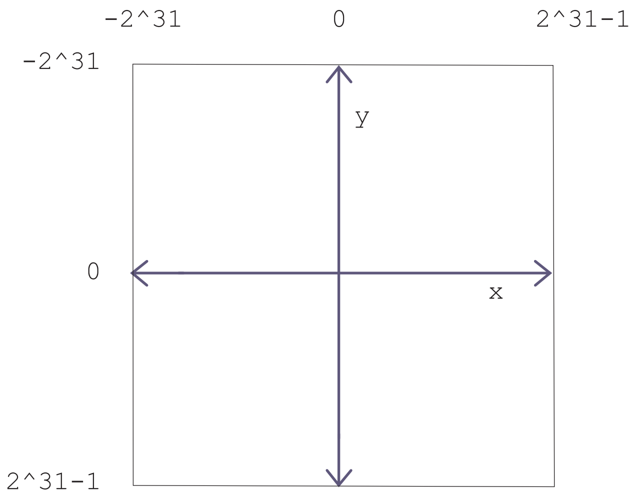


Figure 6.1: Picture coordinates.

Drawing operations on a `Picture` use the coordinate system to define where objects should be drawn. The objects themselves are made up of the pixels, lying between the coordinates. Figure 6.2 zooms in on the coordinate system from zero and increasing. The pixels on x-coordinates 0, 5, 10,... and y-coordinates 0, 5, 10,... are displayed.

Like most other interactive objects in the object I/O library, pictures have attributes. These are the following (they can be found in the module `StdPictureDef`):

```
:: PictureAttribute
   =   PicturePenSize   Int
   |   PicturePenPos    Point
```

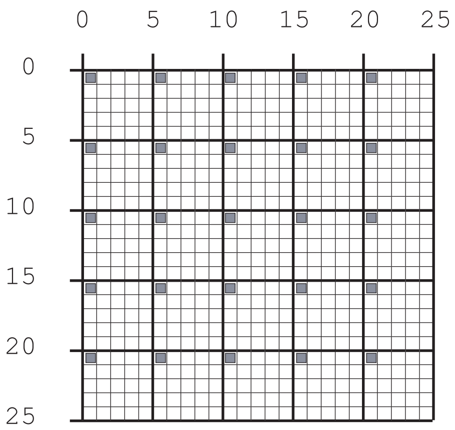


Figure 6.2: Picture coordinates and pixels

```
|   PicturePenFont   Font
|   PicturePenColour Colour
```

`PicturePenSize`: This attribute defines the width and height of the drawing pen. The default value is 1, which means that drawing a point will fill an area of 1 pixel wide and 1 pixel high. Negative or zero values are always set to 1.

`PicturePenPos`: This attribute determines the current position of the drawing pen. Its default value is zero. The definition of `Point` is:

```
:: Point = {x::!Int,y::!Int}
```

`PicturePenFont`: This attribute sets the current font that will be used when drawing text (see also Chapter 5). Drawing text is not affected by the current width and height of the pen. Text is always drawn at the baseline of the font (see Figure 5.1).

`PicturePenColour`: This attribute determines the colour that is used when drawing any element. The `Colour` data type is also defined in the module `StdPictureDef`:

```
:: Colour
   =   Black | DarkGrey | Grey | LightGrey | White
   |   Red   | Green    | Blue
   |   Cyan  | Magenta  | Yellow
   |   RGB RGBColour
:: RGBColour
   =   {r::!Int, g::!Int, b::!Int}
```

A colour can range between black and white (first five alternatives defining 100%, 75%, 50%, 25%, and 0% blackness), be one of red, green, blue, be one of cyan, magenta, yellow, or some custom defined combination of red, green, blue components. Currently the library does not support colour tables or palette management operations, so the use of RGB colours tends to be speculative.

Given a `*Picture` environment, the function `getPicture` returns a 'read-only' picture environment (of type `Picture`). From this value the current attribute values can be obtained by the function `getPictureAttributes`. With `setPictureAttributes` one can change a number of picture attributes using a list of them. For all picture attributes there are also functions to get and set them individually, using an updateable picture (of type `*Picture`).

The drawing operations are divided into three groups, ordered by means of type constructor classes:

Drawables draws the 'outline' of its elements. Its instances are characters, strings, vectors, ovals, curves, boxes, rectangles, polygons, and bitmaps.

Fillables fills the interior of its elements. Its instances are ovals, curves, boxes, rectangles, and polygons.

Hilites fills the interior of its elements in such a way that the current content remains visible. Its instances are boxes and rectangles.

Each of these type constructor classes allows its elements to be drawn at the *current* pen position or at an *absolute* pen position. Because of this reason the data type definition of most of these elements do not specify their location. Exceptions are rectangles, lines, and points.

For making more complex drawings, *clipping* is also supported. Drawing within a clipping area forces all drawing to occur inside that area. Again, a type constructor class, `Clips` is used for this purpose. One can clip within a box, a rectangle, a polygon, or a list of clipping elements.

6.1 Examples

In this section we give small examples of all of the drawable elements.

6.1.1 Drawing text

Given a pen position `{x,y}`, drawing a piece of text (`Char` or `String`) will always draw the text using the current `PicturePenFont`. The shape of the drawn characters relies only the font information, not on the current `PicturePenSize`. Text can be drawn in any of the available colours. The baseline of the particular font determines the position of the first character, which is drawn with its left baseline starting at `{x,y}`. Figure 6.3 shows the result of applying `(drawAt zero "Pop" picture)` applied to an empty `picture`.

The new pen position is, in this case, `{x=24,y=0}`. One might expect the new pen position to be `{x=22,y=0}`, but usually the horizontal character space is also included. This facilitates drawing text character by character. But some caution should be taken. One might expect that the function `(draw "p" (draw "o" (draw "P" picture)))` produces the same result, but this depends on the font (as explained in Chapter 5), so in general one should not assume that this is the case. The only certain way to know how much the pen position will change in case of text is by calculating the width of the *same* text, or by comparing the pen positions before and after drawing.

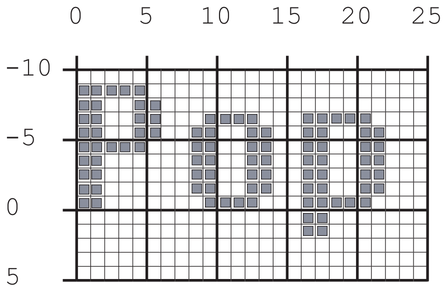


Figure 6.3: Drawing the text "Pop" at zero.

6.1.2 Pen size and position

Given a pen position `{x,y}`, drawing a point, a line, text, always occurs to the right and below the pen position. Figure 6.4 illustrates these cases: from left to right the function `(drawPointAt zero (setPicturePenSize` n `picture))` is applied to an empty `picture`, and `PicturePenSize` attribute values n 1, 2, and 3 respectively.

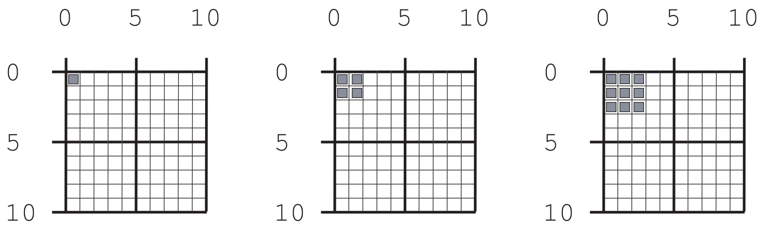


Figure 6.4: Drawing a point at zero with different pen sizes.

6.1.3 Drawing lines

The shape of a line is influenced by the `PicturePenSize` attribute in the same way as the shape of points are changed (see Figure 6.5).

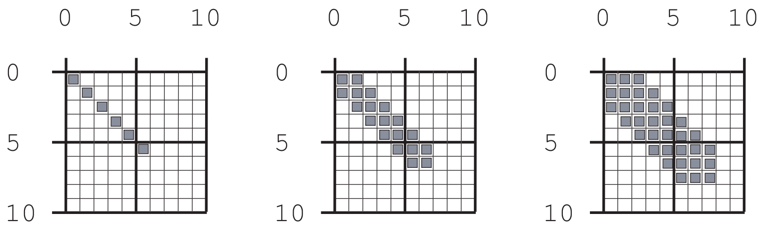


Figure 6.5: Drawing a line from zero to `{x=5,y=5}` with different pen sizes.

There are several ways to draw lines. The function `drawLineTo` draws a line from the current pen position to the argument point. If these happen to be equal, then the result is the same as `drawPointAt` with the same argument. The new pen position is the same as the target point. The function `drawLine` draws a line from the first argument point to the second argument point without changing the pen position.

Lines can also be drawn using the `Vector` instance functions from the `Drawables` type constructor class. The function `draw` applied to a vector `{vx,vy}` draws a line from the current pen position `{x,y}` to the point `{x=x+vx,y=y+vy}`. The function `drawAt` applied to a point `{x,y}` and a vector `{vx,vy}` draws a line from `{x,y}` to the point `{x=x+vx,y=y+vy}`.

6.1.4 Drawing ovals

An `Oval` is a transformed unit circle defined by a horizontal radius, `oval_rx` and a vertical radius, `oval_ry`. For each point `{x,y}` on a unit circle, its corresponding point on the oval is given by `{x=x*oval_rx,y=y*oval_ry}`. The type definition of an `Oval` is:

```
:: Oval = {oval_rx::!Int, oval_ry::!Int}
```

Both radius values are always taken to be atleast zero. If any of these values is negative, then zero is used instead. Ovals are drawn using the `Oval` instance functions from the `Drawables` type constructor class. The function `draw` uses the current pen position as the center of the oval. The function `drawAt` uses the argument `Point` as the center of the oval. Drawing an oval does not change the pen position. In case one of the radius values is taken to be zero drawing the oval displays nothing. Figure 6.6 draws three ovals at `zero` defined as follows:

```
{oval_rx=5, oval_ry=3}
{oval_rx=5, oval_ry=5}
{oval_rx=3, oval_ry=5}
```

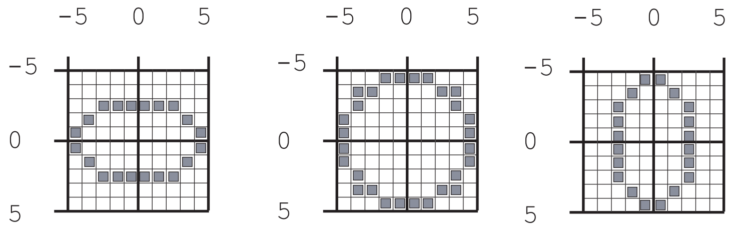


Figure 6.6: Three oval shapes drawn at zero.

The shape of an oval is also affected by the current `PicturePenSize` attribute. Increasing the pen size does not increase the outline of the oval. The only pixels that are affected are inside the oval. Figure 6.7 shows the center oval of Figure 6.6 when drawn with pen size of 1, 2, and 3.

`Ovals` are also an instance of the `Fillables` type constructor class. The function `fill` and `fillAt` *fill* rather than draw the oval. Filling an oval includes its outline and its interior. Figure 6.8 shows the same three ovals as given in Figure 6.6, but now filled.

6.1.5 Drawing curves

A `Curve` is a section of an `Oval`. A curve is defined by the source oval, `curve_oval`, a starting angle, `curve_from` and an ending angle, `curve_to`, both taken in radians,

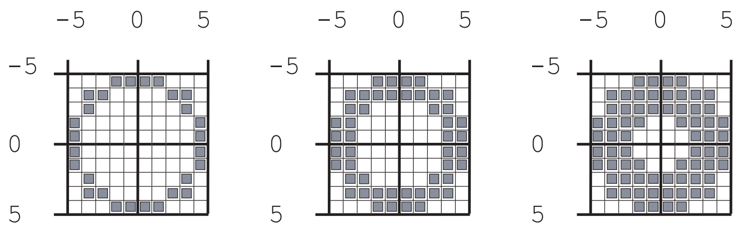


Figure 6.7: Three oval shapes drawn with increasing pen sizes.

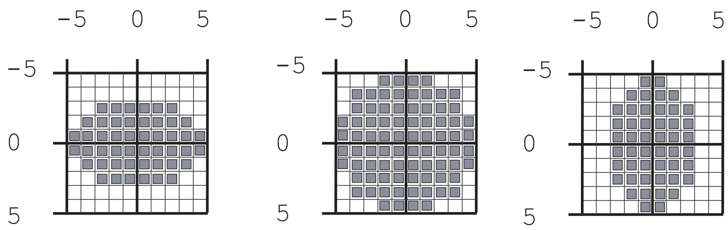


Figure 6.8: Three filled oval shapes.

and the direction in which the section should be taken, `curve_clockwise` which is a Boolean value. The type definition of a `Curve` is:

```
:: Curve
   =   {   curve_oval      :: !Oval
       ,   curve_from      :: !Real
       ,   curve_to        :: !Real
       ,   curve_clockwise :: !Bool
       }
```

The start and end point of the section are again derived from the unit circle. Given an angle `alpha`, and a source oval defined by `{oval_rx, oval_ry}`, then the point on the curve (oval) corresponding with `alpha` is `{x=oval_rx*cos alpha, y=oval_ry*sin alpha}`. If `curve_clockwise` is `True` then the section is taken clockwise from the start point to the end point, otherwise it is taken counter clockwise direction. Figure 6.9 shows two sections of an `Oval`. In both cases the `curve_from` angle is $\frac{\pi}{6}$ and the `curve_to` angle is $\frac{3\pi}{2}$. The left section is taken counter clockwise, and the right section is taken clockwise.

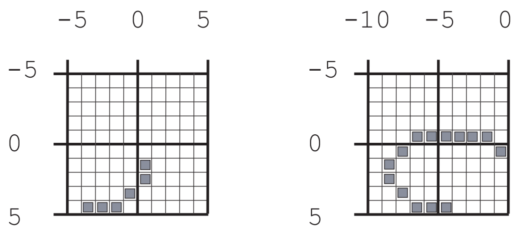


Figure 6.9: Two curves taken clockwise and counter clockwise.

Figure 6.9 not only shows the curve sections that are taken from an oval, but also what happens when these sections are drawn *at* a specific position. In both cases the curves are drawn at `zero`. The starting point, indicated by the starting angle, is determined by the current pen position in case of the `draw` function of the `Drawables` type constructor class, and is determined by the `Point` argument of the `drawAt` function of the `Drawables` type constructor class.

Drawing a `Curve` with varying `PicturePenSizes` is the same as taking the section of the corresponding `Oval` drawn with that pen size. Figure 6.10 shows three times the same curve taken but drawn with pen sizes 1,2, and 3 respectively. The source oval is the same as the one drawn in Figure 6.7. The section is taken counter clockwise from $\frac{1}{4}\pi$ to $1\frac{3}{4}\pi$.

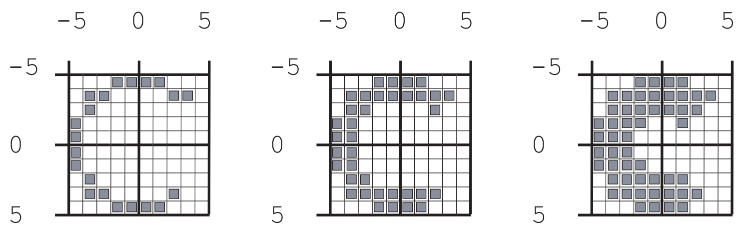


Figure 6.10: Three curves drawn with increasing pen sizes.

`Curves` are also an instance of the `Fillables` type constructor class. When filling a curve, the interior formed by the drawn curve and two lines connecting the center of the source oval and the end points of the curve is filled. Figure 6.11 shows the two curves of Figure 6.9, but now using `fill` rather than `draw`.

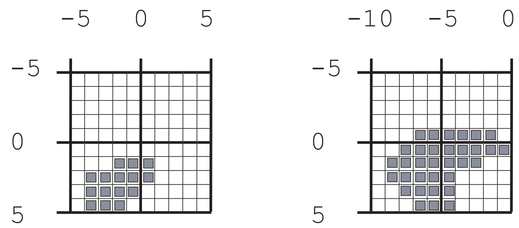


Figure 6.11: Two filled curves taken clockwise and counter clockwise.

6.1.6 Drawing rectangles

A `Rectangle` is a shape of four connected lines that is defined by two diagonally oriented corner `Points`, `corner1` and `corner2`. The type definition of a `Rectangle` is as follows:

```
:: Rectangle = {corner1::!Point, corner2::!Point}
```

`Rectangle` is an instance of the `Drawables` type constructor class. The `drawAt` function is not very useful because it ignores its `Point` argument and proceeds as `draw`. Any two `Points` are valid corner points of a `Rectangle`. In case a `Rectangle` has a zero width or zero height drawing that rectangle will show nothing. It does not matter in what order the two corner points are given. Figure 6.12 shows three `Rectangles`, defined as follows:

```
{corner1= zero,      corner2={x=10,y=6}}
{corner1= zero,      corner2={x=6, y=6}}
{corner1={x=10,y=6}, corner2= zero    }}
```

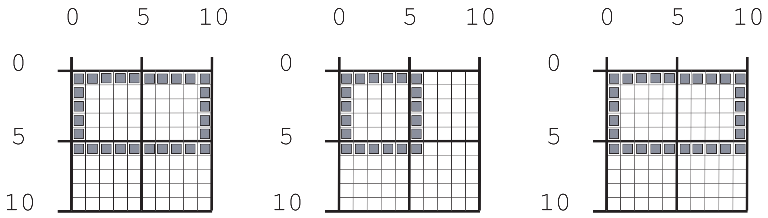


Figure 6.12: Three rectangle shapes.

The shape of a rectangle is also affected by the current `PicturePenSize` attribute. Increasing the pen size does not increase the outline of the rectangle. The only pixels that are affected are inside the rectangle. Figure 6.13 shows the `Rectangle {corner1=zero, corner2={x=10,y=10}}` when drawn with pen size 1, 2, and 3.

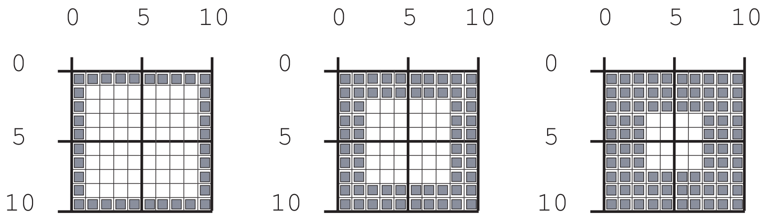


Figure 6.13: Three rectangles drawn with increasing pen sizes.

`Rectangle`s are also an instance of the `Fillables` type constructor class. The function `fill` and `fillAt` *fill* rather than draw the rectangle. Filling a rectangle includes its outline and its interior. Figure 6.14 shows the same three rectangles as given in Figure 6.12, but now filled.

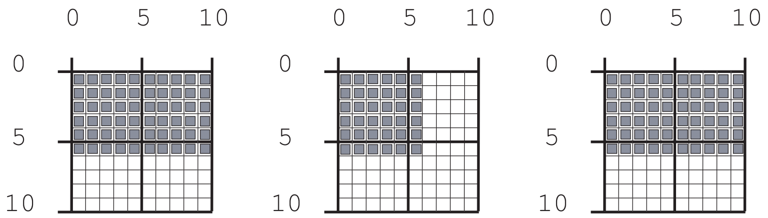


Figure 6.14: Three filled rectangles.

6.1.7 Drawing boxes

A `Box` is a `Rectangle` but without fixing a position. It is therefore only defined by a width, `box_w` and height, `box_h`. The type definition of a `Box` is:

```
:: Box = {box_w::!Int, box_h::!Int}
```

The position of a `Box` is determined by the drawing functions of the type constructor class `Drawables`. In case of `draw`, the current pen position is the base point. In case of `drawAt`, the `Point` argument is the base point. Given this base point `base={x,y}`, and a `Box {box_w,box_h}`, drawing the `Box` is the same as drawing the `Rectangle {corner1=base, corner2={x=x+box_w,y=y+box_h}}`. Any value for `box_w` or `box_h` is permitted (so also zero or negative values).

`Boxes` are drawn and filled in the same way as `Rectangles` are. So the effect of using different `PicturePenSizes` is the same as well as filling boxes.

6.1.8 Drawing polygons

A `Polygon` is an object which shape is formed by a number of `Vectors`, such as triangles, rectangles, but also more exotic shapes. The type definition of a `Polygon` is:

```
:: Polygon = {polygon_shape::![Vector]}
```

A `Polygon` is always a closed shape. A shape `polygon_shape` is closed if the following equation holds:

```
foldr (+) zero polygon_shape = zero
```

The object I/O library will always close the `polygon_shape` if this is not the case, so you don't have to worry about this. Drawing a polygon of shape `polygon_shape` is simply drawing the closed list of vectors in sequence:

```
seq (map draw polygon_shape)
```

Figure 6.15 shows three polygons defined by the following shapes (leaving out the last `Vector` gives the same `Polygon`):

```
[{vx=8,vy=0}, {vx=(-4),vy=8}, {vx=(-4),vy=(-8)}]
[{vx=8,vy=0}, {vx=0,    vy=8}, {vx=(-8),vy=0},  {vx=0,    vy=(-8)}]
[{vx=8,vy=0}, {vx=(-8),vy=8}, {vx=8,    vy=0},  {vx=(-8),vy=(-8)}]
```

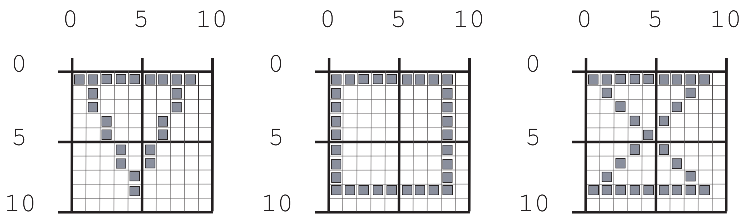


Figure 6.15: Three polygon shapes.

Similar to `Boxes`, `Polygons` do not specify their location. Again, this is determined by the drawing functions of the type constructor class `Drawables`. In case of `draw`, the base point is defined by the current pen position. In case of `drawAt`, the base point is defined by the `Point` argument. In Figure 6.15 all polygons were drawn at `zero`.

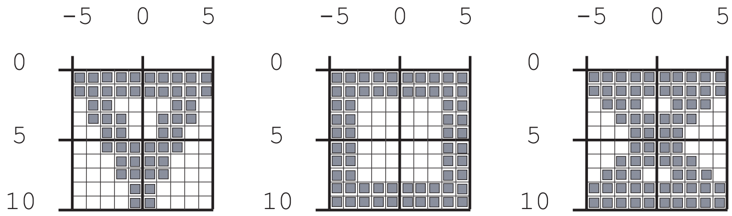


Figure 6.16: Three polygons drawn with pen size 2.

Because a polygon is a collection of vectors, its shape is affected by the current `PicturePenSize` attribute. Figure 6.16 shows the three polygons of Figure 6.15 drawn with pen size 2.

`Polygon`s are also an instance of the `Fillables` type constructor class. The function `fill` and `fillAt` *fill* rather than draw the polygon. Filling a polygon includes its outline and its interior. Figure 6.17 shows the same three polygons as given in Figure 6.15, but now filled.

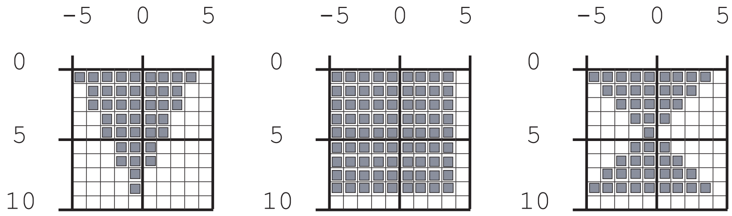


Figure 6.17: Three filled polygons.

6.1.9 Drawing bitmaps

Using the drawing operations discussed so far one can produce images that have an 'algorithmic' nature: they consist of text, lines, curves, and polygons. Not every image can be expressed (easily) in this way (consider for instance the image in Figure 6.18). To produce more complex images *bitmaps* are very useful. A bitmap is a prefabricated image of a certain size (in pixels) that is stored in the file system.

You can use your favourite drawing package and create and store images as a bitmap. The file format depends on the platform. Currently the following formats are supported:

Platform:	**Format:**
Macintosh	PICT
Windows(95/NT)	BMP

The bitmap operations can be found in module `StdBitmap` (Appendix A.1). Bitmaps can be read from file using the function `openBitmap`:

```
openBitmap :: !{#Char} !*env -> (!Maybe Bitmap,!*env)
                              |  FileSystem env
```

Given the full file name of a bitmap file, `openBitmap` reads the bitmap in memory. If this is successful, `(Just bitmap)` is returned. Reasons for failure are illegal file name

Figure 6.18: A non algorithmic image.

arguments, wrong file formats, lack of heap space (in case of Windows(95/NT)), or lack of extra memory (in case of Macintosh[1]).

A `Bitmap` is an abstract data type. The only information one can retrieve from a `Bitmap` value is its size:

```
getBitmapSize :: !Bitmap -> Size
```

`Bitmap`s are instances of the `Drawables` type constructor class. Given a current pen position `pos={x,y}` and a bitmap `bitmap` of size `{w,h}`, the functions `(draw bitmap)` and `(drawAt pos bitmap)` both place the bitmap exactly inside the rectangle `{corner1=pos, corner2={x=x+w,y=y+h}}`.

6.1.10 Drawing in XOR mode

For many programs it is sometimes useful to be able to temporarily draw figures over an existing drawing, and being able to remove it without affecting the source picture. Examples are a flashing cursor in a text editor, and anchor points in a drawing program. For this purpose drawing in XOR mode is supported. The library function `xorPicture` takes a list of drawing functions and a picture and applies the drawing functions in left-to-right order to the picture in XOR mode.

```
xorPicture :: ![DrawFunction] !*Picture -> *Picture
```

Drawing in XOR mode has the important property that drawing the same figure twice results in the same picture. Given a `picture`, and a list of drawing functions `fs`, the following equation holds:

```
xorPicture fs (xorPicture fs picture) = picture
```

Let's explain what happens in the `Picture` when one uses XOR mode. Consider a source picture, `source`, shown left in Figure 6.19, which is a circle. Next to the source picture is the picture to be drawn in XOR mode, a fat rectangle, represented by a list of drawing functions `fs`.

[1]In the current Macintosh implementation bitmaps are not garbage collected. This puts a restriction on the number of bitmaps that can be used inside one application. In a future version bitmaps will become garbage collected.

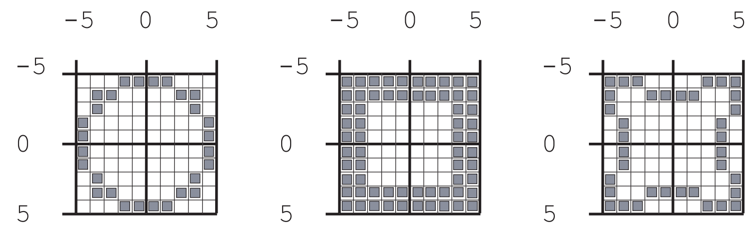


Figure 6.19: A source picture and the figure to be drawn in XOR mode.

If one interprets the white pixels of both pictures as `False`, and the black pixels of both pictures as `True` then the result of drawing `fs` in XOR mode in `source` is the same as taking the Boolean exclusive or on all such interpreted pixels on the same coordinates. So all pixels that have the same colour become black, while all pixels of different colour become white. The result of this is shown in the right picture of Figure 6.19. Applying `fs` once more in `xor` mode to this new picture yields a picture equal to `source`.

What happens when using more interesting colours than black and white is basically the same thing. In one way or another, the exclusive or is taken from the source picture and the drawing operations in such a way that repeating it gives the source picture again. What the colours of the 'xor-ed' picture are depends on the platform, and is not specified by the object I/O library.

6.1.11 Drawing in Hilite mode

Programs that want to indicate selections (for instance text segments in a word processor, or image components in a drawing program) can do this by drawing the selected area in *hilite* mode. For this purpose the type constructor class `Hilites` is used. Its type definition is:

```
class Hilites figure where
   hilite   ::        !figure !*Picture -> *Picture
   hiliteAt :: !Point !figure !*Picture -> *Picture
```

The instances of `Hilites` are `Box` and `Rectangle`. The pixels that are affected by `hilite` and `hiliteAt` are the same as for `fill` and `fillAt`. Drawing in hilite mode has the same property as drawing in XOR mode that drawing the same figure twice on a source picture leaves the source picture unchanged. Given a `picture`, and a figure `figure`, the following equation holds:

```
hilite   figure (hilite   figure picture) = picture
hiliteAt figure (hiliteAt figure picture) = picture
```

The visual effect of hiliting these areas depends on the platform. On some platforms hiliting an area will change the colour of all pixels that have the background colour to a special hilite colour, ignoring all other pixels. Hiliting the area once more will revert the hilite colours back to the background colour. If a platform does not support hilite mode, the area will be drawn in XOR mode. Figure 6.20 shows the two techniques. The source picture is the text "Pop" of Figure 6.3. In this picture the function `hilite {corner1={x=0,y=2}, corner2={x=24,y=(-10)}}` is applied. The left picture shows the result of changing background pixels into the hilite colour, the second picture shows the result of using XOR mode.

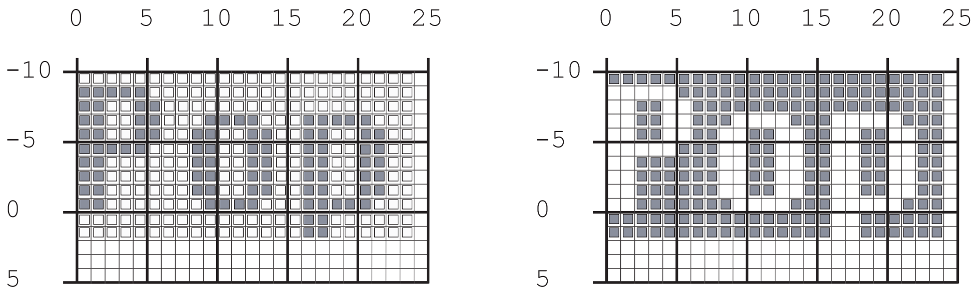


Figure 6.20: Two ways to hilite a rectangular area.

6.1.12 Drawing in Clipping mode

Drawing in *clipping* mode is a powerful technique to create graphics that can not be drawn (or using much more complicated expressions) using only the drawing primitives discussed before. In clipping mode, the programmer specifies an area that works like a mask: drawing proceeds as always, but only those pixels that are inside the clipping area are actually drawn. Clipping is done using the functions of the `Clips` type constructor class:

```
class Clips area where
   clip   ::        !area [DrawFunction] !*Picture -> *Picture
   clipAt :: !Point !area [DrawFunction] !*Picture -> *Picture
```

The clipping area can be a `Box`, `Rectangle`, `Polygon`, or a list of these. In the latter case the union area of the list elements is taken as the clipping area. The location of the clipping area is defined by the current pen position in case of `clip`, and the `Point` argument in case of `clipAt`. In case the clipping area happens to be empty no drawing is done. The list of drawing functions is evaluated as usual in left-to-right order.

Suppose we have the following list of drawing functions, `fs` which draws a number of horizontal lines with a result as shown in Figure 6.21:

```
fs = [drawLine {x=0,y=y} {x=9,y=y} \\ y<-[0,2..8]]
```

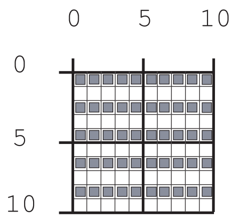


Figure 6.21: The source picture.

.

As clipping regions the polygons shown in Figure 6.15 are used. Figure 6.22 shows the result of the following clipping functions to an empty picture:

```
clipAt zero [{vx=8,vy=0}, {vx=(-4),vy=8}]                   fs
clipAt zero [{vx=8,vy=0}, {vx=0,   vy=8}, {vx=(-8),vy=0}] fs
clipAt zero [{vx=8,vy=0}, {vx=(-8),vy=8}, {vx=8,   vy=0}] fs
```

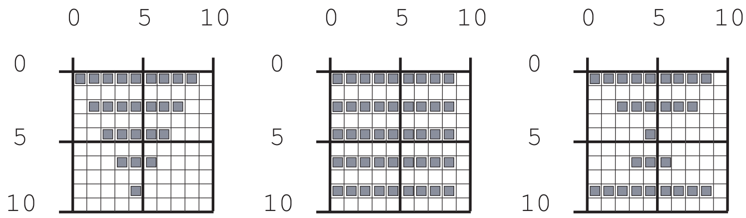


Figure 6.22: The clipped source picture.

Chapter 7

Clipboard handling

The *clipboard* is a universal simple communication metaphor between applications and within applications. One application can write some data to the clipboard (typically text or pictures) which can be read at a later point of time by the same or another application. This mechanism is supported by the functions in the definition module `StdClipboard`. At the moment only text can be handled. Because we intend to incorporate pictures as well in the near future the current version is set up in such a way that it can be extended upward compatibly. For this purpose an abstract data type, `ClipboardItem`, is defined. Two overloaded functions from the type constructor class `Clipboard` take care that data types can be converted to and from `ClipboardItem`s:

```
:: ClipboardItem

class Clipboard item where
   toClipboard   :: !item          -> ClipboardItem
   fromClipboard :: !ClipboardItem -> Maybe item

instance Clipboard {#Char}
```

A further convention of using the clipboard is that applications should provide several 'popular' data formats for the same content in descending order of accuracy. For instance, a text processor can first store its private format for the layn out text including font and style information, followed by an ASCII version of the same text, followed by a picture of the layn out text. For this reason the function `setClipboard` that writes the clipboard is not applied to one single clipboard data item but a list of them. The previous content will be destroyed completely. Because programs are supposed to provide only one data item of each format from this list duplicate types of clipboard items are removed. Note that providing `setClipboard` with an empty list will clear the clipboard.

```
setClipboard :: ![ClipboardItem] !(PSt .l .p) -> PSt .l .p
```

The function that reads the clipboard, `getClipboard`, simply gets a list of the current content of the clipboard in descending order of accuracy. With the conversion function `fromClipboard` an application can determine easily if some data item is present that it can handle.

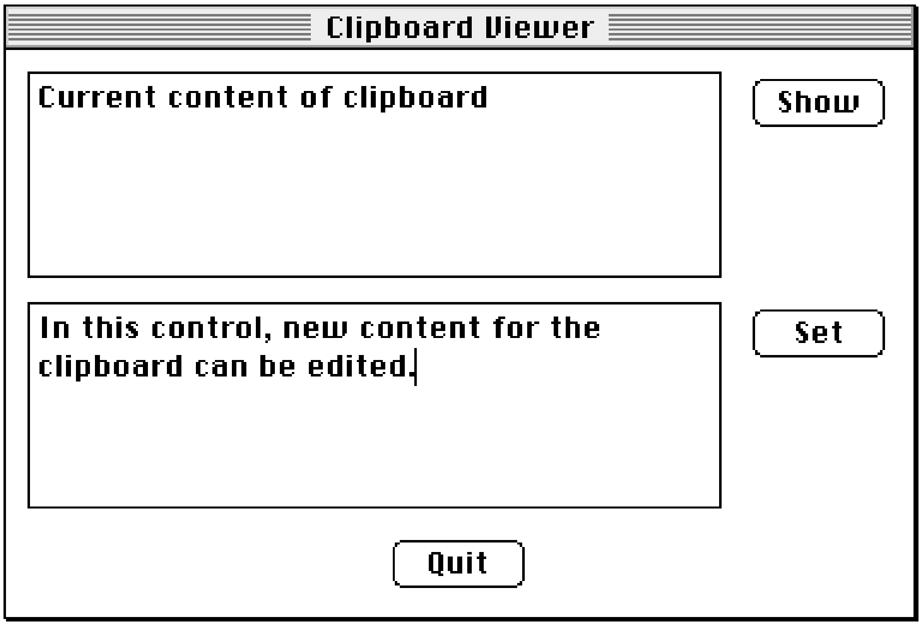
```
getClipboard :: !(PSt .l .p) -> (![ClipboardItem],!PSt .l .p)
```

Finally, because reading in a complete clipboard can be time-consuming or space-consuming a function is provided that checks whether the clipboard has been updated since the last time the program checked it.

```
clipboardHasChanged :: !(PSt .l .p) -> (!Bool,!PSt .l .p)
```

7.1 Example: a clipboard editor

To illustrate the use of the clipboard, we construct a small program that shows the current content of the clipboard and that can write some text to the clipboard (see the picture below). It will create only one dialogue with two text fields. In the first text field, the *show* field, the content of the clipboard can be *loaded* by pressing a button. In the second text field, the *set* field, the content of the clipboard can be *stored* by pressing a button.



The show text field and its activating button can be defined as follows:

```
showclip
  = EditControl   "" width nrlines [ControlSelectState Unable
                                   ,ControlId  showid
                                   ,ControlPos (Center,zero)
                                   ]
    :+:
    ButtonControl "Show" [ControlFunction (noLS show)]
```

The show text field is an edit text control that will not respond to keyboard input (because its `SelectState` attribute is `Unable`). It is identified by some `Id` of value `showid`. It will have some `width` and a height defined by the number of lines `nrlines`. The "Show" button, when selected, must read the content of the clipboard and figure out if there was a text clipboard item. It then will set the text of the show text field to the loaded clipboard content (an empty string if nothing was found). This action can be defined as follows:

```
show process
  # (content,process) = getClipboard process
    text              = getString content
  = appPIO (setWindow viewid [setControlTexts [(showid,text)]])
           process

getString [clip:clips]
  | isNothing item
    = getString clips
  | otherwise
    = fromJust item
where
  item = fromClipboard clip
getString []
  = ""
```

The set text field and its activating button are defined as follows:

```
setclip
  = EditControl   "" width nrlines [ControlId  setid
                                   ,ControlPos (Center,zero)
                                   ]
    :+:
    ButtonControl "Set"  [ControlFunction (noLS set)]
```

The set text field is an edit text control which accepts keyboard input. It is identified by some `Id` of value `setid`. It has the same dimensions as the show text control. The "Set" button, when selected, must get the content of the set text control and write this to the clipboard. This action is defined as follows:

```
set process
  # (dialog,process)  = accPIO (getWindow viewid) process
    text= fromJust (snd (hd (getControlTexts
                   [setid] (fromJust dialog))))
  = setClipboard [toClipboard text] process
```

The definition of the clipboard viewing dialogue simply summaries these elements and adds a "Quit" button to terminate the program:

```
clipview = Dialog "Clipboard Viewer"
                  (showclip :+: setclip :+: quit)
                  [WindowId viewid]
quit     = ButtonControl "Quit"
                  [ControlFunction (noLS closeProcess)
                  ,ControlPos      (Center,zero)
                  ]
```

The last details that remain to be defined are the opening of the interactive program which is very analogous to the "Hello world!" of Section 2.4. For completeness we show the complete program code.

```
module clipboardview
```

```
// ****************************************************************************
// Clean tutorial example program.
//
// This program creates a dialog to display and change the current content of the
// clipboard.
// ****************************************************************************

import StdEnv, StdIO

:: NoState
    =   NoState

Start :: *World -> *World
Start world
    #   (ids,world) = openIds 3 world
    =   startIO NoState NoState [initialise ids] [] world

initialise ids process
    #   (error,process) = openDialog NoState clipview process
    |   error<>NoError
        =   closeProcess process
    |   otherwise
        =   process
where
    (viewid,showid,setid)   = (ids!!0,ids!!1,ids!!2)

    clipview        = Dialog "Clipboard Viewer"
                        (   showclip
                        :+: setclip
                        :+: quit
                        )
                        [   WindowId viewid
                        ]
    showclip        = EditControl "" width nrlines
                        [   ControlSelectState Unable
                        ,   ControlId          showid
                        ,   ControlPos         (Center,zero)
                        ]
                      :+:
                      ButtonControl "Show"
                        [   ControlFunction    (noLS show)
                        ]
    setclip         = EditControl "" width nrlines
                        [   ControlId          setid
                        ,   ControlPos         (Center,zero)
                        ]
                      :+:
                      ButtonControl "Set"
                        [   ControlFunction    (noLS set)
                        ]
    quit            = ButtonControl "Quit"
                        [   ControlFunction    (noLS closeProcess)
                        ,   ControlPos         (Center,zero)
                        ]
    width   = hmm 200.0
    nrlines = 10

    show process
        #   (content,process)   = getClipboard process
            text                = getString content
        =   appPIO (setWindow viewid [setControlTexts [(showid,text)]]) process

    set process
        #   (dialog,process)    = accPIO (getWindow viewid) process
            text                = fromJust (snd (hd (getControlTexts [setid]
                                      (fromJust dialog))))
        =   setClipboard [toClipboard text] process
```

```
getString [clip:clips]
    |   isNothing item
        =   getString clips
    |   otherwise
        =   fromJust item
where
    item    = fromClipboard clip
getString []
    =   ""
```

Chapter 8

Windows and dialogues

Windows and *dialogues* are the major top level interactive objects of the object I/O library. In some aspects they are very similar. For instance, they can contain the same set of *controls*, and virtually all operations on windows and dialogues are similar. Dialogues differ from windows because they usually offer a special, enhanced, user interface to users. Another difference is that dialogues can be opened *modally*. In this mode the user can be forced by the program to handle the dialogue completely before continuing with the program. To emphasize the similarities of windows and dialogues, their algebraic type definitions are almost identical (these can be found in module `StdWindowDef`, Appendix A.32):

```
:: Window c ls ps = Window Title (c ls ps) [WindowAttribute *(ls,ps)]
:: Dialog c ls ps = Dialog Title (c ls ps) [WindowAttribute *(ls,ps)]
```

Before delving into details, we first introduce some basic terminology for windows and dialogues in Section 8.1 and discuss the `WindowAttribute` alternatives in Section 8.2.

8.1 Basic terminology

The main purpose of a *window* is to present to the user a view on a document, represented as an object of type `Picture`. `Picture`s have been discussed in Chapter 6. By using the mouse and keyboard, the user can manipulate the document. Controls in a window can add further manipulation functionality.

The main purpose of a *dialogue* is to present to the user a structured way of passing information to perform actions. This structured communication is realised by means of controls.

8.1.1 Anatomy of windows and dialogues

Although from an application user's perspective windows and dialogues appear to be 'solid' objects (Figure 8.1) it is illustrative to have a look at a window from a different perspective.

A window is composed of three layers, see Figure 8.2. The bottom layer, the *document layer* is formed by the rendered document, the `Picture`. The middle layer, the *control layer* contains all controls of the window. The top layer, the

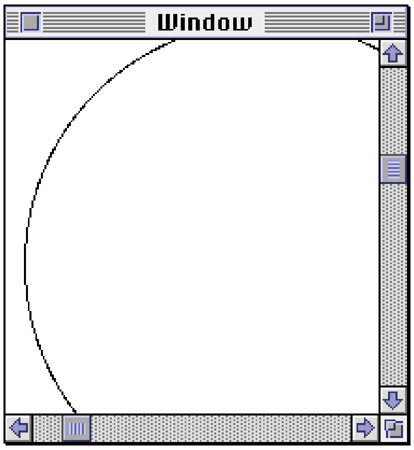


Figure 8.1: A window seen from the user perspective.

window frame typically consists of a title bar, and window components to close and resize the window. The window frame can have any size and is in general smaller than the document layer. It displays only that part of the document layer that is within the window frame. Windows usually contain scrollbars to help the user change the current view on the document layer. The default drawing domain of the document layer `Picture` ranges from 0 to $2^{31}-1$ in both axes. This is in general to large for rendering the document. A window can limit the displayable range of a `Picture` by setting a *picture domain*.

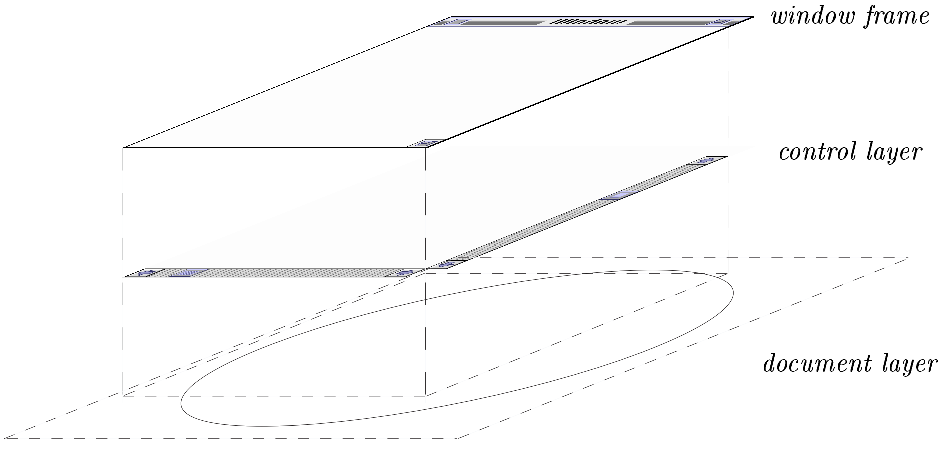


Figure 8.2: A different perspective at a window.

In contrast with windows, a dialogue is composed of only two layers, the control layer and the window frame, the *dialogue frame*. Instead of a document layer, a dialogue has a platform dependent background. The program can not draw into nor navigate the background. The dialogue frame can not be resized by the user and is usually big enough to display the whole control layer.

For both windows and dialogues, the programmer has full control over the view frame, control layer, and document layer. Section 8.3 discusses how to handle the document layer. Handling the control layer is discussed in Section 8.4. The window and dialogue frame are controlled by the platform, see Section 8.5.

8.1.2 Stacking order

In general, a program can have an arbitrary number of windows and dialogues opened at the same time. These elements appear in a *stacking order*, seen by the application user in top to bottom order. For normal windows and dialogues the stacking order is not fixed. *Modal* dialogues however always appear topmost. Windows and modeless dialogues that are opened while (several) modal dialogues are open always appear below the bottom most modal dialogue.

8.1.3 Active window or dialogue

When a user works with a program, exactly one window or dialogue receives all keyboard and mouse input. This element is called the *active* window/dialogue and it has the *input focus*. With the exception of modal dialogues, the active window/dialogue does not necessarily occupy the top most stacking position.

8.2 Window and dialogue attributes

`WindowAttribute` is the type of window and dialogue attributes. The table below shows which attributes are valid for which element.

WindowAttributes	
For windows and dialogues:	**For windows only:**
`WindowId`	`WindowSelectState`
`WindowPos`	`WindowLook`
`WindowIndex`	`WindowViewDomain`
`WindowSize`	`WindowOrigin`
`WindowHMargin`	`WindowHScroll`
`WindowVMargin`	`WindowVScroll`
`WindowItemSpace`	`WindowMinimumSize`
`WindowOk`	`WindowResize`
`WindowCancel`	`WindowActivate`
`WindowHide`	`WindowDeactivate`
`WindowClose`	`WindowMouse`
`WindowInit`	`WindowKeyboard`
	`WindowCursor`

`WindowId`: This attribute identifies the window/dialogue to which it is associated. If you do not provide a `WindowId`, the object I/O system *open function* creates a fresh `Id` for the window/dialogue.

`WindowPos`: This attribute determines the initial position of the window/dialogue (see also Section 9.3). Relative `Ids` that occur in the `ItemPos` refer to other windows/dialogues. The object I/O system will always place a window/dialogue visibly on the current screen. If you do not provide a `WindowPos` to a window, then the window will be placed at the left top of the screen. If you do not provide a `WindowPos` to a dialogue, then the dialogue will be placed at a position conform the platform user interface (typically centered).

`WindowIndex`: This attribute determines the initial stacking position of the window/dialogue (see also 8.1.2). If no `WindowIndex` attribute is provided, then the window/dialogue will be opened frontmost. Modal dialogues are always opened frontmost.

`WindowSize`: This attribute determines the initial size of the window/dialogue. If no `WindowSize` attribute is provided, then the system will derive a proper size. In case of dialogues the size is determined by the set of controls, margins, and item spaces. In case of windows the size is furthermore determined by the picture view domain.

`WindowHMargin`, `WindowVMargin`: These attributes determine the left-right, and top-bottom margin of a window/dialogue respectively. Their default value in case of windows is zero, and platform dependent for dialogues.

`WindowItemSpace`: This attribute determines the space between controls if no further offsets are provided in the layouts of controls. The default values are identical for windows and dialogues.

`WindowOk`, `WindowCancel`: These attributes indicate which control should act conform the platform user interface 'confirm' control and 'cancel' control respectively. If such an attribute is not provided, then no control is selected.

`WindowHide`: This attribute makes the given window/dialogue initially invisible. This attribute is ignored for modal dialogues. If the `WindowHide` attribute is not provided, then the window/dialogue will be opened visible.

`WindowClose`: This attribute adds the platform dependent close control to the window/dialogue. The associated function will be evaluated in case this control is triggered. Actually closing the window/dialogue is the responsibility of this function. In case no `WindowClose` attribute is provided, the window/dialogue can not be closed in that way.

`WindowInit`: This attribute defines a list of actions that should be performed immediately after opening the window/dialogue. This is equivalent to the process initialisation actions (see Section 2.3 and 13). If no `WindowInit` attribute is provided, no additional actions are performed.

`WindowSelectState`: This attribute defines whether the window can be used by the user (`Able`) or not (`Unable`). Dialogues are always `Able`. The default value is `Able`. Note that although a window can be active, it can also be disabled. This only means that all input is ignored by the window.

`WindowLook`: This attribute defines a function that, given the current `SelectState` of the window and information about which part of the window should be displayed, defines what the window should look like. (The `Look` function also plays a role for `CustomButtonControls` (Section 9.1.9), `CustomControls` (Section 9.1.10), and `CompoundControls` (Section 9.1.11).

`WindowViewDomain`: This attribute defines the drawing coordinate system of the document layer (see also 8.1.1). Drawing operations outside this area will be clipped. If no `ViewDomain` is provided, then the window will obtain the `ViewDomain` {`corner1=zero, corner2=`{`x=maxint,y=maxint`}}.

`WindowOrigin`: This attribute determines the initial position of the `ViewFrame`, the rectangular part of the `ViewDomain` that is currently visible in the window. This position is always verified to be within the given `ViewDomain`. If no `WindowOrigin` attribute is provided, then the left-top coordinates of the `ViewDomain` are used.

`WindowHScroll`, `WindowVScroll`: These attributes add a horizontal scrollbar and a vertical scrollbar to the window. If no attribute is given, no scrollbars are added.

`WindowMinimumSize`: This attribute determines the minimum size the window can obtain by resizing. If no attribute is given, a platform dependent minimum size is used.

`WindowResize`: This attribute allows the user to resize the window. If the attribute is not provided, then the window can not be resized. The size of a window may exceed the size of its `ViewDomain`. The area that is not part of the `ViewDomain` will be filled with a platform dependent background.

`WindowActivate`, `WindowDeactivate`: These attributes define the behaviour of the window in case the window becomes the active window (`WindowActivate`), and is no longer the active window (`WindowDeactivate`) respectively (see also 8.1.3). If no attribute is provided, this information will not be passed to the program.

`WindowMouse`, `WindowKeyboard`: These attributes allow a window to respond to user actions with the mouse (`WindowMouse`) and keyboard (`WindowKeyboard`). If no attribute is provided, then this information will not be passed to the program. Both attributes can define an additional filter to ignore some input actions. If the `SelectState` of the window is `Unable` then neither function will obtain input.

`WindowCursor`: This attribute defines the shape of the cursor in case the mouse is over the window and not inside a control that may overrule this shape. In case no attribute is provided moving the mouse over the window will not change its shape.

8.3 Handling the document layer

The document layer of a window is used to present the user visual feedback on the current status of the document that is being manipulated. Only windows have a document layer.

8.3.1 Indirect rendering

Two `WindowAttributes` play a paramount role with respect to the document layer: `WindowViewDomain` and `WindowLook`. Let's have a closer look at them.

The document layer is rendered using a `Picture`. As we have seen in Section 8.2, the default drawing range of a `Picture` is $(0, 2^{31} - 1)$ in both axes. This range can be changed by the `WindowViewDomain` attribute. It has a `ViewDomain` argument which is defined as a `Rectangle`. A `Rectangle` is a record:

```
::  Rectangle
    =   {   corner1 :: !Point
        ,   corner2 :: !Point
        }
::  Point
    =   {   x       :: !Int
        ,   y       :: !Int
        }
```

consisting of the two diagonally opposite corner points of the new drawing range. It is illegal to have identical x or y coordinates. This will cause a run-time error of the application. All drawing that occurs outside of the view domain of the document layer will be clipped.

The `Picture` of the document layer is rendered using the `WindowLook` attribute. This attribute has a `Look` function argument. It is defined as follows:

```
:: Look :== SelectState -> UpdateState -> [DrawFunction]
```

Whenever it is necessary to render (part of) the visible document layer, the object I/O system will apply the look function of the `WindowLook` attribute. It will be parameterised with the current `SelectState` of the window and detailed information about which part of the current view frame needs to be rendered. This information is presented by means of the `UpdateState` record:

```
:: UpdateState
 = { oldFrame  :: !ViewFrame
   , newFrame  :: !ViewFrame
   , updArea   :: !UpdateArea
   }
:: ViewFrame   :== Rectangle
:: UpdateArea  :== [ViewFrame]
```

This record consists of three fields:

`oldFrame`: If the size or orientation of the window was changed, then this field contains the view frame *before* that change. If this is not the cause, then this field contains the *current* view frame.

`newFrame`: If the size or orientation of the window was changed, then this field contains the view frame *after* that change (so, the current view frame). If this is not the cause, then this field contains the *current* view frame.

`updArea`: This field contains a list of `Rectangle`s (not necessarily disjoint) that define the parts of the visible *current* view frame that need to be drawn. This list always consists of atleast one element. Each of its elements is always completely inside the current view frame.

The result of the `Look` function is a list of drawing functions. These describe the rendering actions that should be taken on the current `Picture`. They will be evaluated in left-to-right order. Before doing this the Object I/O system first erases the rectangles of the `updArea` field.

The purpose of the `WindowLook` attribute function is to describe the look of the *current state* of the document layer. If the document does not change, then this function always correctly renders the document. However, it is very likely that the state of the document changes during the life-cycle (Section 2.2) of the window. The `WindowLook` attribute can be changed using the `StdWindow` function `setWindowLook` (Appendix A.32):

```
setWindowLook :: !Id !Bool !Look !(IOSt .l .p) -> IOSt .l .p
```

`setWindowLook` will change the current `WindowLook` attribute of the window indicated by the `Id` argument with the new `Look` function provided this window is present and does not refer to a dialogue. If the `Bool` argument is `False` then that's all. If the `Bool` argument is `True`, then the look function will be applied to the window in the way described above.

8.3.2 Direct rendering

Instead of using only the Look function of a window, one can also draw directly in the document layer Picture. This is done using the function drawInWindow (Appendix A.32):

```
drawInWindow :: !Id ![DrawFunction] !(IOSt .l .p) -> IOSt .l .p
```

drawInWindow applies the list of drawing functions in left-to-right order to the document layer Picture of the window indicated by the Id argument if this window is present and does not refer to a dialogue. For some visual feedback such as drawing blinking cursors or track boxes this method is easier to use than the indirect way of using the Look function. So direct drawing changes the document layer Picture, but does not change the Look function of the window!

8.3.3 Pragmatics

The WindowLook function is used by the object I/O system for all cases that the content of the window needs to rendered. Among others, causes are when the view frame, size, stacking order, or selectstate of a window changes. It is very annoying for the application user when these actions take to much time. Therefore it is worth your while to spend some effort in getting a good performance out of the list of drawing functions.

8.4 Handling the control layer

The control layer contains the controls of a window or dialogue. Both interface elements can have the same set of controls. This is made explicit by the type constructor class instance declarations of their respective creation functions (Appendix A.32):

```
class Windows wdef
where
    openWindow      :: .ls !(wdef .ls (PSt .l .p)) !(PSt .l .p)
                    -> (!ErrorReport, !PSt .l .p)
    ...

class Dialogs wdef
where
    openDialog      :: .ls !(wdef .ls (PSt .l .p)) !(PSt .l .p)
                    -> (!ErrorReport, !PSt .l .p)
    openModalDialog :: .ls !(wdef .ls (PSt .l .p)) !(PSt .l .p)
                    -> (!ErrorReport, !PSt .l .p)
    ...

instance Windows (Window c) | Controls c
instance Dialogs (Dialog c) | Controls c
```

The standard set of Controls instances is defined in the module StdControlClass (Appendix A.4). Many operations with respect to the control layer are identical to operations on controls that are element of CompoundControls. Controls and their operations are discussed in detail in Chapter 9.

8.5 Handling the window and dialogue frame

The window and dialogue frame are the 'physical' borders of windows and dialogues. A user can grab them using the mouse or some keyboard interface and drag them around, change the size (in case of windows), or dispose of them. The program has very limited influence on both the appearence and functionality of the frame. When *opening* a window or dialogue, the `WindowAttributes` are important. This is discussed in Section 8.5.1. User actions on the window or dialogue frame are handled by the program via the callback function mechanism. The program can also change frame properties. This is discussed in Section 8.5.2.

8.5.1 Opening a window or dialogue frame

The attributes of a window and dialogue definition that influence the window and dialogue frame are ofcourse the `Title` and the following `WindowAttributes`:

`WindowSize`: This value gives the prefered size of the view frame of the window or dialogue. Be aware that this is not exterior size, which is in general larger and depends on the underlying platform.

`WindowClose`: If this attribute is present, then the window or dialogue frame is provided with a platform dependent interface element to allow the user to request the program to close that window or dialogue.

`WindowHScroll` and `WindowVScroll`: Although the horizontal and vertical scrollbar of a window are element of the control layer (Figure 8.2), their behaviour is intimately connected with the size of the view frame and therefore also with the size of the window and dialogue frame.

`WindowResize`: If this attribute is present, then the window is provided with a platform dependent interface element to allow the user to change the size of the window view frame.

These are not the only attributes that affect the size of the window frame. The *document interface* (Chapter 13) of the parent interactive process also influences its size and functionality. For instance, on a Windows(95/NT) platform, if a window belongs to a *single document process* its window frame contains all menus (Chapter 10) of that process. If it belongs to a *multiple document interface process* then the size of its frame even depends on its zoom state.

8.5.2 Changing a window and dialogue frame

The two most apparant changes to the window or dialogue frame are changes of orientation and size. In case of windows, these can be caused by the application user, depending on the attributes as explained in Section 8.5.1.

The program can also change the size of the frame for both windows and dialogues. For dialogues this can be done only indirectly by opening or closing controls. For windows this can also be done directly. If the program changes the view frame (either its size or orientation) then the layout of controls is changed in the same way as if the user had caused this change.

8.6 Handling keyboard and mouse input

Of all windows and dialogues that are in control by an application, at most one receives the keyboard and mouse input. This window is the *active window*.

Except when modal dialogues are open, the application user can always select one of the visible windows or dialogues to become the new active window. Dialogues are not notified of these changes. Windows can be notified by adding two `WindowAttributes`: `WindowActivate` and `WindowDeactivate`. Both attributes are parameterised with a function that will be applied by the object I/O system as soon as that window becomes active or has become inactive respectively. It is guaranteed that the `WindowDeactivate` attribute function is applied before the `WindowActivate` function.

The underlying platform always gives visual clues to the application user about which window or dialogue is currently active. The program can retrieve this information using the `getActiveWindow` function (Appendix A.32). One should be aware that it is not correct to assume that the active window or dialogue has the topmost stack order position. As an example, one might try to get the `Id` of the active window by applying `hd` twice to the result list of `getWindowStack`, but this only returns the top most window and not the active window.

The program can also activate windows and dialogues. This is done with the `activateWindow` function (Appendix A.32). Because modal dialogues are always front-most and the front most modal dialogue is active, one can not activate a window or modeless dialogue while modal dialogues are open. Instead, `activateWindow` restacks such a window or dialogue immediately behind the bottom most modal dialogue *without* making it the active window.

The active window receives all keyboard and mouse input. If this window contains controls it can be the case that the input is channelled to one of these controls. That particular control then has the *input focus*. If no control has the input focus, then all input is handled by the active window. In case the active window is a dialogue its response to input is defined entirely by the underlying platform. In case of windows the program can customise the behaviour by adding a `WindowKeyboard` attribute for keyboard input (Section 8.6.1) and by adding a `WindowMouse` attribute for mouse input (Section 8.6.2). Both attributes have a filter function (`KeyboardStateFilter` and `MouseStateFilter` respectively) which is applied before the actual callback function is evaluated. Only if the filter returns `True` then the callback function is evaluated.

8.6.1 Keyboard input

Every keyboard sensitive interface object has a `KeyboardFunction` which is a process state transition function that receives, as a first argument, a value of type `KeyboardState`. This value represents one keyboard event. Keyboard events are always generated in sequences that are characterised by a value of type `KeyState` in the following order:

$$\texttt{(KeyDown False)}\ \{\texttt{(KeyDown True)}\}^{*}\ \texttt{KeyUp}$$

A keyboard event is either an ASCII character (`CharKey` alternative) or a special key (`SpecialKey` alternative).

```
:: KeyboardState
```

```
 = CharKey    Char       KeyState
 | SpecialKey SpecialKey KeyState Modifiers
```

The special keys are imported via the module `StdIOCommon` (Appendix A.12). Among others they define the function keys, arrow keys, page and line keys. The `Modifiers` type is also defined in `StdIOCommon`. It is a record that refers to the state of the meta keys of the keyboard. Because some ASCII characters are generated using these meta keys they are not provided at the `CharKey` alternative. So *shift 'a'* simply generates the `CharKey` alternative with `Char` value `'A'`.

The object I/O system guarantees that at all times only one keyboard alternative is being handled. Assume that a user is pressing the 'a' key on the keyboard. This generates a *character 'a' key down event* (`CharKey 'a' (KeyDown False)`), and then a sequence of *character 'a' repeat key events* (`CharKey 'a' (KeyDown True)`). If the user now also presses the 'b' key the object I/O system inserts two virtual events that force the program to believe that the user first released the 'a' key with a *character 'a' key up event* (`CharKey 'a' KeyUp`), and then pressed the 'b' key with a *character 'b' key down event* (`CharKey 'b' (KeyDown False)`). These are followed by *character 'b' repeat key events*.

8.6.2 Mouse input

Every mouse sensitive interface object has a `MouseFunction` which is a process state transition function that receives, as a first argument, a value of type `MouseState`. This value represents one mouse event.

```
:: MouseState
    =   MouseMove   Point Modifiers
    |   MouseDown   Point Modifiers Int
    |   MouseDrag   Point Modifiers
    |   MouseUp     Point Modifiers
```

The `Point` and `Modifiers` types are defined in the module `StdIOCommon` (Appendix A.12). The `Point` type constructor states the position of the mouse at the mouse event in terms of the view frame coordinates of the interactive object that contains the specific `MouseFunction`. The `Modifiers` type constructor is a record that refers to the state of the meta keys of the keyboard that were pressed at the mouse event.

Mouse events are always generated in sequences that are characterised by the alternative constructor of the `MouseState` type constructor:

$$\{\texttt{MouseMove}\}^* \ [\texttt{MouseDown} \ \{\texttt{MouseDrag}\}^* \ \texttt{MouseUp}]$$

The `Int` argument of the `MouseDown` alternative gives the number of times the mouse was down within the *mouse double down time*. The mouse double down time is a platform dependent time interval that distinguishes two sequential mouse down from a double click. Although an integer is used for this count, its maximum value is usually three. If a mouse down event with count i has occured and a new mouse down event is generated within the mouse double down time, then the next mouse down event has count $i+1$. If the next mouse down event is not generated within the mouse double down time, then the next mouse down event has count 1.

The object I/O system guarantees that every `MouseFunction` of a mouse sensitive interface object is applied to a sequence of mouse events as characterised above. Assume that a certain window is active and the user is pressing the mouse. This generates first a *mouse down event* (`MouseDown` alternative), followed by a sequence of *mouse drag events* (`MouseDrag` alternative). If for some reason another window is being activated, the object I/O system inserts a virtual event that forces the program to believe that the user has released the mouse button with a *mouse up event* (`MouseUp` alternative). If the new window is also mouse sensitive, then its `MouseFunction` is applied to a new virtual event that forces the program to believe that the user has pressed the mouse again with a *mouse down event* (`MouseDown` alternative). These are followed again by *mouse drag event*s.

Chapter 9

Control handling

The previous chapter introduced windows and dialogues. These top level interface elements can contain *controls*, which are handled in this chapter. Control structures can be hierarchical, i.e. they can be composed of controls themselves. Using controls helps a program to provide a consistent and structured user interface. There are a lot of issues involved when working with controls.

First of all we introduce each control object in Section 9.1. Then the glue is introduced to build larger control structures in Section 9.2. An important aspect of controls is to manage their layout, presented in Section 9.3. Related to layout is what should happen in case a window containing controls is resized. This is discussed in Section 9.4. Finally, Section 9.5 contains a number of examples that demonstrate the use of controls.

9.1 The standard controls

Table 2.1 (page 10) shows the standard set of object I/O library controls. They can be divided into three groups:

Platform standard controls: these are the controls that exist on all platforms and that have a well-defined behaviour and look that is platform defined. In the table these are the first seven controls (`RadioControl` ... `ButtonControl`).

Customised controls: the look and feel of these controls is completely or partially defined by the program. In the table these are the `CustomButtonControl` (look is defined by the program, but it feels like a button) and `CustomControl` (look and feel defined by the program).

Hierarchical controls: there is actually only one such control, the `CompoundControl`. It contains other controls which will be positioned relative to this control. Except that it is hierarchical it can also have a look and feel.

There is an extensive set of control attributes that are shared by most controls. `CompoundControls` have an additional set of attributes that are strongly related to window and dialogue attributes. These will be discussed in Section 9.1.11. Before we discuss each of the controls individually we pay some attention to the shared control attributes.

9.1.1 The shared control attributes

The `ControlId` attribute identifies the control to which it is associated. If you do not provide a `ControlId`, the control can not be modified.

The `ControlPos` attribute determines its layout position (see Section 9.3). If you do not provide a `ControlPos`, the control will be placed right next to the previous control (if it happens to be the first control, it will be positioned at the left-top).

The `ControlSize` attribute provides the initial size of the control. For all platform standard controls except the slider control, the size can be derived from their demanded attributes. So if you do not provide a `ControlSize`, then this is calculated by the system. If you do provide one then this value will override the system calculated size (again, except for slider controls). For customised controls the size is mandatory. For compound controls the size can be calculated given the control elements of the compound control. So, if no `ControlSize` is given, the compound control will exactly fit its element controls. If a `ControlSize` is given, then this value overrides the system calculated size.

The `ControlMinimumSize` attribute defines the minimum size of the control. This value is relevant in case of resizing (see Section 9.4). If no `ControlMinimumSize` is provided, the default value zero is chosen.

The `ControlResize` attribute defines that the control is resizeable. If no `ControlResize` is provided, the control is not resizeable. See Section 9.4 about the resizing behaviour of controls.

The `ControlSelectState` attribute defines whether the control can be used by the user (`Able`) or not (`Unable`). Usually this will affect the look of the control. The default value is `Able`.

The `ControlHide` attribute defines that the control is initially invisible. It does occupy space. If you do not provide this attribute then the control is visible.

The `ControlFunction` and `ControlModsFunction` attributes are the primary callback function attributes of controls. The difference between these two attributes is that the former is simply evaluated whenever the control is selected, and that the latter also provides the callback function with the modifier keys that have been pressed at the moment of selecting the control (for the definition of the modifier keys, see definition module `StdIOCommon`). In an attribute list the first of the two attributes is chosen.

The `ControlMouse` and `ControlKeyboard` attributes add mouse and keyboard handling callback functions to the control. This is only possible for non platform standard controls because platform standard controls have a predefined behaviour.

9.1.2 The RadioControl

A *radio control* is a group of radio control items of which exactly one item is selected. All alternatives are visible. The definition of a radio control is as follows:

```
::  RadioControl ls ps
 =  RadioControl [RadioControlItem (ls,ps)] RowsOrColumns Index
                 [ControlAttribute (ls,ps)]
::  RowsOrColumns
 =  Rows    Int
 |  Columns Int
::  RadioControlItem ps :== (TextLine,IOFunction ps)
::  IOFunction       ps :== ps -> ps
```

```
::  TextLine             :== String
::  Index                :== Int
```

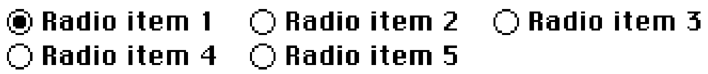
The items are ordered rowwise (the `Rows` alternative of `RowsOrColumns`) or columnwise (the `Columns` alternative of `RowsOrColumns`). The initially selected item is indicated by the `Index` value. As a convention in the object I/O library, when indicating elements indices range from 1 upto the number of elements. So n elements are indexed by $1 \ldots n$. In case the index is out of range, i.e. less than 1 or larger than n, it is set to 1 and n respectively. Valid radio control attributes are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlId`	√	`ControlHide`	√
`ControlPos`	√	`ControlFunction`	
`ControlSize`		`ControlModsFunction`	
`ControlMinimumSize`		`ControlMouse`	
`ControlResize`		`ControlKeyboard`	
`ControlSelectState`	√		

When a radio control item is selected the previously selected radio control item will be unchecked, and the new radio control item gets the check mark. The corresponding callback function is then evaluated. The callback function is also evaluated if the currently selected radio control item is selected.

Example This `RadioControl` consists of five items, layn out in two rows. The first item is initially selected.

```
radiocontrol
  = RadioControl
    [("Radio item "+++toString i,id)\\i<-[1..5]] (Rows 2) 1 []
```



9.1.3 The CheckControl

A *check control* is a group of check control items of which an arbitrary number of items can be selected. All alternatives are visible. The definition of a check control is as follows:

```
::  CheckControl ls ps
 =  CheckControl [CheckControlItem (ls,ps)] RowsOrColumns
                 [ControlAttribute (ls,ps)]
::  RowsOrColumns
 =  Rows    Int
 |  Columns Int
::  CheckControlItem ps :== (TextLine,MarkState,IOFunction ps)
::  IOFunction       ps :== ps -> ps
::  TextLine            :== String
```

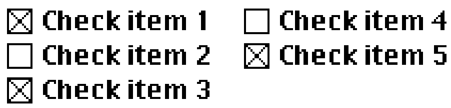
The items are ordered rowwise (the `Rows` alternative of `RowsOrColumns`) or columnwise (the `Columns` alternative of `RowsOrColumns`). The initially selected items are indicated by their `MarkState` value (`Mark` if checked, `Nomark` if not checked). Valid control attributes are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlId`	$\surd$	`ControlHide`	$\surd$
`ControlPos`	$\surd$	`ControlFunction`	
`ControlSize`		`ControlModsFunction`	
`ControlMinimumSize`		`ControlMouse`	
`ControlResize`		`ControlKeyboard`	
`ControlSelectState`	$\surd$		

When a check control item is selected its mark state will be toggled (from `Mark` to `NoMark` and vice versa). No other check control items are affected. The corresponding callback function is then evaluated.

Example This `CheckControl` consists of five items, layn out in two columns. Each odd numbered item has a check mark.

```
checkcontrol
  = CheckControl
    [ ("Check item "+++toString i,if isOdd i Mark NoMark,id)
    \\i<-[1..5]
    ] (Columns 2) []
```



9.1.4 The PopUpControl

A *pop up control* is a group of pop up control items of which exactly one item is selected. The items of a pop up control are presented in a pop up menu. Usually only the currently selected item is displayed in the title of that pop up menu. For this reason pop up controls consume much less space than the functionally equivalent radio controls. The definition of a pop up control is as follows:

```
::  PopUpControl ls ps
 =  PopUpControl [PopUpControlItem (ls,ps)] Index
                 [ControlAttribute (ls,ps)]
::  PopUpControlItem ps :== (TextLine,IOFunction ps)
::  IOFunction       ps :== ps -> ps
::  TextLine            :== String
```

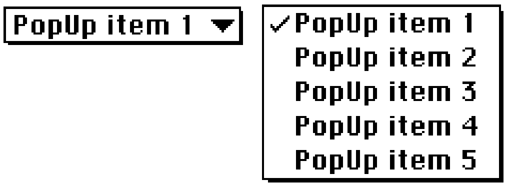
The initially selected item is indicated by the `Index` value. As a convention in the object I/O library, when indicating elements indices range from 1 upto the number of elements. So n elements are indexed by $1 \ldots n$. In case the index is out of range, i.e. less than 1 or larger than n, it is set to 1 and n respectively. Valid control attributes are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlId`	$\surd$	`ControlHide`	$\surd$
`ControlPos`	$\surd$	`ControlFunction`	
`ControlSize`		`ControlModsFunction`	
`ControlMinimumSize`		`ControlMouse`	
`ControlResize`		`ControlKeyboard`	
`ControlSelectState`	$\surd$		

When a pop up control item is selected the previously selected item is unchecked and the new item becomes the selected item. The corresponding callback function is then evaluated. The callback function is also evaluated if the currently selected radio control item is selected.

Example This `PopUpControl` consists of five items. The first item is the initially selected item. The left picture shows the pop up control when not selected by the user, the right picture when selected by the user.

```
popupcontrol
  = PopUpControl
    [("PopUp item "+++toString i,id)\\i<-[1..5]] 1 []
```



9.1.5 The SliderControl

Slider controls are used to change the view to space consuming visual data in one particular dimension. The definition of a slider control is as follows:

```
::  SliderControl ls ps
 =  SliderControl Direction Length SliderState
                 (SliderAction     (ls,ps))
                 [ControlAttribute (ls,ps)]
::  Direction
 =  Horizontal | Vertical
::  Length :== Int
::  SliderState
 =  { sliderMin  :: !Int
    , sliderMax  :: !Int
    , sliderThumb:: !Int
    }
::  SliderAction ps :== SliderMove -> ps -> ps
::  SliderMove
 =  SliderIncSmall | SliderDecSmall
 |  SliderIncLarge | SliderDecLarge
 |  SliderThumb Int
```

A slider control can be layn out in a horizontal direction (the `Horizontal` alternative of `Direction`) or a vertical direction (the `Vertical` alternative of `Direction`). In this direction it can have a certain length. Its width is platform dependent.

The scrolling range of a slider control is defined by the `SliderState` record. The initial slider state determines the integer range: `sliderMin` gives the minimum value, `sliderMax` gives the maximum value. In case these values are given in the wrong order, they will be ordered properly. The initially chosen value is given by the `sliderThumb` value. This value must be inclusively between `sliderMin` and `sliderMax`. If a value smaller than the minimum range is given, then it is set to the minimum. If a value larger than the maximum range is given, then it is set to the maximum.

Valid control attributes for the `SliderControl` are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlId`	√	`ControlHide`	√
`ControlPos`	√	`ControlFunction`	
`ControlSize`		`ControlModsFunction`	
`ControlMinimumSize`		`ControlMouse`	
`ControlResize`	√	`ControlKeyboard`	
`ControlSelectState`	√		

A slider control typically has five regions that can be selected by the user. These regions are shown in Figure 9.1.

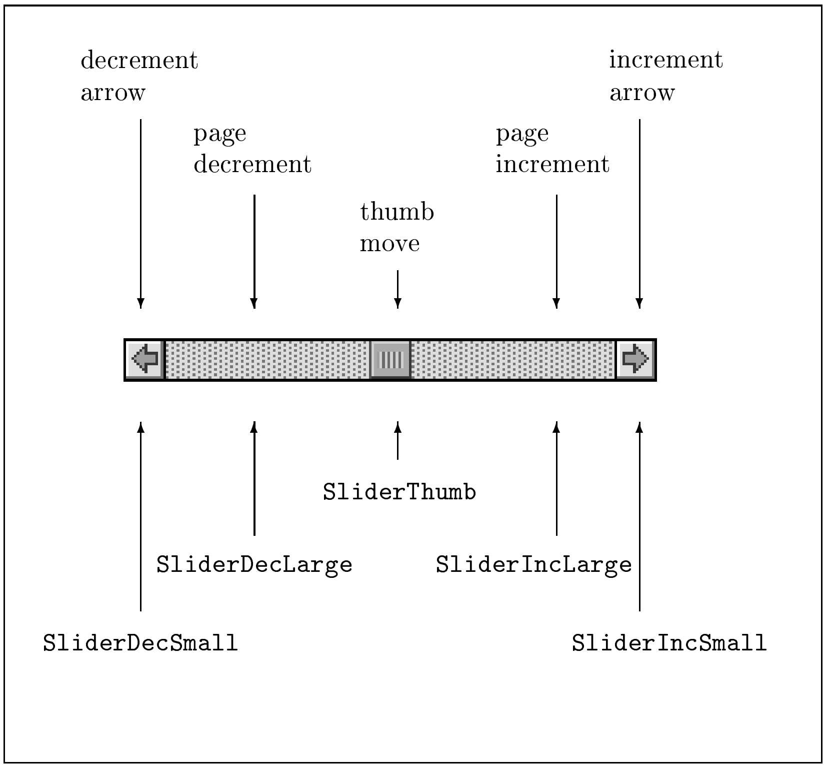


Figure 9.1: The regions of the `SliderControl`.

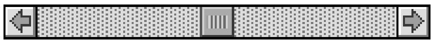
When the user is working with the slider control, its callback function is evaluated. The algebraic data type `SliderMove` has an alternative constructor for each of the regions of the slider control:

`SliderDecSmall`	*decrement arrow*
`SliderIncSmall`	*increment arrow*
`SliderDecLarge`	*page down region*
`SliderIncLarge`	*page up region*
`SliderThumb`	*thumb move*

The program can decide what to do with this information. It is the responsibility of the programmer that the application responds the way the user expects.

Example This `SliderControl` is orientied horizontally and has a length of 200 pixels.

```
slidercontrol
  = SliderControl Horizontal 200
              {sliderMin=(-100),sliderMax=100,sliderThumb=0}
              (\_ ps->ps) []
```



9.1.6 The TextControl

A *text control* displays one line of text that can not be changed by the user. The definition of a text control is as follows:

```
::  TextControl ls ps
 =  TextControl TextLine [ControlAttribute (ls,ps)]
::  TextLine
:== String
```

Currently all control characters such as newlines in the textline argument are ignored. Valid control attributes are:

ControlAttribute:	Valid:	ControlAttribute:	Valid:
`ControlId`	√	`ControlHide`	
`ControlPos`	√	`ControlFunction`	
`ControlSize`	√	`ControlModsFunction`	
`ControlMinimumSize`		`ControlMouse`	
`ControlResize`		`ControlKeyboard`	
`ControlSelectState`	√		

The initial size of a text control is determined by its initial text line. If the text control has a `ControlSize` attribute that is larger than its initial size, then that value becomes its initial size. A text control is not resizeable, and it will also not change in size in case its text line is modified.

Example A text control that contains a text with newline characters.

```
textcontrol
  = TextControl "This is a \ntext control" []
```

9.1.7 The EditControl

The *edit control* is applied to provide the user with an interface to edit (typically small amounts of) textual data. The definition of an edit control is as follows:

```
::  EditControl ls ps
 =  EditControl TextLine Width NrLines
               [ControlAttribute (ls,ps)]
::  TextLine :== String
::  Width    :== Int
::  NrLines  :== Int
```

An edit control initially displays some text line. If the textline contains newlines then these are interpreted as line breaks. The edit control has an initial interior width (in terms of pixels) and shows an integral number of lines. Valid control attributes are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlId`	√	`ControlHide`	√
`ControlPos`	√	`ControlFunction`	
`ControlSize`		`ControlModsFunction`	
`ControlMinimumSize`		`ControlMouse`	
`ControlResize`		`ControlKeyboard`	√
`ControlSelectState`	√		

If the `SelectState` of an edit control is `Unable`, the user can not type in data in edit control. The program can keep track of the inserted text by setting the *control keyboard* attribute. For each typed key the keyboard function is evaluated.

Example An `EditControl` that contains text with newline characters. Its interior width is 80 pixels and it shows three lines of text.

```
editcontrol
  = EditControl "This is an \nEditControl" 80 3 []
```

9.1.8 The ButtonControl

The *button control* represents an action that should occur given the current state of the window or dialogue. The definition of a button control is as follows:

```
::  ButtonControl ls ps
 =  ButtonControl TextLine [ControlAttribute (ls,ps)]
::  TextLine :== String
```

A button control has a name, given by a text line. Control characters are not interpreted. Valid control attributes are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlId`	√	`ControlHide`	√
`ControlPos`	√	`ControlFunction`	√
`ControlSize`	√	`ControlModsFunction`	√
`ControlMinimumSize`		`ControlMouse`	
`ControlResize`		`ControlKeyboard`	
`ControlSelectState`	√		

The initial size of a button control is determined by its initial text line. If the button control has a `ControlSize` attribute that is larger than its initial size, then that value becomes its initial size. The callback function of a button control is the function that is evaluated in case the button control was selected by the user and its `SelectState` was `Able`. If the name of the button control is modified to a new text line, then its size is not changed.

Example A `ButtonControl` with a title that contains newlines.

```
buttoncontrol
  = ButtonControl "This a \nButtonControl" []
```

9.1.9 The CustomButtonControl

A *custom button control* is a control which *feels* like a button control, but which *look* is customised by the program. The definition of a custom button control is as follows:

```
:: CustomButtonControl ls ps
 = CustomButtonControl Size Look [ControlAttribute (ls,ps)]
:: Size
 = {w::Int,h::Int}
:: SelectState
 = Able | Unable
:: UpdateState
 = { oldFrame  :: !ViewFrame
   , newFrame  :: !ViewFrame
   , updArea   :: !UpdateArea
   }
:: Look        :== SelectState -> UpdateState -> [DrawFunction]
:: ViewFrame   :== Rectangle
:: UpdateArea  :== [ViewFrame]
:: DrawFunction:== *Picture -> *Picture
```

Both the initial size and look of a custom button control are defined by the program. The look of a custom button control is identical to the look of a window as discussed in Section 8.3.1. The `UpdateState` argument contains zero based rectangles of the same size as the custom button control itself. Every custom button control has a `*Picture` environment to which the look drawing functions are applied. Valid control attributes are:

ControlAttribute:	Valid:	ControlAttribute:	Valid:
`ControlId`	√	`ControlHide`	√
`ControlPos`	√	`ControlFunction`	√
`ControlSize`		`ControlModsFunction`	√
`ControlMinimumSize`	√	`ControlMouse`	
`ControlResize`	√	`ControlKeyboard`	
`ControlSelectState`	√		

The callback function of a custom button control is the function that is evaluated in case the custom button control was selected by the user and its `SelectState` was `Able`.

Example A `CustomButtonControl` which look depends on its `SelectState`. The picture on the left shows the custom button control in `Able` state, the picture on the right in `Unable` state.

```
custombuttoncontrol
  = CustomButtonControl {w=50,h=50} look []
where
  look :: SelectState UpdateState -> [DrawFunction]
  look Able {newFrame}
    = [ setPenColour DarkGrey, fill newFrame
      , setPenColour Black,    draw newFrame
      ]
  look Unable {newFrame}
    = [ setPenColour LightGrey,fill newFrame
      ]
```

9.1.10 The CustomControl

A *custom control* is a control of which both the look and feel are program defined. The definition of a custom control is as follows:

```
:: CustomControl ls ps
 = CustomControl Size Look [ControlAttribute (ls,ps)]
:: Size
 = {w::Int,h::Int}
:: SelectState
 = Able | Unable
:: UpdateState
 = { oldFrame  :: !ViewFrame
   , newFrame  :: !ViewFrame
   , updArea   :: !UpdateArea
   }
:: Look        :== SelectState -> UpdateState -> [DrawFunction]
:: ViewFrame   :== Rectangle
:: UpdateArea  :== [ViewFrame]
:: DrawFunction:== *Picture -> *Picture
```

Both the initial size and look of a custom control are defined by the program. The look of a custom control is identical to the look of a window as discussed in Section 8.3.1. The `UpdateState` argument contains zero based rectangles of the same size as the custom control itself. Every custom control has a `*Picture` environment to which the look drawing functions are applied. Valid control attributes are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlId`	√	`ControlHide`	√
`ControlPos`	√	`ControlFunction`	
`ControlSize`		`ControlModsFunction`	
`ControlMinimumSize`	√	`ControlMouse`	√
`ControlResize`	√	`ControlKeyboard`	√
`ControlSelectState`	√		

The *feel* of a custom control is defined by its mouse and keyboard callback functions. If the user selects the custom control with the mouse, then the mouse callback function handles all feedback. If the custom control has the keyboard input focus, and the user is typing, then the keyboard callback function handles all feedback.

Example A `CustomControl` which look depends on its `SelectState`. The picture on the left shows the custom control in `Able` state, the picture on the right in `Unable` state.

```
customcontrol
  = CustomControl {w=50,h=50} look []
where
  look :: SelectState UpdateState -> [DrawFunction]
  look Able {newFrame}
    = [ setPenColour DarkGrey, fill newFrame
      , setPenColour Black,    draw newFrame
      ]
  look Unable {newFrame}
    = [ setPenColour LightGrey,fill newFrame
      ]
```

9.1.11 The CompoundControl

The *compound control* is a control that contains other controls. A compound control is actually a window within a window. It introduces a new layout scope: i.e. controls inside it are positioned relative to the bounds of the compound control. The compound control can have an additional look and feel. The definition and `Controls` class instance declaration of a compound control is as follows:

```
::  CompoundControl  c ls ps
 =  CompoundControl (c ls ps) [ControlAttribute (ls,ps)]

instance Controls (CompoundControl c) | Controls c
```

The `c` parameter of the `CompoundControl` type constructor is a type constructor variable that corresponds with the control elements of the compound controls. Any composition of controls that is an instance of the `Controls` type constructor class is a valid argument of `CompoundControl`.

Compared with the previous controls, compound controls have an additional number of control attributes that are irrelevant to the other controls. These attributes are similar to some attributes of windows and dialogues, and ofcourse they intend to have the same meaning. Valid control attributes are:

ControlAttribute:	Valid:	ControlAttribute:	Valid:
ControlId	√	ControlItemSpace	√
ControlPos	√	ControlHMargin	√
ControlSize	√	ControlVMargin	√
ControlMinimumSize	√	ControlLook	√
ControlResize	√	ControlViewDomain	√
ControlSelectState	√	ControlOrigin	√
ControlHide	√	ControlHScroll	√
ControlFunction		ControlVScroll	√
ControlModsFunction			
ControlMouse	√		
ControlKeyboard			

The size of a compound control, if not provided as a `ControlSize` attribute, is derived by the system from its control elements (see for more information on the layout of control Section 9.3). Related to the size of a compound control and layout of its elements are a number of attributes. The `ControlMinimumSize` attribute determines the minimum size of the compound control (see resizing controls, Section 9.4), the `ControlResize` attribute controls the resize behaviour (see also Section 9.4), the `ControlItemSpace`, `ControlHMargin`, and `ControlVMargin` attributes define the distance between the elements themselves and the horizontal and vertical distance of the elements to the border of the compound control respectively.

The compound control anatomy is the same as that of a window (Section 8.1.1). It consists of the same three layers as a window except that we call its top layer the *compound frame* rather than window frame. The compound frame has no title nor features like resize controls and so on.

Analogous to windows, compound controls have a view domain if the `ControlViewDomain` is given. Otherwise it has a zero based arbitrarily large view domain. A view domain defines a finite area in which can be drawn (see Chapter 6), but also is used as an area to place controls (see Section 9.3). Given a view domain, and a compound control that can in principle be smaller than the view domain, two scrolling attributes can be added: `ControlHScroll` and `ControlVScroll`. These attributes control the current view frame of the compound control. The left top point of the view frame that is currently visible is called the *origin*. This value can be set initially with the `ControlOrigin` attribute.

The document layer of a `CompoundControl` can be drawn into only if *both* the `ControlLook` *and* `ControlViewDomain` attributes are given. In that case the system provides the compound control with a drawing environment. If a `ControlLook` is not given, then the compound control is *transparant*.

The feel of a compound control is ofcourse partially determined by its element controls. If the `ControlMouse` attribute is given, then the compound control can handle all mouse events that are directed to one of its element controls. This is not the case for keyboard input.

An example of a `CompoundControl` is given once we have discussed the means to create compositions of controls in the next section.

9.2 Control glue

In the previous section the standard set of controls has been discussed. This list does not cover all controls class instances. In the library module `StdControlClass` (Appendix A.4) a number of additional instances are defined, namely the type constructors `AddLS`, `NewLS`, `ListLS`, `NilLS`, and `:+:` (their definition can be found in Appendix A.12). These additional instances are required to *glue* controls. They are treated below.

9.2.1 `:+:`

The most common constructor to glue controls is `:+:`. Its type constructor definition and `Controls` class instance declaration are as follows:

```
::  :+: t1 t2 local context
 = (:+:) infixr 9 (t1 local context) (t2 local context)

instance Controls ((:+:) c1 c2) | Controls c1 & Controls c2
```

Given two `Controls` instances `c1` and `c2`, working on the same local state of type `local` and context state `context`, the expression `c1:+:c2` is also a `Controls` instance working on the same local state and context state. Because `:+:` is right associative, the expression `c1:+:c2:+:c3` should be read as `c1:+:(c2:+:c3)`.

9.2.2 `ListLS` and `NilLS`

In principle the `:+:` glue is sufficient to create all required control structures. In case of working with a number of control instances of the *same type*, it is much more convenient to use lists and list comprehensions. This glue is provided by the type constructors `ListLS` and `NilLS`. Their type constructor definitions and `Controls` class instance declarations are as follows:

```
::  ListLS t local context = ListLS [t local context]
::  NilLS    local context = NilLS

instance Controls (ListLS c) | Controls c
instance Controls NilLS
```

Given a list of `Controls` instances `cs = ` $[c_1 \ldots c_n]$, working on the same local state of type `local` and context state `context`, the expression `ListLS cs` is also a `Controls` instance working on the same local state and context state. The type constructor `NilLS` is a shorthand for `ListLS []`. It can also be conveniently used to state that a `CompoundControl` or window or dialogue has no controls.

9.2.3 `AddLS` and `NewLS`

The previously discussed glueing type constructors always glue controls that work on the same local state and context state. Two other glueing constructors are defined to extend and change the local state, `AddLS` and `NewLS`. Their type constructor definitions and `Controls` class instance declarations are as follows:

```
:: AddLS t local context
 = E..add:
   { addLS :: add
   , addDef:: t *(add,local) context
   }
:: NewLS t local context
```

```
 = E..new:
   { newLS :: new
   , newDef:: t   new        context
   }

instance Controls (AddLS c) | Controls c
instance Controls (NewLS c) | Controls c
```

Given a `Controls` instance `c1` that works on a local state of type `local` and a context state of type `context`, one can add another `Controls` instance `c2` that works on an *extended* local state of type `(add,local)` and the same context state of type `context`. Let `x` be a value of type `add`, then this is done by the expression `c1 :+: {addLS=x, addDef=c2}`.

Given a `Controls` instance `c1` that works on a local state of type `local` and a context state of type `context`, one can add another `Controls` instance `c2` that works on a *new* local state of type `new` and the same context state of type `context`. Let `x` be a value of type `new`, then this is done by the expression `c1 :+: {newLS=x, newDef=c2}`.

In both cases the extended part of the local state and the new local state are encapsulated completely from the external context using existential quantification.

9.2.4 Example: a counter control

In this section we show an example of glueing controls to form a new `Controls` class instance. A `CompoundControl` is defined that consists of three other controls. It implements a manually incrementable counter. To display the current count value it uses an `Unable EditControl`. Two `ButtonControls` are used to decrement and increment the counter.

```
counter windowid displayid
  = { newLS =initcount
    , newDef=CompoundControl
             (   EditControl (toString initcount) (hmm 50.0) 1
                                  [ControlSelectState Unable
                                  ,ControlPos         (Center,zero)
                                  ,ControlId          displayid
                                  ]
             :+: ButtonControl "-" [ControlFunction    (count (-1))
                                  ,ControlPos         (Center,zero)
                                  ]
             :+: ButtonControl "+" [ControlFunction    (count 1)]
             )   []
    }
where
  initcount = 0

  count :: Int (Int,PSt .l .p) -> (Int,PSt .l .p)
  count dx (count,ps)
    = (count+dx, setText windowid displayid (count+dx) ps)

  setText :: Id Id x (PSt .l .p) -> PSt .l .p | toString x
  setText wid cid x ps
```

```
    = appPIO
        (setWindow wid [setControlTexts [(cid,toString x)]]) ps
```

Figure 9.2 shows the counter after some user manipulations that resulted in the counter value `-15`.

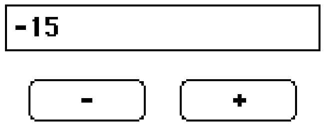


Figure 9.2: The counter control.

9.3 Control layout

The object I/O library offers the programmer an expressive layout mechanism to define the layout of controls. Controls can be element of windows, dialogues, and compound controls. The layout rules that are discussed in this section are followed in all these cases. Some layout rules refer to the view domain and frame of the parent object. In case of dialogues these two are identical. They are zero based rectangles with a size equal to the dialogue interior.

As we have seen, every control can have a `ControlPos` attribute. This attribute is defined as follows:

```
::  ControlAttribute ps
 =  ... | ControlPos ItemPos | ...
::  ItemPos
:== (ItemLoc,ItemOffset)
::  ItemLoc
 =  Fix Point
 |  LeftTop    | RightTop    | LeftBottom | RightBottom
 |  Left       | Center      | Right
 |  LeftOf Id  | RightTo Id  | Above Id   | Below Id
 |  LeftOfPrev | RightToPrev | AbovePrev  | BelowPrev
::  ItemOffset
:== Vector
```

The layout position of a control consists of two values: an `ItemLoc` value and an `ItemOffset` value which is a vector. The `ItemLoc` actually determines the location of the control, the `ItemOffset` value adds an offset to this position (which ofcourse influences the position of other controls). The `ItemLoc` values can be divided into four groups:

Fixed position: this is actually only the `Fix` alternative of `ItemLoc`. `Fix point` places the left top corner of the control of which it is the attribute at the specified `point`. The `point` is given in the coordinate system of the compound control. This coordinate system is determined by the current *view domain* of the parent object.

Boundary aligned: these are the alternatives `LeftTop` ...`RightBottom`. When applied their control is placed at the left-top, right-top, left-bottom, or right-bottom of the current *view frame* of the parent object.

Line aligned: these are the alternatives `Left` ...`Right`. When applied their control is placed below all previous line aligned controls and either left-aligned, centered, or right-aligned with respect to the current *view frame* of the parent object.

Relative position: these are the remaining alternatives `LeftOf` ...`BelowPrev`. The first four alternatives must be parameterised with the `Id` of a control that is element of the same parent object, otherwise a runtime error will occur. The latter four alternatives can be defined in terms of the first four but have the advantage that you do not have to think of `Id`s for controls that you only want to relatively place other controls to. Placing controls relatively to other controls must construct a *tree* of related controls: cyclic references are not allowed and result also in a runtime error.

Controls that are layn out relatively form a *layout tree* with one *layout root* control. The layout attribute of the layout root control determines the layout positions of the whole layout tree. If it is at a fixed position, the layout tree obtains a fixed position. If it is boundary aligned, the layout tree is aligned at the same boundary. If it is line aligned, the layout tree is line aligned.

Except for the *first* layout root control the default layout attribute for controls is `(RightToPrev,zero)`. For the first layout root control the default attribute is `(Left,zero)`. Consequently, the default layout order is from left to right in one single row.

Controls are allowed to overlap partially or completely. This is particularly useful in case of combinations of hidden and visible controls when at all times only one is visible. It allows the program to change the control structure in an easy way by hiding and showing controls.

In the remaining part of this section a number of examples are given to illustrate control layout. In each of the examples we assume that the controls that are being layn out are placed in a parent object with a view *domain* and view *frame* as given in Figure 9.3. The x axis (the horizontal arrow) and y axis (the vertical arrow) intersect at the coordinate `zero`.

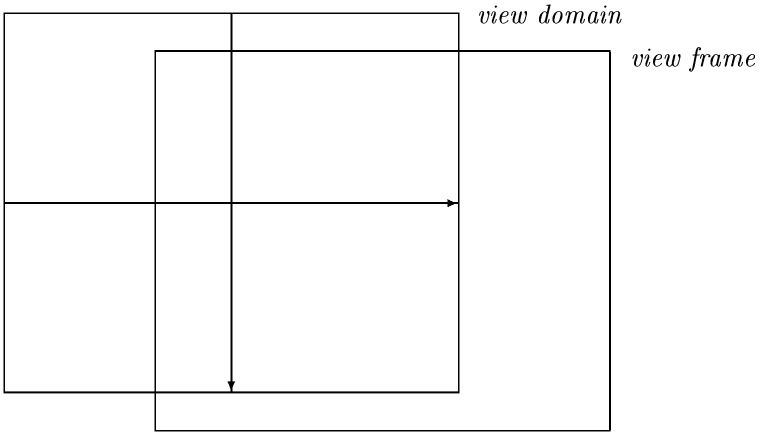


Figure 9.3: View domain and view frame.

Controls are being displayed as boxes. The control configuration shown in Figure 9.4 occurs frequently in the examples. It consists of five equally sized controls,

$c_0 \ldots c_4$. The layout root control is c_0. The controls $c_1 \ldots c_4$ are layn out relatively to c_0 and have the layout attributes `LeftOf`, `RightTo`, `Above`, and `Below` respectively with `zero` offsets.

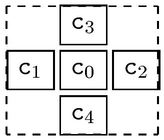


Figure 9.4: A layout tree of five controls.

9.3.1 Layout at fixed position

Controls and layout trees that have a `Fix` layout attribute are being placed relative to the view domainof the parent object. So their visibility depends on the current orientation of the parent view frame (recall that the view frame clips everything that is outside of it). Figure 9.5 shows the control configuration of Figure 9.4 when the layout root control has the attribute `(Fix zero,zero)`.

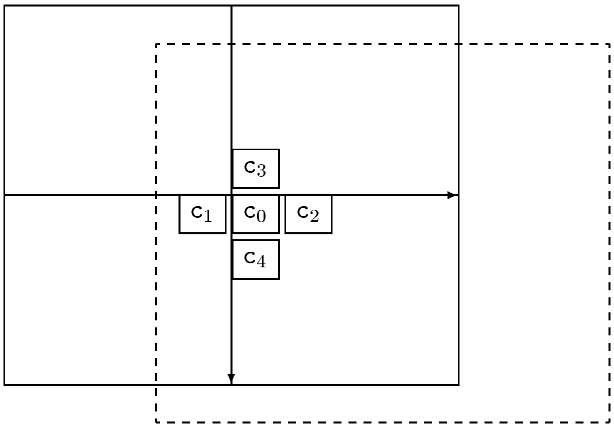


Figure 9.5: The layout tree at `(Fix zero,zero)`.

9.3.2 Layout at view frame boundary

Controls and layout trees that are laynout relative to the parent view frame boundary will always be visible, provided ofcourse that the view frame is suffiently large. If the view frame is not large enough, these controls may become overlapped. Figure 9.6 shows the positions of the control configuration of Figure 9.4 when positioned at every corner of the view frame (using `LeftTop`, `RightTop`, `LeftBottom`, and `RightBottom` with `zero` offsets.

9.3.3 Layout in lines

Laying out controls and layout trees in lines is similar to writing characters in an English piece of text: each new character is placed right next to the previous

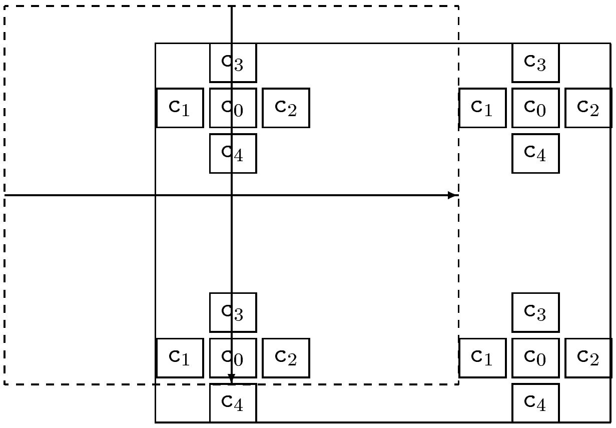


Figure 9.6: The layout tree at `LeftTop`, `RightTop`, `LeftBottom`, and `RightBottom`.

character until a new line is started. A new line starts below the previous line. This new line can be *left* aligned, *centered*, or *right* aligned. The layout attributes `Left`, `Center`, and `Right` introduce both a new line and its alignment. Figure 9.7 shows the positions of the control configuration of Figure 9.4 when positioned at `Left`, `Center`, and `Right` respectively, using `zero` offsets. It also illustrates that if the view frame is not big enough, controls may become partially invisible.

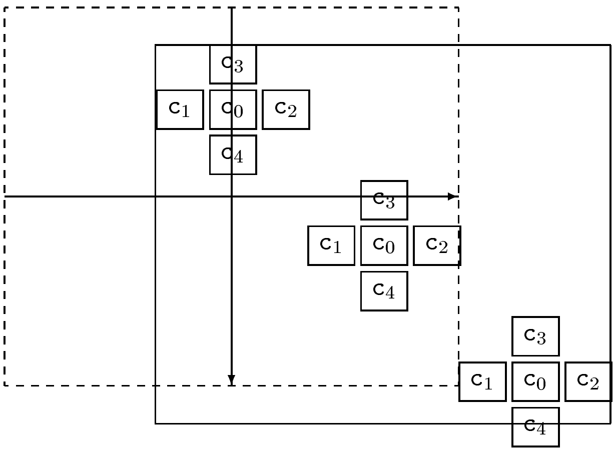


Figure 9.7: The layout tree at `Left`, `Center`, and `Right`.

9.3.4 Layout offsets

So far we have used `zero` offsets in the layout attribute examples. The layout position of a control is changed by an offset vector value `v = {vx,vy}` as follows: first, the layout position of the control is calculated as explained above, using a `zero` offset. Now assume that this results in the *exact* location `pos = {x,y}`. Then the real position of the control is `{x=x+vx, y=y+vy}`. Figure 9.8 illustrates this. Given two controls c_0 and c_1 it shows the result of placing c_1 at `RightTo` control c_0 with an offset value `v = {vx,vy}`. The dashed box shows the location of c_1 using a `zero` offset.

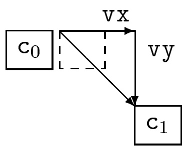


Figure 9.8: Laying out controls using an offset vector.

9.3.5 Layout relative to the previous control

As explained earlier in this section, the default layout attribute of a control is `(RightToPrev,zero)`. The other layout attributes that refer to the previous control are `LeftOfPrev`, `AbovePrev`, and `BelowPrev`. In this section we explain what the previous control is.

Section 9.2 introduced the glue to create control structures. The best way to look at such a control structure is to have a look at its *numbered* graph structure. Consider the following expression: `(a:+:b:+:c)` with `a`, `b`, and `c` standard `Controls` class instances as introduced in Section 9.1. Figure 9.9 shows the graph structure (recall that `:+:` is right associative).

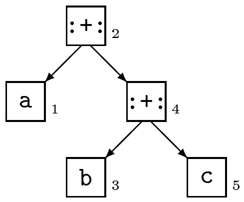


Figure 9.9: The numbered graph of `(a:+:b:+:c)`.

Each node in the graph has an index. If a node is a glue node, then first number the left sub tree, then the node itself, then the right sub tree. If a node is a standard `Controls` class instance, then number it. The nodes of the sub tree of a `CompoundControl` are not numbered. Proceeding in this way, one obtains the index figures at each of the nodes in Figure 9.9. If a node in the graph with index i represents one of the standard `Controls` class instances then its *previous* control is represented by that node in the graph that has the highest index less than i and represents also one of the standard `Controls` class instances. So the previous control of `c` is not $\texttt{:+:}_4$ but $\texttt{b}_3$ because we assumed that `b` is an instance of the standard `Controls` class. Analogously, the previous element of `b` is neither one of the two `:+:` nodes, but $\texttt{a}_1$. Finally, `a` has no previous control.

9.4 Resizing controls

The object I/O system has a simple mechanism to let controls respond to resize actions of their parent interface element (window, dialogue, or compound control). If a control wants to respond to resize events, it should have a `ControlResize` attribute. It is defined as follows:

```
::  ControlAttribute ps
 =  ... | ControlResize ControlResizeFunction | ...
::  ControlResizeFunction
:== Size -> Size -> Size -> Size
```

The control resize function is applied to its current size, old size of its parent, and the new size of its parent. It returns its own new size. This calculation is performed for all controls that are part of the control that is being resized. If a `CompoundControl` has a resize function, and the new size is different from its previous size, then this computation continues recursively, otherwise the layout of its elements is not recalculated. Given the new sizes of the controls, the layout is recalculated and adjusted accordingly. The effect of this strategy is that the relative layout of controls is never changed in case of resizing a window, dialogue, or compound control.

As an example, consider one wants to have a `CompoundControl` that always displays three `CustomControls` next to each other at the top of its view frame. The `CompoundControl` takes care that it always has a width dividable by 3, using its own `ControlResize` function `compoundresize`. Its `ControlLook` function draws a rectangle fitting its current view frame.

```
compound      = CompoundControl
                  (ListLS [custom,custom,custom])
                  [ControlResize     compoundresize
                  ,ControlSize       compoundsize
                  ,ControlLook       (\_ {newFrame}->[draw newFrame])
                  ,CompoundHMargin   0 0
                  ,CompoundVMargin   0 0
                  ,CompoundItemSpace 0 0
                  ]
compoundsize = {w=60,h=75}
compoundresize _ _ newparentsize=:{w}
             = {newparentsize & w=w/3*3}
```

The `CustomControls` resize their widths according to the new width of their parent control, using the `ControlResize` function `customresize`. The look of the `CustomControl` simply draws a rectangle fitting its current view frame and the two diagonals.

```
custom       = CustomControl
                   {w=compoundsize.w/3,h=6}
                   look
                   [ControlResize customresize]
customresize customsize _ {w}
             = {customsize & w=w/3}
look _ {newFrame}
             = [draw     newFrame
               ,drawLine newFrame.corner1
                         newFrame.corner2
               ,drawLine {newFrame.corner1 & y=newFrame.corner2.y}
                         {newFrame.corner2 & y=newFrame.corner1.y}
               ]
```

Figure 9.10 shows what happens with the controls when the parent object is resized. At the left the initial state of the `CompoundControl` and its `CustomControls` is displayed. As explained in Section 9.3, the three custom controls form a layout tree with the layout root control having the layout attribute `(Left,zero)` and the other `CustomControls` `(RightToPrev,zero)`. In the middle, the `CompoundControl` is resized in both directions. This resize action causes first recalculation of the size of the `CompoundControl`, using `compoundresize`. Because this value differs from the old size, recalculation continues for each `CustomControl`. The final result is shown at the right.

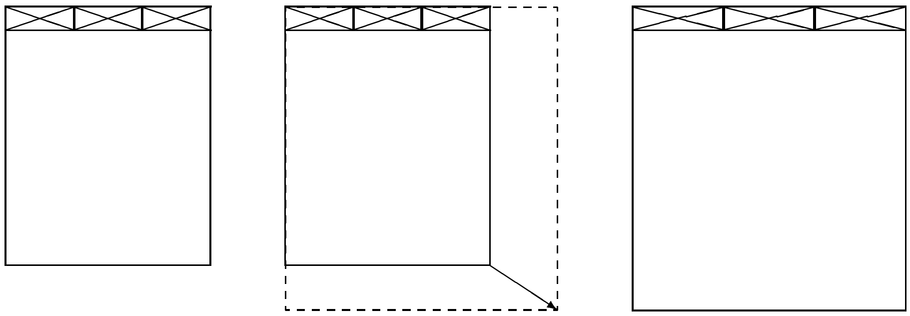


Figure 9.10: Resizing a `CompoundControl` with three `CustomControls`.

9.5 Examples

In this section a number of examples are given to clarify the use of controls.

9.5.1 Keyspotting

To illustrate keyboard handling we discuss a small example program that monitors keyboard events, called 'keyspotting'. It creates a `Window` that contains a `CompoundControl` that contains a `CustomControl`. Before we discuss each of the components below, we have a look at the way they handle keyboard input.

Each of the components is keyboard sensitive and uses the same `KeyboardFunction` `spotting`. Given a string that states who currently has the input focus, `spotting` simply changes the `Look` function of the `CustomControl` and forces its update (by using a `True` boolean). We assume that there is a conversion function that transforms a `KeyboardState` into a `String`. The `Look` function `look` draws the text that it is parameterised with within a framed box.

```
spotting who key ps
  = appPIO (setWindow windowid
             [setControlLooks
               [(controlid,True,look (who+++":"+++toString key))
               ]
             ]
           ) ps

look text _ {newFrame}
  = [setPenColour White
    ,fill         newFrame
    ,setPenColour Black
    ,draw         newFrame
```

```
    ,drawAt       {x=10,y=customsize.h/2} text
    ]
```

The CustomControl custom displays which component is currently receiving what keyboard input. The control is identified by controlid. It parameterises its KeyboardFunction spotting function with the string "Control".

```
custom
  = CustomControl customsize (look "")
    [ControlKeyboard (const True) Able (noLS1 (spotting "Control"))
    ,ControlId       controlid
    ]
```

The CompoundControl compound contains only custom. It parameterises its KeyboardFunction spotting with the string "Compound". Its size is chosen such that it is large enough to display custom completely. It conveniently uses the look function to draw a box around itself.

```
compound
  = CompoundControl custom
    [ControlKeyboard (const True) Able (noLS1 (spotting "Compound"))
    ,ControlSize     {w=customsize.w+2*margin
                     ,h=customsize.h+2*margin
                     }
    ,ControlLook     (look "")
    ]
```

Finally, the Window window contains only compound. It parameterises its spotting function with the string "Window". Its size is chosen such that it is large enough to display compound completely. For this purpose also the margin layout attributes are set. Termination of the program is taken care of by having the program quit when the user closes the window.

```
window
  = Window "keyspotting" compound
      [WindowKeyboard (const True) Able (noLS1 (spotting "Window"))
      ,WindowId       windowid
      ,WindowSize     {w=customsize.w+4*margin
                      ,h=customsize.h+4*margin
                      }
      ,WindowHMargin  margin margin
      ,WindowVMargin  margin margin
      ,WindowClose    (noLS closeProcess)
      ]
```

Remaining details that need to be defined are the actual creation of the interactive program and its window. For completeness, we include the program code of keyspotting below. We assume that the module stringconv contains a string conversion function for KeyboardStates.

```
module keyspotting

//  **********************************************************************************
//  Clean tutorial example program.
```

```
//
//  This program monitors keyboard input that is sent to a Window which consists
//  of a CompoundControl which consists of a CustomControl.
//  ****************************************************************************

import StdEnv,StdIO,stringconv

::  NoState
    =   NoState

Start :: *World -> *World
Start world
    #   (windowid, world)= openId world
    #   (controlid,world)= openId world
    =   startIO NoState NoState [initialise windowid controlid] [] world
where
    initialise windowid controlid ps
        # custom        = CustomControl customsize (look "")
                            [   ControlKeyboard (const True) Able
                                                (noLS1 (spotting "Control"))
                            ,   ControlId       controlid
                            ]
        # compound      = CompoundControl custom
                            [   ControlKeyboard (const True) Able
                                                (noLS1 (spotting "Compound"))
                            ,   ControlSize     {w=customsize.w+2*margin
                                                ,h=customsize.h+2*margin
                                                }
                            ,   ControlLook     (look "")
                            ]
        # window        = Window "keyspotting" compound
                            [   WindowKeyboard  (const True) Able
                                                (noLS1 (spotting "Window"))
                            ,   WindowId        windowid
                            ,   WindowSize      {w=customsize.w+4*margin
                                                ,h=customsize.h+4*margin
                                                }
                            ,   WindowClose     (noLS closeProcess)
                            ,   WindowHMargin   margin margin
                            ,   WindowVMargin   margin margin
                            ]
        # (error,ps)    = openWindow NoState window ps
        | error<>NoError
            =   abort "keyspotting could not open window"
        | otherwise
            =   ps
    where
        customsize  = {w=550,h=100}
        margin      = 10
        spotting who key ps
            =   appPIO (setWindow windowid
                        [setControlLooks [(controlid,True,look text)]]
                        ) ps
        where
            text    = who+++":"+++toString key
        look text _ {newFrame}
            =   [ setPenColour White, fill newFrame
                , setPenColour Black, draw newFrame
                , drawAt {x=10,y=customsize.h/2} text
                ]
```

9.5.2 Mousespotting

To illustrate mouse handling we discuss a small example program that monitors mouse events, called 'mousespotting'. It creates a `Window` that contains a `Compound-`

Control that contains a CustomControl. Before we discuss each of the components below, we have a look at the way they handle mouse input.

Each of the components is mouse sensitive and uses the same MouseFunction spotting. Given a string that states who currently has the input focus, spotting simply changes the Look function of the CustomControl and forces its update (by using a True boolean). We assume that there is a conversion function that transforms a MouseState into a String. The Look function look draws the text that it is parameterised with within a framed box.

```
spotting who mouse ps
  = appPIO (setWindow windowid
             [setControlLooks [(controlid,True,look text)]]) ps
where
  text    = who+++":"+++toString mouse

look text _ {newFrame}
  = [ setPenColour  White, fill newFrame
    , setPenColour  Black, draw newFrame
    , drawAt {x=10,y=customsize.h/2} text
    ]
```

The CustomControl custom displays which component is currently receiving what mouse input. The control is identified by controlid. It parameterises its MouseFunction spotting function with the string "Control".

```
custom
  = CustomControl customsize (look "")
    [ControlMouse (const True) Able (noLS1 (spotting "Control"))
    ,ControlId    controlid
    ]
```

The CompoundControl compound contains only custom. It parameterises its MouseFunction spotting with the string "Compound". Its size is chosen such that it is large enough to display custom completely. It conveniently uses the look function to draw a box around itself.

```
compound
  = CompoundControl custom
    [ControlMouse (const True) Able (noLS1 (spotting "Compound"))
    ,ControlSize  {w=customsize.w+2*margin,h=customsize.h+2*margin}
    ,ControlLook  (look "")
    ]
```

Finally, the Window window contains only compound. It parameterises its spotting function with the string "Window". Its size is chosen such that it is large enough to display compound completely. For this purpose also the margin layout attributes are set. Termination of the program is taken care of by having the program quit when the user closes the window.

```
window
  = Window "mousespotting" compound
    [WindowMouse   (const True) Able (noLS1 (spotting "Window"))
    ,WindowId      windowid
```

```
,WindowSize    {w=customsize.w+4*margin,h=customsize.h+4*margin}
,WindowHMargin margin margin
,WindowVMargin margin margin
,WindowClose   (noLS closeProcess)
]
```

Remaining details that need to be defined are the actual creation of the interactive program and its window. For completeness, we include the program code of `mousespotting` below. We assume that the module `stringconv` contains a string conversion function for `MouseState`s.

```
module mousespotting

//  **********************************************************************************
//  Clean tutorial example program.
//
//  This program monitors mouse input that is sent to a Window which consists
//  of a CompoundControl which consists of CustomControl.
//  **********************************************************************************

import StdEnv,StdIO,stringconv

::  NoState
    =   NoState

Start :: *World -> *World
Start world
    #   (windowid, world)= openId world
    #   (controlid,world)= openId world
    =   startIO NoState NoState [initialise windowid controlid] [] world
where
    initialise windowid controlid ps
        # custom    = CustomControl customsize (look "")
                        [   ControlMouse    (const True) Able
                                            (noLS1 (spotting "Control"))
                        ,   ControlId       controlid
                        ]
        # compound  = CompoundControl custom
                        [   ControlMouse    (const True) Able
                                            (noLS1 (spotting "Compound"))
                        ,   ControlSize     {w=customsize.w+2*margin
                                            ,h=customsize.h+2*margin
                                            }
                        ,   ControlLook     (look "")
                        ]
        # window    = Window "mousespotting" compound
                        [   WindowMouse     (const True) Able
                                            (noLS1 (spotting "Window"))
                        ,   WindowId        windowid
                        ,   WindowSize      {w=customsize.w+4*margin
                                            ,h=customsize.h+4*margin
                                            }
                        ,   WindowHMargin   margin margin
                        ,   WindowVMargin   margin margin
                        ,   WindowClose     (noLS closeProcess)
                        ]
        # (error,ps)    = openWindow NoState window ps
        | error<>NoError
            =   abort "mousespotting could not open window"
        | otherwise
            =   ps
    where
        customsize  = {w=550,h=100}
        margin      = 10
        spotting who mouse ps
```

```
    =   appPIO (setWindow windowid
               [setControlLooks [(controlid,True,look text)]]) ps
where
    text    = who+++":"+++toString mouse
look text _ {newFrame}
    =   [ setPenColour  White, fill newFrame
        , setPenColour  Black, draw newFrame
        , drawAt {x=10,y=customsize.h/2} text
        ]
```

Chapter 10

Menus

Many interactive applications allow a user to manipulate a number of documents. As we saw in the previous chapter, the user can issue these manipulations by means of the keyboard and mouse. Another common source of manipulations is by issueing *commands* to the application. This is where *menus* come in. Menus help a program to structure the set of available commands. To the user of an application, the use of menus provides a consistent and easily browsable graphical display of the set of available commands. For these reasons it is recommended to use menus in a program.

In Section 10.1 we introduce the standard set of menu definitions that are at a programmers disposal. Then the glue is introduced to create larger menu structures in Section 10.2. One special menu is available for interactive processes that have the multiple document interface (MDI) attribute, the *windows* menu. This menu enumerates the current open and visible windows of that process and give some commands to organise them. This is treated in Section 10.3. Menus provide a consistent graphical interface to users. To enhance the consistency, a number of programming conventions have evolved. These are discussed in Section 10.4.

10.1 Menus and menu elements

Menus and menu elements can be defined by means of the type definitions in module `StdMenuDef` (Appendix A.15). Analogous to `Windows`, `Dialogs`, and `Compound-Controls`, `Menus` are parameterised with a type constructor variable. The admissable instances are the *menu elements.* Below we introduce each of the components.

10.1.1 The menu attributes

The menu attributes are used by both menus and menu elements. They are the following:

```
:: MenuAttribute ps
 = MenuId           Id
 | MenuSelectState  SelectState
 | MenuIndex        Int
 | MenuShortKey     Char
 | MenuMarkState    MarkState
 | MenuFunction     (IOFunction     ps)
```

```
 | MenuModsFunction (ModsIOFunction ps)
```

The `MenuId` attribute identifies the menu or menu element to which it is associated. If you do not provide a `MenuId` the menu (element) can not be modified.

The `MenuSelectState` attribute defines whether the menu (element) can be used by the user (`Able`) or not (`Unable`). Usually this will affect the look of the menu (element). The default value is `Able`.

The `MenuIndex` attribute defines the index position of a menu. Index positions range from one (for the first menu) upto the number of menus. A negative or zero `MenuIndex` attribute value will place the menu in front of all current menus. A `MenuIndex` attribute value that is larger than the current number of menus will place the menu behind all current menus. Other `MenuIndex` attribute values place the menu *behind* the menu with that index value.

The `MenuShortKey` attribute defines a character that can be used by user to select the menu element to which the character is associated by means of the keyboard. The menu element can be selected by pressing that character and some special, platform dependent meta key.

The `MenuMarkState` attribute can add a check mark symbol (`Mark`) or leave it out (`NoMark`) to a menu element. The default value is `NoMark`.

The `MenuFunction` and `MenuModsFunction` attributes add callback functions to menu elements that are evaluated when the menu element to which they are associated is selected by the user. The difference between these two attributes is that the former is simply evaluated whenever the menu element is selected, and that the latter also provides the callback function with the modifier keys that have been pressed at the moment of selecting the menu element (for the definition of the modifier). In a `MenuAttribute` list, the first of these two attributes is chosen.

10.1.2 The Menu

A *menu* is a top level interface element that contains a group of related commands. The definition of a menu is as follows:

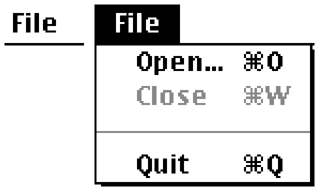
```
:: Menu m ls ps
 = Menu Title (m ls ps) [MenuAttribute *(ls,ps)]
```

The usual appearance of a menu is by its title. The user can browse through its commands by mouse or by a platform dependent keyboard interface. Valid menu attributes are:

MenuAttribute:	Valid:
MenuId	$\surd$
MenuSelectState	$\surd$
MenuIndex	$\surd$
MenuShortKey	
MenuMarkState	
MenuFunction	
MenuModsFunction	

Example Here is an example of a menu. The left picture shows the menu when not selected by the user, the right picture when selected by the user.

```
menu
  = Menu "Menu"
      (   MenuItem "Open..." [MenuShortKey    'o']
      :+: MenuItem "Close"   [MenuSelectState Unable
                             ,MenuShortKey    'w'
                             ]
      :+: MenuSeparator      []
      :+: MenuItem "Quit"    [MenuShortKey    'q']
      )   []
```



10.1.3 The MenuItem

The *menu item* is the standard element that refers to a command. The definition of a menu item is as follows:

```
:: MenuItem ls ps = MenuItem Title [MenuAttribute *(ls,ps)]
```

The title of a menu element is displayed as a member of its parent menu. When the user selects the menu item its `Menu(Mods)Function` attribute is evaluated if the menu item, and all of its parent menus are `Able`. The appearence of the menu item reflects this state. Valid menu item attributes are:

MenuAttribute:	Valid:
`MenuId`	√
`MenuSelectState`	√
`MenuIndex`	
`MenuShortKey`	√
`MenuMarkState`	√
`MenuFunction`	√
`MenuModsFunction`	√

Examples of menu items are given in Section 10.1.2.

10.1.4 The MenuSeparator

The *menu separator* is a menu element that is only used to separate groups of related menu elements within a parent menu. Graphically, a menu separator usually inserts some vertical space within a menu. Menu separators have no further functionality. The definition of a menu separator is as follows:

```
:: MenuSeparator ls ps = MenuSeparator [MenuAttribute *(ls,ps)]
```

Because menu separators have no other purpose than providing some 'white space' between menu elements, the only valid menu separator attribute is the `MenuId`:

MenuAttribute:	Valid:
`MenuId`	√
`MenuSelectState`	
`MenuIndex`	
`MenuShortKey`	
`MenuMarkState`	
`MenuFunction`	
`MenuModsFunction`	

In the menu example of Section 10.1.2 a menu separator has been used.

10.1.5 The RadioMenu

A *radio menu* element is a group of menu items of which exactly one menu item is selected. All alternatives are visible. The definition of a radio menu is as follows:

```
:: RadioMenu  ls ps = RadioMenu [MenuRadioItem *(ls,ps)] Index
                                [MenuAttribute *(ls,ps)]
:: MenuRadioItem ps :== (Title,Maybe Id,Maybe Char,IOFunction ps)
```

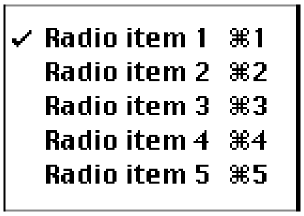
The initially selected item is indicated by the `Index` value. As a convention in the object I/O library, when indicating elements indices range from 1 upto the number of elements. So n elements are indexed by $1 \ldots n$. In case the index is out of range, i.e. less than 1 or larger than n, it is set to 1 and n respectively. Valid radio menu attributes are:

MenuAttribute:	Valid:
`MenuId`	√
`MenuSelectState`	√
`MenuIndex`	
`MenuShortKey`	√
`MenuMarkState`	
`MenuFunction`	
`MenuModsFunction`	

When an item of the radio menu is selected the previously selected radio menu item will be unchecked, and the new radio menu item gets the check mark. The corresponding callback function is then evaluated. The callback function is also evaluated if the currently selected radio menu item is selected.

Example Here is an example of a radio menu and what it initially looks like (it is instructive to compare this with the radio *control* example at page 63).

```
radiomenu
  = RadioMenu
    [  ("Radio item "+++toString i,Nothing,Just (iChar i),id)
    \\ i<-[1..5]
    ]  1 []
where
  iChar i = toChar (toInt '1'+i-1)
```



10.1.6 The SubMenu

The *sub menu* is a menu element that contains other menu elements. So it is a menu within a menu, and ofcourse can contain sub menus as well. The definition of a sub menu is as follows:

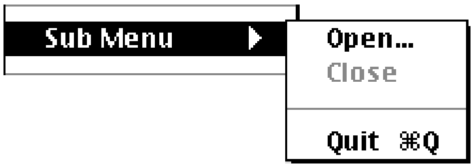
```
:: SubMenu m ls ps = SubMenu Title (m ls ps) [MenuAttribute *(ls,ps)]
```

The usual appearance of a sub menu is by its title. The user can browse through its elements by mouse or by a platform dependent keyboard interface. Valid sub menu attributes are:

MenuAttribute:	**Valid:**
`MenuId`	√
`MenuSelectState`	√
`MenuIndex`	
`MenuShortKey`	
`MenuMarkState`	
`MenuFunction`	
`MenuModsFunction`	

Example Here is an example of a sub menu. It is actually the same definition as the menu at page 88 except that it has a `SubMenu` data constructor rather than the `Menu` data constructor and a different title. The left picture shows the sub menu when not selected by the user, the right picture when selected by the user.

```
submenu
  = SubMenu "Sub Menu"
      (   MenuItem "Open..." [MenuShortKey    'o']
      :+: MenuItem "Close"   [MenuSelectState Unable
                             ,MenuShortKey    'w'
                             ]
      :+: MenuSeparator      []
      :+: MenuItem "Quit"    [MenuShortKey    'q']
      )   []
```



10.2 Menu glue

In the previous section the standard set of menus and menu elements has been discussed. This list is not complete. In the library module `StdMenuElementClass` (Appendix A.17) a number of additional instances are defined, namely the type constructors `AddLS`, `NewLS`, `ListLS`, `NilLS`, and `:+:` (their definition can be found in Appendix A.12). These additional instances are required to *glue* menus. They are treated below.

10.2.1 `:+:`

The most common constructor to glue menu elements is `:+:`. Its type constructor definition and `MenuElements` class instance declaration are as follows:

```
::  :+: t1 t2 local context
 = (:+:) infixr 9 (t1 local context) (t2 local context)

instance MenuElements ((:+:) m1 m2) | MenuElements m1
                                    & MenuElements m2
```

Given two `MenuElements` instances `m1` and `m2`, working on the same local state of type `local` and context state `context`, the expression `m1:+:m2` is also a `MenuElements` instance working on the same local state and context state. Because `:+:` is right associative, the expression `m1:+:m2:+:m3` should be read as `m1:+:(m2:+:m3)`.

10.2.2 `ListLS` and `NilLS`

In principle the `:+:` glue is sufficient to create all required menu element structures. In case of working with a number of menu instances of the *same type*, it is much more convenient to use lists and list comprehensions. This glue is provided by the type constructors `ListLS` and `NilLS`. Their type constructor definitions and `MenuElements` class instance declarations are as follows:

```
::  ListLS t local context = ListLS [t local context]
::  NilLS    local context = NilLS

instance MenuElements (ListLS m) | MenuElements m
instance MenuElements NilLS
```

Given a list of `MenuElements` instances `ms = [`m_1 `...`m_n`]`, working on the same local state of type `local` and context state `context`, the expression `ListLS ms` is also a `MenuElements` instance working on the same local state and context state. The type constructor `NilLS` is a shorthand for `ListLS []`. It can also be conveniently used to state that a `SubMenu` or `Menu` has no menu elements.

10.2.3 `AddLS` and `NewLS`

The previously discussed glueing type constructors always glue menu elements that work on the same local state and context state. Two other glueing constructors are defined to extend and change the local state, `AddLS` and `NewLS`. Their type constructor definitions and `MenuElements` class instance declarations are as follows:

```
:: AddLS t local context
 = E..add:
   { addLS :: add
   , addDef:: t *(add,local) context
   }
:: NewLS t local context
 = E..new:
   { newLS :: new
   , newDef:: t   new        context
   }

instance MenuElements (AddLS m) | MenuElements m
instance MenuElements (NewLS m) | MenuElements m
```

Given a `MenuElements` instance `m1` that works on a local state of type `local` and a context state of type `context`, one can add another `MenuElements` instance `m2` that works on an *extended* local state of type `(add,local)` and the same context state of type `context`. Let `x` be a value of type `add`, then this is done by the expression `m1 :+: {addLS=x, addDef=m2}`.

Given a `MenuElements` instance `m1` that works on a local state of type `local` and a context state of type `context`, one can add another `MenuElements` instance `m2` that works on a *new* local state of type `new` and the same context state of type `context`. Let `x` be a value of type `new`, then this is done by the expression `m1 :+: {newLS=x, newDef=m2}`.

In both cases the extended part of the local state and the new local state are encapsulated completely from the external context using existential quantification.

10.3 The Windows menu

In this section the special MDI Windows menu should be discussed. This section is not done because currently that menu has not yet been implemented for the Windows(95/NT) platform.

10.4 Menu conventions

The use of menus provides application users with a consistent and uniform access to the available set of commands. In this section we discuss a number of conventions that are usually followed to increase the consistency level.

10.4.1 Subsetting the available commands

In general when using an interactive application, the application will move through several states. In each state a particular subset of the complete set of available commands will be applicable to the user while the remaining commands should not be selected. A well designed application should make this clear to the user by *subsetting* the available commands.

The easiest way to subset commands is by *disabling* and *enabling* the menu elements that should be unselectable and selectable respectively. For this purpose the functions `enableMenuElements` and `disableMenuElements` (module `StdMenuElement`,

Appendix A.16) are available to enable and disable individual menu elements. Complete menus can be enabled and disabled using the `StdMenu` (Appendix A.14) functions `enableMenus` and `disableMenus`. The whole current set of menus can be enabled and disabled using `enableMenuSystem` and `disableMenuSystem`.

10.4.2 Command conventions

In this section we discuss some conventions that are found frequently in many applications with respect to commands.

Clipboard commands

Applications that support the use of the clipboard (Chapter 7) to *cut*, *copy*, and *paste* private and external data usually are found in an `"Edit"` menu. Conventions are:

cut This command should be enabled only if the application is in a state that an object has been selected that can be transfered to the clipboard. Issueing this command should remove that object from its context and place it in the clipboard. Its name should be `"Cut"` and it should have the shortkey attribute `'x'`.

copy This command should be enabled only if the application is in a state that an object has been selected that can be transfered to the clipboard. Issueing this command should place it in the clipboard, but not remove it from its context. Its name should be `"Copy"` and it should have the shortkey attribute `'c'`.

paste This command should be enabled only if the clipboard contains an object that can be currently incorporated in the application. Issueing this command should read the clipboard and put that object in the application. Its name should be `"Paste"` and it should have the shortkey attribute `'v'`.

Undo command

Applications that allow users to manipulate documents by sequences of commands can support an *undo* command. The undo command can also be undone by the *redo* command. These commands are usually found in an `"Edit"` menu. Conventions are:

undo This command should be enabled only if the user has issued a sequence of commands that can be undone. The number of undoable commands depends on the sophistication of the application. The name of this command is `"Undo"` and it has the shortkey attribute `'z'`.

redo This command should be enabled only if a (sequence of) undo command has been issued. At each selection it restores the changes of the undo command. The name of this command is `"Redo"` and it has the shortkey attribute `'y'`.

Document commands

The document commands are frequently found commands to create new documents, open existing documents, and save and close open documents. These commands are usually found in an `"File"` menu. Conventions are:

new This command should be enabled only if the application can modify a new document. Issueing this command should create or reuse a window containing the new document. Its name should be `"New"` and it should have the shortkey attribute `'n'`.

open This command should be enabled only if the application can modify an additional, existing document. Issueing this command should give the user the opportunity to search for a file that will be opened by the application. For this purpose the `StdFileSelect` function `selectInputFile` can be used (Appendix A.7). The name of the command should be `"Open..."` and it should have the shortkey attribute `'o'`.

close This command should be enabled only if the application has an open document window or dialogue. Issueing this command should close the currently active window or dialogue. It is good programming practice to check if the document has been recently saved. If this is not the case, then the user should be asked if the document should be saved before closing. The name of this command should be `"Close"` and it should have the shortkey attribute `'w'`.

save This command should be enabled only if the currently active document window version differs from a (possibly not present) file version. Issueing this command should save the current state of the document in the active window to file. If there is no file associated yet, then the application should first ask for a file name. For this purpose the `StdFileSelect` function `selectOutputFile` can be used (Appendix A.7). The name of the command should be `"Save"` and it should have the shortkey attribute `'s'`.

save as This command should be enabled only if the application has an open document window. Issueing this command should give the user the possibility to browse the file system and provide a file name. For this purpose the `StdFileSelect` function `selectOutputFile` can be used (Appendix A.7). The name of the command should be `"Save As..."`

Quit command

Users can leave an application using the *quit* command. A user should always be allowed to quit the application. It is good programming practice to check if there are any unsaved documents in the application. If this is the case then the user should be asked if these documents should be saved before closing. The name of the quit command is usually `"Quit"` and has the shortkey attribute `'q'`. This command is usually found in a `"File"` menu.

Chapter 11

Timers

Timers provide interactive programs with a tool to let actions occur at regular time intervals. These actions respond to *timer events*, and so they can be properly defined as callback functions. Typical examples of timer uses are blinking cursors, clocks, and time-out mechanisms.

The definition types of timers can be found in module `StdTimerDef`, Appendix A.29. The main type definitions are as follows:

```
::  Timer t ls ps
 =  Timer TimerInterval (t ls ps) [TimerAttribute *(ls,ps)]
::  TimerInterval
:== Int

::  TimerAttribute   ps
 =  TimerId          Id
 |  TimerSelectState SelectState
 |  TimerFunction    (TimerFunction ps)

::  TimerFunction    ps
:== NrOfIntervals->ps->ps
```

A `TimerInterval` is an integer value that must be atleast zero. The time unit is platform dependent and is defined by the `StdSystem` function `ticksPerSecond` (see Appendix A.26).

The `TimerAttribute`s are the following:

`TimerId` This attribute identifies the timer. If you do not provide a `TimerId` the timer can not be modified.

`TimerSelectState` This attribute defines whether the timer will respond to timer events (`Able`) or not (`Unable`). The default value is `Able`.

`TimerFunction` This attribute is the callback function that is evaluated when a timer event is handled. Its first argument is the number of whole timer intervals that have elapsed since its previous evaluation, so this value is atleast 1. If the timer interval is zero, then this number is always 1. Enabling a timer from a disabled state resets the last evaluation time.

The `Timer` type constructor is parameterised with a type constructor variable. Analogous to menus that contain menu elements, timers can contain *timer elements*.

The instances must be member of the `TimerElements` class. This is expressed by the timer *creation* function, `openTimer` which can be found in module `StdTimer` (Appendix A.28):

```
class Timers tdef where
  openTimer :: .ls !(tdef .ls (PSt .l .p)) !(PSt .l .p)
             -> (!ErrorReport,!PSt .l .p)

instance Timers (Timer t) | TimerElements t
```

Currently, the instances of the `TimerElements` class are *receivers* and the usual *glueing* type constructors that we have already encountered in controls (Section 9.2) and menu elements (Section 10.2), namely `AddLS`, `NewLS`, `ListLS`, `NilLS`, and `:+:`. The receiver instances are declared in module `StdTimerReceiver` (Appendix A.31). Receivers are not handled in this chapter but in Chapter 12. The glueing constructors are declared in the same module `StdTimerElement` that contains the `TimerElements` class (Appendix A.30).

11.1 Examples

In this section we give some examples to illustrate the use of timers.

11.1.1 Expanding circles

In this example we create a program that uses a timer to draw a number of growing concentric circles in a fixed size window periodically. It also opens a menu containing only the quit command. Figure 11.1 shows the program in action. We discuss these object I/O components in reverse order (menu, window, timer).

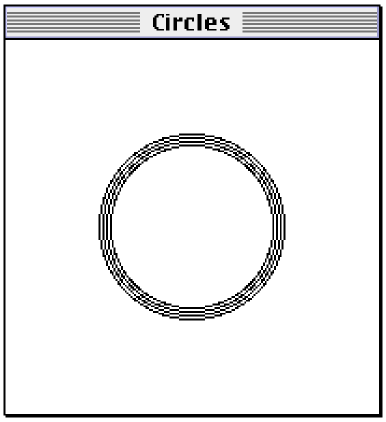


Figure 11.1: The circles program in action.

The menu definition is very simple, as it contains only one command to terminate the example program. Recall that to terminate an interactive process the `StdProcess` function `closeProcess` must be used. The `StdIOCommon` function `noLS` suitably turns `closeProcess` into the desired type. Here is the menu definition.

```
mdef = Menu "Circles"
```

```
         (MenuItem "Quit" [MenuFunction (noLS closeProcess)
                          ,MenuShortKey 'q'
                          ]
         ) []
```

For simplicity, the window in which the circles are to be drawn has a fixed size, namely `windowEdge` by `windowEdge` (200 in the example). To ease drawing, we take care that the view domain of the window has the origin `zero` right in the center of the view domain. This can be done conveniently by defining the `WindowViewDomain` attribute to be:

```
windowViewDomain
 = { corner1 = {x= ~windowEdge/2,y= ~windowEdge/2}
   , corner2 = {x=  windowEdge/2,y=  windowEdge/2}
   }
```

The window does not contain controls (expressed by the `Controls` type instance `NilLS`) and is identified by the value `windowid`. The window definition is as follows:

```
wdef = Window "Circles" NilLS
         [WindowId         windowid
         ,WindowSize       (rectangleSize windowViewDomain)
         ,WindowViewDomain windowViewDomain
         ]
```

The timer draws a number of concentric circles that have an increasing radius. For this purpose it uses a local state of the following type and initial value:

```
:: TimerState
 = { nrCircles    :: Int
   , equiDistance :: Int
   , minRadius    :: Int
   }
initTimerState
 = { nrCircles    = 4
   , equiDistance = 2
   , minRadius    = 0
   }
```

The `nrCircles` field contains the number of circles that are drawn. The `equiDistance` field is the difference of radius between two neighbouring circles. The `minRadius` field keeps track of the radius of the smallest visible circle.

The timer interval is set to a twentieth of a second (`ticksPerSecond/20`). The timer contains no timer elements. Its definition is as follows:

```
tdef = Timer (ticksPerSecond/20) NilLS [TimerFunction timer]
```

The timer function `timer` will be evaluated by the object I/O system every twentieth of a second (if possible). The timer function actually ignores the number of elapsed intervals and simply draws the next sequence of circles. There are two cases to distinguish:

If the smallest circle still fits entirely inside the window (tested by `minRadius < windowEdge/2`), then `timer` erases the smallest circle by drawing it in white. It then draws the new circle in black which should have a radius equal to `minRadius + nrCircles * equiDistance`. Finally, the `minRadius` field is changed to reflect the fact that the smallest visible circle now has radius `minRadius + equiDistance`.

If the smallest circle does not fit entirely inside the window, then `timer` completely erases the window by filling the document layer picture with a white rectangle. By setting the new local `TimerState` back to `initTimerState` the circles are drawn again from the center.

```
timer :: NrOfIntervals (TimerState,PSt .l .p)
                     -> (TimerState,PSt .l .p)
timer _ (ls=:{nrCircles,equiDistance,minRadius},ps)
  | minRadius<windowEdge/2
    # ls        = {ls & minRadius=minRadius+equiDistance}
      newRadius = minRadius+nrCircles*equiDistance
    # ps        = appPIO (drawInWindow windowid
                         [setPenColour White
                         ,draw         {oval_rx=minRadius
                                       ,oval_ry=minRadius
                                       }
                         ,setPenColour Black
                         ,draw         {oval_rx=newRadius
                                       ,oval_ry=newRadius
                                       }
                         ]) ps
    = (ls,ps)
  | otherwise
    # ps        = appPIO (drawInWindow windowid
                         [setPenColour White
                         ,fill         windowViewDomain
                         ]) ps
    = (initTimerState,ps)
```

The last details that remain to be defined are the actual opening of the menu, window, and timer, the opening of the interactive process, and the creation of the `windowid`. For completeness we show the complete program code.

```
module circles

//  ********************************************************************************
//  Clean tutorial example program.
//
//  This program creates a window that displays growing concentric circles.
//  For this purpose it uses a timer.
//  ********************************************************************************

import StdEnv, StdIO

::  NoState = NoState

::  TimerState
    =   {   nrCircles       :: Int
        ,   equiDistance    :: Int
        ,   minRadius       :: Int
        }

Start :: *World -> *World
Start world
    #   (windowid,world)    = openId world
```

```
    =   startIO NoState NoState [initialise windowid] [] world

initialise windowid ps
    #   (error,ps)      = openMenu NoState mdef ps
    |   error<>NoError
        =   closeProcess ps
    #   (error,ps)      = openWindow NoState wdef ps
    |   error<>NoError
        =   closeProcess ps
    #   (error,ps)      = openTimer initTimerState tdef ps
    |   error<>NoError
        =   closeProcess ps
    |   otherwise
        =   ps
where
    mdef            = Menu "Circles"
                        (   MenuItem "Quit"       [MenuFunction (noLS closeProcess)
                                                  ,MenuShortKey 'q'
                                                  ]
                        )   []
    wdef            = Window "Circles" NilLS
                             [   WindowId         windowid
                             ,   WindowSize       (rectangleSize windowViewDomain)
                             ,   WindowViewDomain windowViewDomain
                             ]
    tdef            = Timer (ticksPerSecond/20) NilLS
                             [   TimerFunction    timer
                             ]
    windowEdge      = 200
    windowViewDomain = { corner1={x= ~windowEdge/2,y= ~windowEdge/2}
                       , corner2={x=  windowEdge/2,y=  windowEdge/2}
                       }
    initTimerState  = { nrCircles   = 4
                      , equiDistance= 2
                      , minRadius   = 0
                      }
    timer _ (ls=:{nrCircles,equiDistance,minRadius},ps)
        |   minRadius<windowEdge/2
            #   ls          = {ls & minRadius=minRadius+equiDistance}
                newRadius   = minRadius+nrCircles*equiDistance
            #   ps          = appPIO
                                (drawInWindow windowid
                                [setPenColour White
                                ,draw         {oval_rx=minRadius,oval_ry=minRadius}
                                ,setPenColour Black
                                ,draw         {oval_rx=newRadius,oval_ry=newRadius}
                                ]
                                ) ps
            =   (ls,ps)
        |   otherwise
            #   ps          = appPIO
                                (drawInWindow windowid
                                [setPenColour White,fill windowViewDomain]
                                ) ps
            =   (initTimerState,ps)
```

11.1.2 Internal clock

In this example we create a program that uses three timers to track the elapsed time since startup. The timers track the elapsed seconds, minutes, and hours respectively. A dialogue is used to provide visual feedback. Figure 11.2 shows the application in action. We first have a look at the dialogue, and then the three timers.

The dialogue `ddef` simply uses `TextControls` to display the hours, minutes, and seconds. These controls are identified by the `Id` values `hoursId`, `minutesId`, and

Figure 11.2: The timing program in action.

`secondsId` respectively. The dialogue itself is identified by the value `dialogId`. `CompoundControl`s are used to get the layout done in the desired way. Observe the use of list comprehensions and the `ListLS` `Controls` class instance. In this example closing the dialogue also terminates the application. Here is the dialogue definition:

```
ddef
= Dialog "Stopwatch"
  (   CompoundControl
  (   ListLS [TextControl text [ControlPos (Left,zero)]
             \\ text<-["Hours:","Minutes:","Seconds:"]
             ]
  )   []
  :+: CompoundControl
  (   ListLS [TextControl "00" [ControlPos (Left,zero)
                               ,ControlId  id
                               ]
             \\ id<-[hoursId,minutesId,secondsId]
             ]
  )   []
  )
  [   WindowClose (noLS closeProcess)
  ,   WindowId    dialogId
  ]
```

Because the operation of each of the three timers is very similar, we use one function `tdef` to define them and parameterise it with their respective `TimerInterval`s. The timers do not contain any timer elements. Each timer has a local integer state (initially zero) that keeps the current number of evaluated time units modulo their maximum number. This number is derived from the `TimerInterval` using the straightforward function `maxunit`. The timerfunction `tick` now relies on the `NrOfIntervals` parameter. Given the current number of evaluated time units in its local integer state, each timer function adds the `NrOfIntervals` value to it. Depending on the `TimerInterval` value, the proper modulo value is taken (using `maxunit`). This new value is the new local state and is also drawn in the dialogue, using a local convenience function `setText` and `textid` to determine the `Id` of the corresponding text control.

```
tdef timerInterval
  = Timer timerInterval NilLS [TimerFunction tick]
where
```

```
  tick nrElapsed (time,ps)
    # time = (time+nrElapsed) mod (maxunit timerInterval)
    = (time,setText (textid timerInterval) (toString time) ps)

  setText id text ps
    = appPIO (setWindow dialogId [setControlTexts [(id,text)]]) ps

maxunit interval
  | interval==second = 60
  | interval==minute = 60
  | interval==hour   = 24
textid interval
  | interval==second = secondsId
  | interval==minute = minutesId
  | interval==hour   = hoursId
```

The last details that remain to be defined are the actual opening of the three timers, the dialogue, the opening of the interactive process, and the creation of the proper `Ids`. For completeness we show the complete program code.

```
module stopwatch

//  ****************************************************************************
//  Clean tutorial example program.
//
//  This program creates a window that tracks the elapsed time since startup.
//  For this purpose it uses three timers to track the seconds, minutes, and hours
//  separately.
//  ****************************************************************************

import StdEnv,StdIO

::  NoState
    =   NoState
::  DialogInfo
    =   {   secondsId   :: Id
        ,   minutesId   :: Id
        ,   hoursId     :: Id
        ,   dialogId    :: Id
        }

second  :== ticksPerSecond
minute  :== 60*second
hour    :== 60*minute

openDialogInfo :: *env -> (DialogInfo,*env) | Ids env
openDialogInfo env
    #   ([secondsid,minutesid,hoursid,dialogid:_],env) = openIds 4 env
    =   (   {   secondsId=secondsid
            ,   minutesId=minutesid
            ,   hoursId  =hoursid
            ,   dialogId =dialogid
            }
        ,   env
        )

Start :: *World -> *World
Start world
    =   startIO NoState NoState [initialise`] [] world
where
    initialise` ps
        #   (dialogInfo,ps) = accPIO openDialogInfo ps
        =   initialise dialogInfo ps
```

```
initialise :: DialogInfo (PSt .l .p) -> (PSt .l .p)
initialise {secondsId,minutesId,hoursId,dialogId} ps
    #   (errors,ps)     = seqList [  openTimer 0 (tdef timerinfo)
                                  \\ timerinfo<-[second,minute,hour]
                                  ]  ps
    |   any ((<>) NoError) errors
        =   closeProcess ps
    #   (error,ps)      = openDialog NoState ddef ps
    |   error<>NoError
        =   closeProcess ps
    |   otherwise
        =   ps
where
    tdef timerInterval
        =   Timer timerInterval NilLS [TimerFunction tick]
    where
        tick nrElapsed (time,ps)
            #   time    = (time+nrElapsed) mod (maxunit timerInterval)
            =   (time,setText (textid timerInterval) (toString time) ps)

        setText id text ps
            =   appPIO (setWindow dialogId [setControlTexts [(id,text)]]) ps

        textid interval
            |   interval==second   = secondsId
            |   interval==minute   = minutesId
            |   interval==hour     = hoursId
        maxunit interval
            |   interval==second   = 60
            |   interval==minute   = 60
            |   interval==hour     = 24

    ddef=   Dialog "Stopwatch"
                (   CompoundControl
                (   ListLS  [  TextControl text [ControlPos (Left,zero)]
                            \\ text<-["Hours:","Minutes:","Seconds:"]
                            ]
                )   []
                :+: CompoundControl
                (   ListLS  [  TextControl "00" [ControlPos (Left,zero)
                                                ,ControlId  id]
                            \\ id<-[hoursId,minutesId,secondsId]
                            ]
                )   []
                )
                [   WindowClose (noLS closeProcess)
                ,   WindowId    dialogId
                ]
```

Chapter 12

Receivers

All the interactive object I/O components discussed so far have in common that the events to which they respond are *abstract*. In this context abstract means that it is not specified in detail what *concrete* events cause a specific callback function to be evaluated. For instance, a callback function associated with a `MenuItem` (Section 10.1.3) is evaluated when it has been selected by the user. How this selection takes place is not specified.

In this section we discuss an interactive object I/O component that responds to program defined events, or rather *messages*. This component is the *receiver*. It plays an important role in the construction of interactive components. There are *no restrictions* on the kind of messages that can be *sent* or *received*, provided that they are type correct. The latter is obtained by using special identification values for receivers, the *receiver ids* (Chapter 4).

We will first have a look at the definition of receivers in Section 12.1. Receivers can be opened as top level object I/O components, but also as elements of windows, dialogues, and menus. This is discussed in Section 12.2. Knowing how to define and open receivers, we show which functions are available to send messages in Section 12.3. Section 12.4 contains a number of examples to demonstrate the use of receivers.

12.1 Receiver definitions

There are two kinds of receivers. *Uni-directional* receivers respond only to messages. *Bi-directional* receivers respond to messages and also reply with a message. The types needed to define receivers can be found in the module `StdReceiverDef`, Appendix A.25.

A uni-directional receiver that responds to messages of type `msg` is defined by:

```
::  Receiver  msg   ls ps
 =  Receiver  (RId  msg)      (ReceiverFunction  msg      *(ls,ps))
                              [ReceiverAttribute          *(ls,ps)]
::  ReceiverFunction msg ps
:== msg -> ps -> ps
```

A bi-directional receiver that responds to messages of type `msg` and returns a response message of type `resp` is defined by:

```
::  Receiver2 msg resp ls ps
 =  Receiver2 (R2Id msg resp) (Receiver2Function msg resp *(ls,ps))
                              [ReceiverAttribute         *(ls,ps)]
::  Receiver2Function msg resp ps
:== msg -> ps -> (resp,ps)
```

The set of receiver attributes is currently limited to the `ReceiverSelectState` which default value is `Able`.

```
::  ReceiverAttribute ps
 =  ReceiverSelectState SelectState
```

12.2 Receiver creation

Receivers can be opened as top level interface elements such as windows, dialogues, menus, and timers. This is done in the usual, overloaded way (module `StdReceiver`, Appendix A.24):

```
class Receivers rdef where
  openReceiver   :: .ls !(rdef .ls (PSt .l .p)) !(PSt .l .p)
                               -> (!ErrorReport,!PSt .l .p)
  ...

instance Receivers (Receiver  msg)
instance Receivers (Receiver2 msg resp)
```

Receivers can also be opened as elements of windows, dialogues, menus, and timers. So, in a window or dialogue one can not only add the set of controls as discussed in Chapter 9 but also receivers. This is accomplished by declaring receivers to be instances of the `Controls` type constructor class in module `StdControlReceiver` (Appendix A.6). Analogously, receivers can be *menu elements* (module `StdMenu-Receiver`, Appendix A.18), and *timer elements* (module `StdTimerReceiver`, Appendix A.31).

In all cases, when opening receivers, their `R(2)Id` values must be unique (Chapter 4). The reason is that the message passing functions that we are about to discuss next require the `R(2)Id` value.

12.3 Message passing

In contrast with the previously discussed object I/O components, receivers *must* have an identification value. The *message passing* functions require this identification value to ensure that a message of the correct type is sent. The message passing functions can be found in module `StdReceiver`, Appendix A.24.

All message passing functions return a report about the message passing action. This report is an algebraic type `SendReport` has the following alternatives:

```
::  SendReport
 =  SendOk
 |  SendUnknownProcess
 |  SendUnknownReceiver
```

```
 |  SendUnableReceiver
 |  SendDeadlock
```

For all functions, the alternative value `SendOk` is returned in case message passing was successful. The alternative `SendUnknownReceiver` is returned in case the indicated receiver is not open at the moment of sending the message. The other `SendReport` alternatives are discussed below.

We start with message passing to uni-directional receivers in Section 12.3.1. Bi-directional message passing is discussed in Section 12.3.2.

12.3.1 Uni-directional message passing

There are two functions a programmer can use to send a message to a uni-directional receiver: `asyncSend` and `syncSend`. which have the same function types:

```
asyncSend :: !(RId msg) msg !(PSt .l .p) -> (!SendReport,!PSt .l .p)
syncSend  :: !(RId msg) msg !(PSt .l .p) -> (!SendReport,!PSt .l .p)
```

Using `asyncSend`, a message is placed at the end of the asynchronous message queue of the indicated receiver and will, at some point, be handled by that receiver. It is unspecified *when* that event occurs. Because events are handled one by one, the programmer can be certain that this will not occur within evaluation of the callback function that applies `asyncSend`. Asynchronous messages that are sent in sequence will be evaluated in that sequence. Asynchronous message passing only fails in case the indicated receiver is not open, as discussed above. One should observe that successfully queueing an asynchronous message does not guarantee that the message will be handled. The receiver might be disabled or closed before all of its messages have been handled.

If one wants to enforce a receiver to handle a message, one should use `syncSend`. This function does not place the message in the message queue of the indicated receiver, but evaluates its receiver function given the message. This implies that `syncSend` has to *switch context* because the indicated receiver may be part of another object I/O component. Another consequence is that synchronous messages may overtake asynchronous messages. However, synchronous messages that are sent in sequence will be evaluated in that sequence.

Sending a synchronous message fails in case the indicated receiver does not exist. If the indicated receiver does exist, but its `ReceiverSelectState` attribute is `Unable`, then the message is also not handled. In this case, the `SendReport` alternative `SendUnableReceiver` is returned.

Examples of uni-directional message passing are given in Section 12.4. The first example, in Section 12.4.1, demonstrates the use of asynchronous message passing, while the second example, in Section 12.4.2, demonstrates synchronous message passing.

12.3.2 Bi-directional message passing

Bi-directional message passing is *synchronous*. A message is sent using the function `syncSend2`:

```
syncSend2 :: !(R2Id msg resp) msg !(PSt .l .p)
   -> (!(!SendReport,!Maybe resp), !PSt .l .p)
```

Analogous to `syncSend`, `syncSend2` locates the indicated bi-directional receiver and applies the argument message immediately to the corresponding receiver function. If this receiver could be found and happens to be `Able`, evaluation of the receiver function will yield a response message `resp`. In this case the `SendReport` result of `syncSend2` is `SendOk`, and the response value is returned as `(Just resp)`. In all exceptional cases, there is no response value and `Nothing` is returned. To evaluate the receiver function, `syncSend2` has to switch context as well.

Bi-directional receivers can be used to retrieve local encapsulated data. Example 12.4.3 demonstrates this.

12.4 Examples

In this section we give some examples to illustrate the use of receivers.

12.4.1 Talk windows

In this example we create a program that uses receivers to send keyboard input from one window to another window, and vice versa. This results in a talk like application (although it is not very useful as a talk application because it runs on one computer). Figure 12.1 shows the application in action.

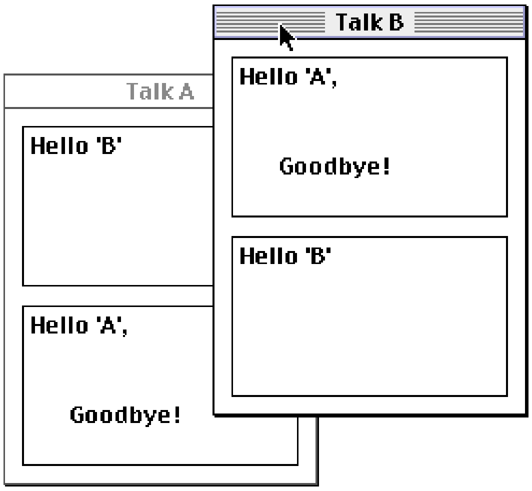


Figure 12.1: The talk program in action.

The initialisation actions of the talk program, defined by the function `initialise`, first create a menu that has only one quit menu item. Then two `RId` values are generated that will be used to identify the two receivers. The function `openTalkWindow` is then applied twice to create the two windows. Its parameters are the name of the window and the two receiver ids. The first receiver id parameter identifies the private receiver, while the second identifies the other receiver.

```
initialise :: (PSt .l .p) -> PSt .l .p
initialise ps
  # menu       = Menu "Talk"
                  (MenuItem "Quit" [MenuShortKey'q'
                                   ,MenuFunction (noLS closeProcess)
```

```
                                ]
                 ) []
  # (error,ps) = openMenu NoState menu ps
  | error<>NoError
    = abort "talk could not open menu."
  # (a,ps)     = accPIO openRId ps
  # (b,ps)     = accPIO openRId ps
  # ps         = openTalkWindow "A" a b ps
  # ps         = openTalkWindow "B" b a ps
  | otherwise
    = ps
```

`openTalkWindow` creates a dialogue in which the user can type text and see the messages of the other talk window. The dialogue consists of three components: in the first `EditControl`, identified by `inId`, the user can type text. The `ControlKeyboard` attribute takes care that the program can respond to keyboard input. The second `EditControl`, identified by `outId`, is used to present the messages coming from the other talk window. To prevent the user from typing text in this control its initial `ControlSelectState` attribute is `Unable`. The `Receiver` component is the one to which the messages are being sent.

```
openTalkWindow :: String (RId String) (RId String) (PSt .l .p)
                                                 -> PSt .l .p
openTalkWindow name me you ps
  # (wId,  ps) = accPIO openId ps
  # (inId, ps) = accPIO openId ps
  # (outId,ps) = accPIO openId ps
  # talk       = Dialog ("Talk "+++name)
                 (   EditControl "" (hmm 50.0) 5
                     [ ControlId       inId
                     , ControlKeyboard inputfilter Able
                                       (noLS1 (input wId inId you))
                     ]
                 :+: EditControl "" (hmm 50.0) 5
                     [ ControlId       outId
                     , ControlPos      (BelowPrev,zero)
                     , ControlSelectState Unable
                     ]
                 :+: Receiver me (noLS1 (receive wId outId)) []
                 )
                 [   WindowId    wId
                 ]
  # (error,ps)  = openDialog NoState talk ps
  | error<>NoError
    = abort "talk could not open window."
  | otherwise
    = ps
```

The input `EditControl` has a keyboard filter, `inputfilter`, that accepts only `KeyDown` keyboard input.

```
inputfilter :: KeyboardState -> Bool
inputfilter keystate = getKeyboardStateKeyState keystate<>KeyUp
```

For accepted keyboard input value of type `KeyboardState`, the callback function `input` is evaluated. It first gets the current state of the dialogue, using the `StdControl` library function `getWindow` (Appendix A.3). From this value the current content of the input `EditControl` can be retrieved, using the function `getControlTexts`. This new content, which is a `String`, is being sent asynchronously to the other receiver. Note that `input` assumes that both `getWindow` and `asyncSend` never fail.

```
input :: Id Id (RId String) KeyboardState (PSt .l .p) -> PSt .l .p
input wId inId you _ ps
  # (Just window,ps)
         = accPIO (getWindow wId) ps
    text = fromJust (snd (hd (getControlTexts [inId] window)))
  = snd (asyncSend you text ps)
```

For every string message received from the other talk window, the receiver function `receive` is evaluated. It simply replaces the current content of the output `EditControl` with the new text. This is done using the function `setControlTexts`. The function `setEditControlCursor` makes sure that the end of the text is visible.

```
receive :: Id Id String (PSt .l .p) -> PSt .l .p
receive wId outId text ps
  = appPIO (setWindow wId [setControlTexts      [(outId,text)]
                          ,setEditControlCursor outId (size text)
                          ]) ps
```

For completeness the whole program is shown here.

```
module talk

//  ********************************************************************************
//  Clean tutorial example program.
//
//  This program creates two windows that communicate with each other using message
//  passing. Text that has been typed in one window is being sent to the other, and
//  vice versa.
//  ********************************************************************************

import  StdEnv, StdIO

::  NoState
    =   NoState

Start :: *World -> *World
Start world
    =   startIO NoState NoState [initialise] [] world
where
    initialise :: (PSt .l .p) -> PSt .l .p
    initialise ps
        #   menu        = Menu "Talk"
                            (   MenuItem "Quit" [   MenuShortKey 'q'
                                                ,   MenuFunction (noLS closeProcess)
                                                ]
                            )   []
        #   (error,ps)  = openMenu undef menu ps
        |   error<>NoError
            =   abort "talk could not open menu."
        #   (a,ps)      = accPIO openRId ps
        #   (b,ps)      = accPIO openRId ps
```

```
        #   ps          = openTalkWindow "A" a b ps
        #   ps          = openTalkWindow "B" b a ps
        |   otherwise
            =   ps

openTalkWindow :: String (RId String) (RId String) (PSt .l .p) -> PSt .l .p
openTalkWindow name me you ps
    #   (wId,  ps)  = accPIO openId ps
    #   (inId, ps)  = accPIO openId ps
    #   (outId,ps)  = accPIO openId ps
    #   wdef        = Dialog ("Talk "+++name)
                        (   EditControl "" (hmm 50.0) 5
                                    [   ControlId       inId
                                    ,   ControlKeyboard inputfilter Able
                                                        (noLS1 (input wId inId you))
                                    ]
                        :+: EditControl "" (hmm 50.0) 5
                                    [   ControlId       outId
                                    ,   ControlPos      (BelowPrev,zero)
                                    ,   ControlSelectState Unable
                                    ]
                        )
                        [   WindowId    wId
                        ]
    #   (error,ps)  = openDialog undef wdef ps
    |   error<>NoError
        =   abort "talk could not open window."
    #   rdef        = Receiver me (noLS1 (receive wId outId)) []
    #   (error,ps)  = openReceiver NoState rdef ps
    |   error<>NoError
        =   abort "talk could not open receiver"
    |   otherwise
        =   ps
where
    inputfilter :: KeyboardState -> Bool
    inputfilter keystate
        =   getKeyboardStateKeyState keystate<>KeyUp

    input :: Id Id (RId String) KeyboardState (PSt .l .p) -> PSt .l .p
    input wId inId you _ ps
        #   (Just window,ps) = accPIO (getWindow wId) ps
            text             = fromJust (snd (hd (getControlTexts [inId] window)))
        =   snd (asyncSend you text ps)

    receive :: Id Id String (PSt .l .p) -> PSt .l .p
    receive wId outId text ps
        =   appPIO (setWindow wId [ setControlTexts      [(outId,text)]
                                  , setEditControlCursor outId (size text)
                                  ])  ps
```

12.4.2 Resetting the counter

In this example we extend the example counter in Section 9.2.4 (page 74) with a means to reset the counter to zero. We proceed in a bottom-up style: the counter control is a compound control extended with a receiver that, when it receives a message, will reset the counter to zero. This control encapsulates its local counter state. Then a button control is defined that, when selected, sends a message to the receiver component of the counter control. The whole is being placed in a dialogue. Figure 12.2 gives a snapshot of the program.

The main component is ofcourse the counter control, defined by `counter`. It encapsulates an integer local state with initial value `initcount`. Its definition is almost identical to the one shown on page 74 except that a `Receiver` has been added.

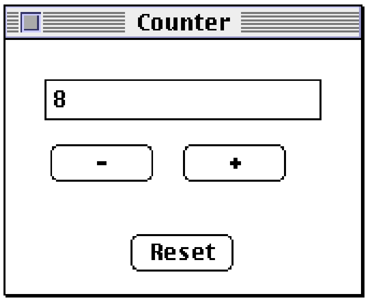


Figure 12.2: The counter control with reset button.

```
counter
  = {newLS = initcount
    ,newDef= CompoundControl
             (   EditControl (toString initcount) (hmm 50.0) 1
                                   [ControlSelectState Unable
                                   ,ControlPos         (Center,zero)
                                   ,ControlId          displayid
                                   ]
             :+: ButtonControl "-" [ControlFunction    (count (-1))
                                   ,ControlPos         (Center,zero)
                                   ]
             :+: ButtonControl "+" [ControlFunction    (count   1 )]
             :+: Receiver resetid reset []
             )   []
    }
```

The only purpose of the receiver is to reset the current local counter value to `initcount` and show this by changing the content of the `EditControl`. Note that the receiver function `reset` is not interested at all in the message.

```
reset :: m (Int,PSt .l .p) -> (Int,PSt .l .p)
reset _ (_,ps)
  = (initcount,setText windowid displayid initcount ps)
```

The reset button is a straightforward `ButtonControl`. It is centered below the counter control. Its `ControlFunction` is just to send a message synchronously to the receiver component of the counter control. Because this component does not care about the message, the button function can be as bold to send the `StdMisc` library function `undef`. This function, when evaluated, aborts the application. This demonstrates that message passing is truely lazy in the message argument.

```
resetbutton
  = ButtonControl "Reset"
       [ControlFunction (noLS (snd o syncSend resetid undef))
       ,ControlPos      (Center,zero)
       ]
```

The final details of the program are to generate the proper identification values and to create the initial process and dialogue. For completeness, the program code is given here.

```
module counterreset

// ****************************************************************************
// Clean tutorial example program.
//
// This program defines a Controls component that implements a manually settable
// counter. A receiver is used to add a reset option.
// ****************************************************************************

import StdEnv, StdIO

:: NoState
    = NoState

Start :: *World -> *World
Start world
    = startIO NoState NoState [initialise] [] world
where
    initialise ps
        # (windowid, ps)  = accPIO openId  ps
        # (displayid,ps)  = accPIO openId  ps
        # (resetid,  ps)  = accPIO openRId ps
        # (error,ps)      = openDialog NoState
                               (dialog windowid displayid resetid) ps
        | error<>NoError
            = abort "counter could not open Dialog."
        | otherwise
            = ps

dialog windowid displayid resetid
    = Dialog "Counter"
            ( counter
            :+: resetbutton
            )
            [ WindowId    windowid
            , WindowClose (noLS closeProcess)
            ]
where
    counter
        = { newLS  = initcount
          , newDef = CompoundControl
                     ( EditControl (toString initcount) (hmm 50.0) 1
                                       [ControlSelectState  Unable
                                       ,ControlPos          (Center,zero)
                                       ,ControlId           displayid
                                       ]
                     :+: ButtonControl "-" [ControlFunction      (count (-1))
                                       ,ControlPos          (Center,zero)
                                       ]
                     :+: ButtonControl "+" [ControlFunction      (count   1 )]
                     :+: Receiver resetid reset []
                     ) []
          }
    where
        initcount   = 0

        count :: Int (Int,PSt .l .p) -> (Int,PSt .l .p)
        count dx (count,ps)
            = (count+dx,setText windowid displayid (count+dx) ps)

        reset :: m (Int,PSt .l .p) -> (Int,PSt .l .p)
        reset _ (_,ps)
            = (initcount,setText windowid displayid initcount ps)

        setText :: Id Id x (PSt .l .p) -> PSt .l .p | toString x
        setText wid cid x ps
            = appPIO (setWindow wid [setControlTexts [(cid,toString x)]]) ps
```

```
resetbutton
    =   ButtonControl "Reset"
            [ControlFunction (noLS (snd o syncSend resetid undef))
            ,ControlPos      (Center,zero)
            ]
```

12.4.3 Reading the counter

In this example we extend the counter example once more and add a dialogue that reads the local counter value. To be able to do this a bi-directional receiver is added to the counter. Figure 12.3 gives a snapshot of the program.

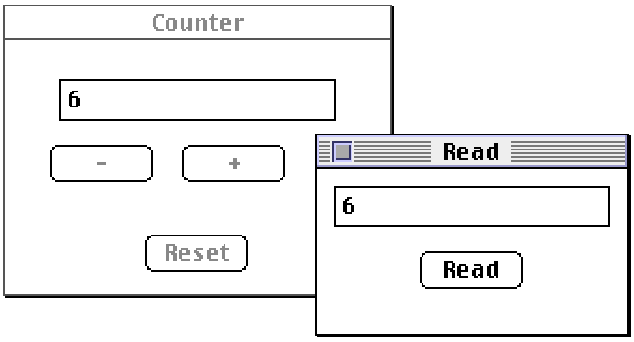


Figure 12.3: Reading the counter control with reset button.

The first change is to extend the counter component with a bi-directional receiver component defined by the expression `(Receiver2 readid read [])`, where `readid` is a `R2Id` identification value. The receiver function `read` is very straightforward: it returns the current local counter value without changing anything (also in this case `read` is not interested in the input message type):

```
read :: m (Int,PSt .l .p) -> (Int,(Int,PSt .l .p))
read _ (count,ps)
    =  (count,(count,ps))
```

The initialisation actions open another dialogue defined by `display` that obtains and shows the counter value on request. This request can be done by a user by pressing the `ButtonControl` labeled `"Read"`.

```
display windowid displayid readid
  = Dialog "Read"
    (   EditControl   "" (hmm 50.0) 1
                         [ControlSelectState Unable
                         ,ControlId          displayid
                         ]
    :+: ButtonControl "Read" [ControlFunction    (noLS read)
                         ,ControlPos         (Center,zero)
                         ]
    )
    [   WindowId    windowid
    ,   WindowClose (noLS closeProcess)
    ]
```

When the "Read" button has been selected, its callback function `read` is evaluated. It sends a message to the bi-directional receiver component of the counter control using `syncSend2`. If this action fails it aborts the program. Although this situation will never occur, this check has been added for program hygiene. In a successfull communication, `value` will be `(Just count)`, and it is this value that is then displayed in the `EditControl` of the dialogue.

```
read ps
  # ((error,value),ps)  = syncSend2 readid undef ps
  | case error of SendOk -> False; _ -> True
    = abort "could not read counter value"
  | otherwise
    = setText windowid displayid (fromJust value) ps
```

The initialisation actions create the necessary identification values. Because of the `Id` assignment rules (Chapter 4) the `Id displayid` can be used in both dialogues. Here is the complete program code.

```
module counterread

//  ******************************************************************************
//  Clean tutorial example program.
//
//  This program defines a Controls component that implements a manually settable
//  counter.
//  A bi-directional receiver is added to give external access to the counter value.
//  ******************************************************************************

import StdEnv, StdIO

::  NoState
    =   NoState

Start :: *World -> *World
Start world
    =   startIO NoState NoState [initialise] [] world

initialise ps
    #   (windowid, ps)  = accPIO openId   ps
    #   (displayid,ps)  = accPIO openId   ps
    #   (resetid,  ps)  = accPIO openRId  ps
    #   (readid,  ps)   = accPIO openR2Id ps
    #   (error,ps)      = openDialog NoState
                            (dialog windowid displayid resetid readid) ps
    |   error<>NoError
        =   abort "counter could not open counter dialog."
    #   (windowid,ps)   = accPIO openId   ps
    #   (error,ps)      = openDialog NoState (display windowid displayid readid) ps
    |   error<>NoError
        =   abort "counter could not open display dialog."
    |   otherwise
        =   ps
where
    dialog windowid displayid resetid readid
        =   Dialog "Counter"
                (   counter
                :+: resetbutton
                )
                [   WindowId    windowid
                ,   WindowClose (noLS closeProcess)
                ]
    where
        counter
```

```
        =   {   newLS   = initcount
            ,   newDef  = CompoundControl
                          (   EditControl (toString initcount) (hmm 50.0) 1
                                                [ControlSelectState Unable
                                                ,ControlPos      (Center,zero)
                                                ,ControlId       displayid
                                                ]
                          :+: ButtonControl "-" [ControlFunction (count (-1))
                                                ,ControlPos      (Center,zero)
                                                ]
                          :+: ButtonControl "+" [ControlFunction (count   1 )]
                          :+: Receiver  resetid reset []
                          :+: Receiver2 readid  read  []
                          )   []
            }
    where
        initcount = 0

        count :: Int (Int,PSt .l .p) -> (Int,PSt .l .p)
        count dx (count,ps)
            #   count   = count+dx
            =   (count,setText windowid displayid count ps)

        reset :: m (Int,PSt .l .p) -> (Int,PSt .l .p)
        reset _ (_,ps)
            =   (initcount,setText windowid displayid initcount ps)

        read :: m (Int,PSt .l .p) -> (Int,(Int,PSt .l .p))
        read _ (count,ps)
            =   (count,(count,ps))

    resetbutton
        =   ButtonControl "Reset" [ControlFunction (noLS reset)
                                  ,ControlPos      (Center,zero)
                                  ]
    where
        reset ps
            =   snd (syncSend resetid undef ps)

display windowid displayid readid
    =   Dialog "Read"
            (   EditControl   "" (hmm 50.0) 1
                                  [ControlSelectState    Unable
                                  ,ControlId             displayid
                                  ]
            :+: ButtonControl "Read" [ControlFunction       (noLS read)
                                  ,ControlPos            (Center,zero)
                                  ]
            )
            [   WindowId    windowid
            ,   WindowClose (noLS closeProcess)
            ]
where
    read ps
        #   ((error,value),ps)  = syncSend2 readid undef ps
        |   case error of SendOk -> False; _ -> True
            =   abort "could not read counter value"
        |   otherwise
            =   setText windowid displayid (fromJust value) ps

setText :: Id Id x (PSt .l .p) -> PSt .l .p | toString x
setText wid cid x ps
    =   appPIO (setWindow wid [setControlTexts [(cid,toString x)]]) ps
```

Chapter 13

Interactive processes

All of the examples discuss so far created an interactive program using the `StdProcess` function `startIO` (Appendix A.21). This function creates an interactive process that engages in some graphical user interface actions with a user and then terminates, using the `StdProcess` function `closeProcess`. In this chapter we show how an interactive process can spawn new interactive processes that will run *interleaved*. Instead of using only `startIO` and these interactive process creation functions, it is also possible to create a whole *process topology* at once.

We start the discussion by looking at the ways to define interactive processes in Section 13.1. This is followed by the functions to open interactive processes. Finally we give some examples to illustrate their application.

13.1 Defining interactive processes

The types to define interactive processes can be found in the module `StdProcessDef` (Appendix A.22). In the object I/O library, interactive processes are distinguished by their *document interface*. There are three kinds of document interfaces, and there are also three corresponding type constructors to define an individual interactive process:

No Document Interface (NDI): An interactive process with this document interface does not present a document to a user. It has no menus. It is typically used for 'background' interactive processes. The interactive process is allowed to open dialogues, timers, and receivers. The type constructor that defines a NDI process is `NDIProcess`:

```
:: NDIProcess p
 = E..l:NDIProcess l (ProcessInit      (PSt l p))
                     [ProcessAttribute (PSt l p)]
```

Single Document Interface (SDI): An interactive process with this document interface presents exactly one document to the user. All menu commands are associated with this document. The interactive process is allowed to open dialogues, timers, and receivers. The type constructor that defines a SDI process is `SDIProcess`:

```
:: SDIProcess wdef p
```

```
 = E..l ls:SDIProcess l ls (wdef          ls (PSt l p))
                           (ProcessInit      (PSt l p))
                           [ProcessAttribute (PSt l p)]
```

Multiple Document Interface (MDI): An interactive process with this document interface can present an arbitrary number of documents to the user (even zero). In its menu system all available commands are presented. The interactive process is allowed to open dialogues, timers, and receivers.

```
:: MDIProcess p
 = E..l:MDIProcess l (ProcessInit      (PSt l p))
                     [ProcessAttribute (PSt l p)]
```

(The function `startIO` actually creates a MDI process.)

The type constructor definitions of each of the three kinds of interactive processes are identical in case of NDI and MDI processes. A SDI process has two additional parameters: `ls` and `(wdef ls (PSt l p))`. The type constructor variable `wdef` must be a `Windows` type constructor class instance. The variable `ls` is the local state of that window. A SDI process is allowed to close this window, but can not open another window.

The interactive process type constructors are polymorphic in the public process state `p`. Each constructor introduces an initial local process state `l` and encapsulates it using existential quantification. The initialisation actions are given by a list of process state transition functions:

```
::  ProcessInit ps :== [IdFun ps]
::  IdFun       ps :== ps -> ps
```

Process definitions can be attributed with the following alternatives:

ProcessWindowPos, ProcessWindowSize, ProcessWindowResize: Depending on the underlying platform and document interface of the interactive process, the *process window* is the root window in which all user interface elements are created. On a Macintosh the process window is simply the screen. On Windows(95/NT) it is also the screen in case of NDI and SDI processes. For MDI processes it is a window that can be resized by the user. If this happens, and a `ProcessWindowResize` attribute has been specified, then a function of type `ProcessWindowResizeFunction` is evaluated:

```
::  ProcessWindowResizeFunction ps
:== Size -> Size -> ps -> ps
```

The two `Size` parameters are the size of the process window before and after resizing.

ProcessHelp, ProcessAbout: These two attributes can be provided by an interactive process to display information about itself. The `ProcessHelp` attribute typically displays online 'user manual' information. The `ProcessAbout` attribute typically displays version information.

ProcessActivate, ProcessDeactivate: These two attributes correspond closely to the `WindowActivate` and `WindowDeactivate` attributes. Recall that keyboard and mouse input is always directed to the so called active window. The parent interactive process that contains this window is the active process. If the input focus is moved to a window that is not owned by the interactive process then the `ProcessDeactivate` attribute function is evaluated to inform the program that the interactive process has become inactive. If an inactive process obtains the input focus, its `ProcessActivate` attribute function is evaluated to inform the program that it has become active again.

ProcessClose: This attribute corresponds closely to the `WindowClose` attribute. If for some reason the interactive process is requested to be closed, its `ProcessClose` attribute function is evaluated. It can take the opportunity to save data to disk and to ask the user if the process can be closed safely. It is however the responsibility of the program to terminate the process.

ProcessShareGUI: Interactive processes have a private graphical user interface administration, encoded by the `IOSt` field of their process state record. In the default case the graphical user interface elements are maintained together as if the interactive process is an independent application: activating a (root)window or dialogue brings the whole user interface structure into the foreground. When this attribute is set, the user interface elements of this interactive process (the *sub process*) will be merged with user interface elements of the interactive process that spawned it (the *parent process*).

ProcessNoWindowMenu: This attribute is valid only for MDI processes. In the default case every MDI process has a special "Window" menu (discussed in Section 10.3). If this attribute is set, then this menu is not added to the menu system of the interactive process. (This is actually the case for the MDI processes that are created by `startIO`.)

13.2 Interactive process creation

As mentioned briefly in the beginning of this chapter, interactive processes can be created one by one, or by groups. In this section we will first discuss individual creation of interactive processes in Section 13.2.1, and then have a look at the creation of multiple interactive processes in Section 13.2.2. Finally, we discuss the relations between interactive processes in Section 13.2.3.

13.2.1 Creating single processes

The basic function to create individual processes from a `World` environment is the function `startIO` that we have encountered many times. For completeness we repeat its type definition once more:

```
startIO :: !.l !.p !(ProcessInit      (PSt .l .p))
                   ![ProcessAttribute (PSt .l .p)]
                   !*World
                -> *World
```

Interactive processes can also create individual processes. Ofcourse they can not use `startIO` for this purpose because the `World` environment is not retrievable from this context. For this purpose the overloaded function `shareProcesses` is available:

```
class shareProcesses pdef :: !(pdef .p) !(PSt .l .p) -> PSt .l .p

instance shareProcesses  NDIProcess
instance shareProcesses (SDIProcess wdef) | Windows wdef
instance shareProcesses  MDIProcess
```

When applied to a NDI, SDI, or MDI process definition, `shareProcesses` adds an initial version of that interactive process to the process state administration. At some point in time that interactive process will initialise itself and join the game. So process creation is asynchronous.

From the type definition of `shareProcesses` one can see that the public process state components of the process states have the same type. One should also observe that no new public process state value is introduced. Interactive processes that are created by `shareProcesses` *share* the public state component of the process that spawned them. Interactive processes that share the same public process state value constitute a *process group*. Because interactive processes run interleaved, the public process state component can be changed in turn by each of the members of that group. This explains why it is called 'public'.

If one wants to spawn an interactive process that has a public process state of its own using only the `shareProcesses` class instances is not sufficient because they assume a public process state is already given. For this purpose the `ProcessGroup` type constructor (module `StdProcessDef`) can be used:

```
:: ProcessGroup pdef
 = E..p:ProcessGroup p (pdef p)
```

`ProcessGroup` introduces a public process state value and encapsulates it using existential quantification. The type constructor variable `pdef` can be any of the instances of the `shareProcesses` class defined above. In this way we obtain a function that can spawn an interactive process with an independent public process state:

```
class Processes pdef where
    startProcesses :: !pdef !*World      -> *World
    openProcesses  :: !pdef !(PSt .l .p) -> PSt .l .p

instance Processes (ProcessGroup pdef) | shareProcesses pdef
```

The two constructor class functions spawn an interactive process that does not share its public process state with the process that created it (in case of `openProcesses`). We also see the other process creation function that operates on a `World` environment, `startProcesses`. It is this function that is used by `startIO` to do its job. This is the way it is implemented:

```
startIO :: !.l !.p !(ProcessInit      (PSt .l .p))
                   ![ProcessAttribute (PSt .l .p)]
                   !*World
                -> *World
startIO local public initialise attributes world
 = startProcesses
   ( ProcessGroup public
     ( MDIProcess local initialise [ProcessNoWindowMenu:attributes]
     )
   ) world
```

13.2.2 Creating multiple processes

In the previous section we have shown all functions that are needed to create individual interactive processes. Because these functions are overloaded they are also suited to create a number of processes within one function application. This is achieved by declaring suitable *glueing* instances for the type constructor classes `Processes` (`:^:` and `[]`) and `shareProcesses` (`:~:`, and `ListCS`). Except for lists, these type definitions can be found in module `StdIOCommon` (Appendix A.12).

`:^:` The type constructor `:^:` glues two arbitrary type constructors. Its type constructor definition and `Processes` class instance declaration are as follows:

```
:: :^: t1 t2 = (:^:) infixr 9 t1 t2

instance Processes (:^: pdef1 pdef2) | Processes pdef1
                                     & Processes pdef2
```

Given two `Processes` instances `p1` and `p2`, then the expression `p1:^:p2` is also a `Processes` instance. Because `:^:` is right associative, an expression such as `p1 :^: p2 :^: p3` should be read as `p1 :^: (p2 :^: p3)`.

`[]` In principle the `:^:` glue is sufficient to create all required process structures. In case of working with a number of process instances of the *same type*, it is much more convenient to use lists and list comprehensions. Because the process type constructors are not parameterised we can use the normal lists for this purpose. So we get the following `Processes` class instance declaration:

```
instance Processes [pdef] | Processes pdef
```

Given a list of `Processes` instances `ps = [`$\mathtt{p}_1$ `...`$\mathtt{p}_n$`]`, then the expression `ps` itself is also a `Processes` instance.

`:~:` The type constructor `:~:` glues two type constructors that work on the same context. Its type constructor definition and `shareProcesses` class instance declaration are as follows:

```
:: :~: t1 t2 context = (:~:) infixr 9 (t1 context) (t2 context)

instance shareProcesses (:~: pdef1 pdef2) | shareProcesses pdef1
                                          & shareProcesses pdef2
```

Given two `shareProcesses` instances `p1` and `p2` working on the same context state of type `context`, then the expression `p1 :~: p2` is also a `shareProcesses` instance working on the same context state. Because `:~:` is right associative, an expression such as `p1 :~: p2 :~: p3` should be read as `p1 :~: (p2 :~: p3)`.

`ListCS` In principle the `:~:` glue is sufficient to create all required process group structures. In case of working with a number of process group instances of the *same type*, it is much more convenient to use lists and list comprehensions. This glue is provided by the type constructor `ListCS`. Its type constructor definition and `shareProcesses` class instance declaration is as follows:

```
:: ListCS t context = ListCS [t context]

instance shareProcesses (ListCS pdef) | shareProcesses pdef
```

Given a list of `shareProcesses` instances `ps = [p`$_1$` ...p`$_n$`]`, working on the same context state of type `context`, then the expression `ListCS ps` is also a `shareProcesses` instance working on the same context state.

13.2.3 Process relations

An interactive program in general consists of a number of process groups, each of which consists of a number of interactive processes. As explained above, interactive processes and process groups can be created dynamically.

Except for sub processes, there is no special parent-child relationship between an interactive process that creates an interactive process. For instance, termination of one process (using `closeProcess`) has no consequence for the other processes. Sub processes are the exception to the rule (recall that sub processes have the `ProcessShareGUI` attribute set (Section 13.1, page 119). Closing an interactive process closes also all of its sub processes.

Process groups exist by virtue of their element processes. As soon as all interactive processes of one process group have been closed, also the process group and the shared public state are closed.

The process creation functions that work on the `World` environment, `startProcesses` and its derived function `startIO`, terminate as soon as all of the process groups that have been created during their life cycle have been closed.

13.3 Examples

In this section we give some examples of the use of interactive processes.

13.3.1 Talk revisited

In this example we have a new look at the talk example of Section 12.4.1. In that version, the program created one interactive process, using `startIO`, which opened the two talk windows. In the new version for each talk window we create an interactive process. Receivers are still used to send the user typed messages to each of the talk windows. The menu is now created for both processes. Below we discuss the differences.

The initialisation of the new talk program is ofcourse different. Creation of the menu is now moved to `openTalkWindow`. The talk processes are going to be created as one process group. This implies that the `RId`s are also created earlier. We obtain the following `Start` rule:

```
Start :: *World -> *World
Start world
  # (a,world) = openRId world
  # (b,world) = openRId world
  = startProcesses
      (ProcessGroup NoState
        (ListCS [talk "A" a b,talk "B" b a]
        )
      ) world
where
  talk :: String (RId Message) (RId Message) -> MDIProcess .p
```

```
  talk name me you
    = MDIProcess NoState [openTalkWindow name me you]
                         [ProcessNoWindowMenu]
```

The menu definition that is moved to `openTalkWindow` is almost identical to the old version. The only difference is termination of the application. In the old version termination was no issue because there was only one interactive process. In the new version, closing one talk process does not close the other. Instead, before closing its parent process, the `"Quit"` function sends a new message to the other process to request termination. To do this the message type needs to be changed. The message type is now the following algebraic data type:

```
:: Message
 = NewLine String
 | Quit
```

Instead of sending a `String s` in the old version we send the value `(NewLine s)` in the new version. The request to terminate is done by sending the `Quit` message. The `"Quit"` `MenuFunction` is now defined as:

```
quit :: (PSt .l .p) -> PSt .l .p
quit ps
 = closeProcess (snd (asyncSend you Quit ps))
```

Ofcourse the definition of the receiver function `receive` must be changed accordingly:

```
receive :: Id Id Message (PSt .l .p) -> PSt .l .p
receive wId outId (NewLine text) ps
 = appPIO (setWindow wId [setControlTexts      [(outId,text)]
                         ,setEditControlCursor outId (size text)
                         ])  ps
receive _ _ Quit ps
 = closeProcess ps
```

On receipt of the `Quit` message, the receiver only needs to terminate its parent process, knowing that the requesting process will terminate itself.

These are the major differences. Below is the complete program.

```
module talk

//  ********************************************************************************
//  Clean tutorial example program.
//
//  This program creates two interactive processes. Each process opens a window in
//  which the user can type text. Text that has been typed in one window is being
//  sent to the other, and vice versa.
//  ********************************************************************************

import  StdEnv, StdIO

::  Message
    =   NewLine String
    |   Quit
::  NoState
    =   NoState
```

```
Start :: *World -> *World
Start world
    #   (a,world)   = openRId world
    #   (b,world)   = openRId world
    =   startProcesses
            (ProcessGroup NoState (ListCS [talk "A" a b,talk "B" b a])
            ) world
where
    talk :: String (RId Message) (RId Message) -> MDIProcess .p
    talk name me you
        =   MDIProcess NoState [openTalkWindow name me you] [ProcessNoWindowMenu]

openTalkWindow :: String (RId Message) (RId Message) (PSt .l .p) -> PSt .l .p
openTalkWindow name me you ps
    #   menu        = Menu ("Talk "+++name)
                        (   MenuItem "Quit" [   MenuShortKey 'q'
                                            ,   MenuFunction (noLS quit)
                                            ]
                        )   []
    #   (error,ps)  = openMenu NoState menu ps
    |   error<>NoError
        =   abort "talk could not open menu."
    #   (wId,  ps)  = accPIO openId ps
    #   (inId, ps)  = accPIO openId ps
    #   (outId,ps)  = accPIO openId ps
    #   wdef        = Dialog ("Talk "+++name)
                        (   EditControl "" (hmm 50.0) 5
                                [   ControlId       inId
                                ,   ControlKeyboard inputfilter Able
                                                    (noLS1 (input wId inId you))
                                ]
                        :+: EditControl "" (hmm 50.0) 5
                                [   ControlId       outId
                                ,   ControlPos      (BelowPrev,zero)
                                ,   ControlSelectState Unable
                                ]
                        )
                        [   WindowId    wId
                        ]
    #   (error,ps)  = openDialog undef wdef ps
    |   error<>NoError
        =   abort "talk could not open window."
    #   rdef        = Receiver me (noLS1 (receive wId outId)) []
    #   (error,ps)  = openReceiver NoState rdef ps
    |   error<>NoError
        =   abort "talk could not open receiver"
    |   otherwise
        =   ps
where
    inputfilter :: KeyboardState -> Bool
    inputfilter keystate
        =   getKeyboardStateKeyState keystate<>KeyUp

    input :: Id Id (RId Message) KeyboardState (PSt .l .p) -> PSt .l .p
    input wId inId you _ ps
        #   (Just window,ps) = accPIO (getWindow wId) ps
            text             = fromJust (snd (hd (getControlTexts [inId] window)))
        =   snd (asyncSend you (NewLine text) ps)

    receive :: Id Id Message (PSt .l .p) -> PSt .l .p
    receive wId outId (NewLine text) ps
        =   appPIO (setWindow wId [ setControlTexts      [(outId,text)]
                                  , setEditControlCursor outId (size text)
                                  ])  ps
    receive _ _ Quit ps
        =   closeProcess ps
```

```
quit :: (PSt .l .p) -> PSt .l .p
quit ps
    =   closeProcess (snd (asyncSend you Quit ps))
```

13.3.2 Clock revisited

In this example we are going to turn the clock example of Section 11.1.2 into a stopwatch component that can be added in an arbitrary interactive process. The stopwatch commands will be to *reset* timing, *pause* timing, *continue* timing, and *close* the stopwatch component. All stopwatch definitions are placed in the module `stopwatch.icl`. The function `stopwatch` defines the stopwatch component. A main module, `usestopwatch.icl` that opens and controls the stopwatch is also defined. We first look at the stopwatch and then at the main program.

The stopwatch component

The original clock program created an interactive process with three timers and a dialogue. Each of the three timers changes a *local* state that keeps track of the elapsed seconds, minutes, and hours. This situation is schematised in Figure 13.1.

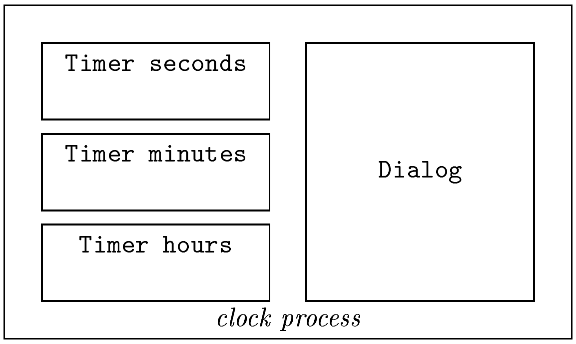


Figure 13.1: The structure of the clock process.

The stopwatch process is controlled by sending messages to a 'gateway' receiver. The timers are extended with a receiver component that handle the commands *reset*, *pause*, and *continue*. These commands will be sent to them by the gateway receiver. This gateway receiver handles the *close* command. Finally, the stopwatch process creates the same dialogue as the original clock program. The stopwatch process is schematised in Figure 13.2.

So the new components are the gateway receiver and the receiver components of the timers. We first look at the gateway receiver and then timer receivers. Both receivers accept messages of the following type:

```
:: StopwatchCommands
 = Reset
 | Pause
 | Continue
 | Close
```

The alternatives of the algebraic type `StopwatchCommands` correspond ofcourse with the stopwatch commands *reset*, *pause*, *continue*, and *close*. The gateway receiver

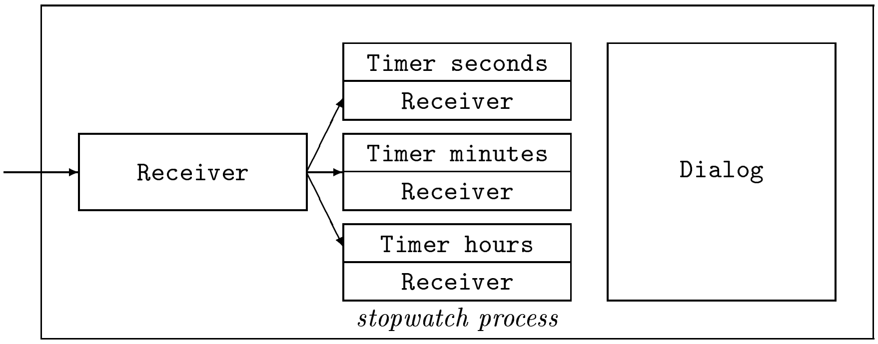


Figure 13.2: The structure of the stopwatch process.

function `receive`, on receiving the `Close` message simply terminates the interactive process by applying `closeProcess` to its process state. Every other message is routed to the timer receiver components which are identified by the list `timerinfos`. Here is the function definition of `receive`:

```
receive :: StopwatchCommands (PSt .l .p) -> PSt .l .p
receive Close ps
 = closeProcess ps
receive msg ps
 = snd (seqList [syncSend timerRId msg\\{timerRId}<-timerinfos] ps)
```

The timer receiver components receive only the `StopwatchMessage` alternatives `Reset`, `Pause`, and `Continue`. In the clock example, the timer was parameterised with its timer interval. In this example, we also need to identify both the timer and its receiver component. So the definition of a stopwatch timer now is:

```
tdef :: TimerInfo
     -> Timer (Receiver StopwatchCommands) Int (PSt .l .p)
tdef {timerId,timerRId,timerInterval}
 = Timer timerInterval
   ( Receiver timerRId receive []
   )
   [ TimerId       timerId
   , TimerFunction tick
   ]
```

The *reset* command should set the timer back to zero. One might suppose that it is sufficient to change only the value of the local state to zero, but that is not completely true. Resetting the stopwatch can occur at any moment. At that moment the timer should be synchronised with its local state. This can be done by first *disabling* and then *enabling* the timer. For this purpose the `StdTimer` functions `disableTimer` and `enableTimer` should be used. The reason that is works is because `enableTimer`, when applied to a disabled timer, synchronises the timer with the moment of evaluation. It does nothing in case the indicated timer was already enabled. Finally, on receiving the `Reset` message, the timer receiver component must set the corresponding text field of the dialogue to zero. This gives the following definition of the `Reset` alternative.

```
receive Reset (time,ps)
```

```
 # ps = appListPIO [disableTimer timerId,enableTimer timerId] ps
 # ps = setText (textid timerInterval) "00" ps
 = (0,ps)
```

The *pause* command should halt the timer until further notice (either *reset* or *continue*). This is easily done by disabling the timer:

```
receive Pause (time,ps)
 = (time,appPIO (disableTimer timerId) ps)
```

The *continue* command should let the timer continue from where it was paused. This is easily done by enabling the timer:

```
receive Continue (time,ps)
 = (time,appPIO (enableTimer timerId) ps)
```

The final details of the stopwatch component are to create the proper identification values for the timers and their receiver components, and to export its definition as an interactive process. This is done by the function `stopwatch`. The stopwatch process is defined as a process group with no interesting local or public process state (using the ubiquitous `NoState` singleton type constructor). Its initialisation actions first create the `Ids` necessary for the dialogue, and then the parameters required for the timers. Then initialisation proceeds as described above. The interesting aspect of this definition is that the process has the `ProcessShareGUI` attribute set (Section 13.1, page 119). This attribute makes sure that the dialogue of the stopwatch process shares the windows stack (Section 8.1.2) of the parent process.

```
stopwatch :: (RId StopwatchCommands) -> ProcessGroup NDIProcess
stopwatch rid
 = ProcessGroup NoState
    ( NDIProcess NoState [initialise‘] [ProcessShareGUI]
    )
where
 initialise‘ ps
  # (dialogIds, ps) = accPIO openDialogIds  ps
  # (timerInfos,ps) = accPIO openTimerInfos ps
  = initialise rid dialogIds timerInfos ps
```

For completeness, the definition module and implementation module of the stopwatch component are given below.

```
definition module stopwatch

//  ********************************************************************************
//  Clean tutorial example program.
//
//  This module exports the types and functions needed to incorporate a stopwatch
//  component.
//  ********************************************************************************

import StdIO

::  StopwatchCommands
    =   Reset
    |   Pause
    |   Continue
    |   Close

stopwatch :: (RId StopwatchCommands) -> ProcessGroup NDIProcess
```

```
implementation module stopwatch

//  ****************************************************************************
//  Clean tutorial example program.
//
//  This program defines a stopwatch process component.
//  It uses three timers to track the seconds, minutes, and hours separately.
//  Message passing is used to reset, pause, and continue timing.
//  The current time is displayed using a dialogue.
//  ****************************************************************************

import StdEnv,StdIO

::  NoState
    =   NoState
::  DialogIds
    =   {   secondsId       :: Id
        ,   minutesId       :: Id
        ,   hoursId         :: Id
        ,   dialogId        :: Id
        }
::  TimerInfo
    =   {   timerId         :: Id
        ,   timerRId        :: RId StopwatchCommands
        ,   timerInterval   :: TimerInterval
        }
::  StopwatchCommands
    =   Reset
    |   Pause
    |   Continue
    |   Close

second  :== ticksPerSecond
minute  :== 60*second
hour    :== 60*minute

openDialogIds :: *env -> (DialogIds,*env) | Ids env
openDialogIds env
    #   ([secondsid,minutesid,hoursid,dialogid:_],env) = openIds 4 env
    =   (   {   secondsId=secondsid
            ,   minutesId=minutesid
            ,   hoursId  =hoursid
            ,   dialogId =dialogid
            }
        ,   env
        )

openTimerInfos :: *env -> ([TimerInfo],*env) | Ids env
openTimerInfos env
    #   (tids,env)  = openIds  3 env
    #   (rids,env)  = openRIds 3 env
    #   intervals   = [second,minute,hour]
    =   (   [   {timerId=tid,timerRId=rid,timerInterval=i}
            \\  tid<-tids & rid<-rids & i<-intervals
            ]
        ,   env
        )

stopwatch :: (RId StopwatchCommands) -> ProcessGroup NDIProcess
stopwatch rid
    =   ProcessGroup NoState (NDIProcess NoState [initialise`] [ProcessShareGUI])
where
    initialise` ps
        #   (dialogIds,ps)  = accPIO openDialogIds ps
        #   (timerInfos,ps) = accPIO openTimerInfos ps
        =   initialise rid dialogIds timerInfos ps
```

```
initialise :: (RId StopwatchCommands) DialogIds [TimerInfo]
              (PSt .l .p) -> (PSt .l .p)
initialise rid {secondsId,minutesId,hoursId,dialogId} timerinfos ps
    #   (errors,ps)     = seqList [  openTimer 0 (tdef timerinfo)
                                  \\ timerinfo<-timerinfos
                                  ]  ps
    |   any ((<>) NoError) errors
        =   closeProcess ps
    #   (error,ps)      = openDialog NoState ddef ps
    |   error<>NoError
        =   closeProcess ps
    #   (error,ps)      = openReceiver NoState rdef ps
    |   error<>NoError
        =   closeProcess ps
    |   otherwise
        =   ps
where
    tdef {timerId,timerRId,timerInterval}
        =   Timer timerInterval
                (   Receiver timerRId receive []
                )
                [   TimerId         timerId
                ,   TimerFunction   tick
                ]
    where
        tick nrElapsed (time,ps)
            #   time    = (time+nrElapsed) mod (maxunit timerInterval)
            =   (time,setText (textid timerInterval) (toString time) ps)

        setText id text ps
            =   appPIO (setWindow dialogId [setControlTexts [(id,text)]]) ps

        receive Reset (time,ps)
            #   ps  = appListPIO [disableTimer timerId,enableTimer timerId] ps
            #   ps  = setText (textid timerInterval) "00" ps
            =   (0,ps)
        receive Pause (time,ps)
            =   (time,appPIO (disableTimer timerId) ps)
        receive Continue (time,ps)
            =   (time,appPIO (enableTimer timerId) ps)

        textid interval
            |   timerInterval==second
                =   secondsId
            |   timerInterval==minute
                =   minutesId
            |   timerInterval==hour
                =   hoursId
        maxunit interval
            |   timerInterval==second
                =   60
            |   timerInterval==minute
                =   60
            |   timerInterval==hour
                =   24

    ddef=   Dialog "Stopwatch"
                (   CompoundControl
                (   ListLS  [  TextControl text [ControlPos (Left,zero)]
                            \\ text<-["Hours:","Minutes:","Seconds:"]
                            ]
                )   []
                :+: CompoundControl
                (   ListLS  [  TextControl "00" [ControlPos (Left,zero)
                                                ,ControlId  id
                                                ]
                            \\ id<-[hoursId,minutesId,secondsId]
```

```
                       ]
            )   []
            )
            [   WindowClose (noLS closeProcess)
            ,   WindowId    dialogId
            ]
rdef
    =   Receiver rid (noLS1 receive) []
where
    receive Close ps
        =   closeProcess ps
    receive msg ps
        =   snd (seqList [syncSend timerRId msg \\ {timerRId}<-timerinfos] ps)
```

Using the stopwatch

Given the stopwatch component module `stopwatch`, we can create a program that opens, uses, and closes the stopwatch. The program will be as simple as possible. As its initialisation action it will create a `RId` needed to open the stopwatch component. It also opens a menu, `mdef`, that triggers the stopwatch commands *reset*, *pause*, *continue*, and *close*. It also contains the *quit* command to terminate the whole program. Its definition is as follows:

```
mdef
 = Menu "Stopwatch"
   (   MenuItem "Reset"    [MenuFunction (noLS (send Reset))]
   :+: MenuItem "Pause"    [MenuFunction (noLS (send Pause))]
   :+: MenuItem "Continue" [MenuFunction (noLS (send Continue))]
   :+: MenuItem "Close"    [MenuFunction (noLS (send Close))]
   :+: MenuSeparator       []
   :+: MenuItem "Quit"     [MenuFunction (noLS closeProcess)]
   )   []
```

Each of the stopwatch commands menu functions is defined by the `send` function which is parameterised with the corresponding `StopwatchCommands` message alternative. Its purpose is to send its argument message to the gateway receiver of the stopwatch process, identified by `stopwatchid`. If this fails it also emits a system beep. Here is its definition:

```
send msg ps
 # (error,ps) = syncSend stopwatchid msg ps
 | case error of SendOk->False; _->True
   = appPIO beep ps
 | otherwise
   = ps
```

Here is the complete code of the main program.

```
module usestopwatch

//  ********************************************************************************
//  Clean tutorial example program.
//
//  This program creates a simple program that uses the stopwatch process.
//  The program only has a menu to open the stopwatch and control it.
//  ********************************************************************************
```

```
import StdEnv, StdIO
import stopwatch

::  NoState
    =   NoState

Start :: *World -> *World
Start world
    #   (stopwatchid,world) = openRId world
    =   startIO NoState NoState [initialise stopwatchid] [] world

initialise :: (RId StopwatchCommands) (PSt .l .p) -> PSt .l .p
initialise stopwatchid ps
    #   ps          = openProcesses (stopwatch stopwatchid) ps
    #   (error,ps)  = openMenu NoState mdef ps
    |   error<>NoError
        =   closeProcess ps
    |   otherwise
        =   ps
where
    mdef    = Menu "Stopwatch"
                (   MenuItem "Reset"    [MenuFunction (noLS (send Reset))]
                :+: MenuItem "Pause"    [MenuFunction (noLS (send Pause))]
                :+: MenuItem "Continue" [MenuFunction (noLS (send Continue))]
                :+: MenuItem "Close"    [MenuFunction (noLS (send Close))]
                :+: MenuSeparator          []
                :+: MenuItem "Quit" [MenuFunction (noLS closeProcess)]
                )   []
    send msg ps
        #   (error,ps)  = syncSend stopwatchid msg ps
        |   case error of SendOk->False; _->True
            =   appPIO beep ps
        |   otherwise
            =   ps
```

Appendix A

I/O library

A.1 StdBitmap

```
definition module StdBitmap


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdBitmap contains functions for reading bitmap files and drawing bitmaps.
//  ********************************************************************************


import  StdMaybe
from    StdFile import FileSystem
import  StdPicture


getBitmapSize   :: !Bitmap -> Size
/*  getBitmapSize returns the size of the given bitmap.
    In case the bitmap is the result of an erroneous openBitmap, then the size is
    zero.
*/

::  Bitmap

openBitmap      :: !{#Char} !*env -> (!Maybe Bitmap,!*env)  | FileSystem env
/*  openBitmap reads in a bitmap from file.
    The String argument must be the file name of the bitmap.
    If the bitmap could be read, then (Just bitmap) is returned, otherwise Nothing
    is returned.
*/

instance Drawables Bitmap
/*  draw        bitmap
        draws the given bitmap with its left top at the current pen position.
    drawAt pos bitmap
        draws the given bitmap with its left top at the given pen position.
*/
```

A.2 StdClipboard

```
definition module StdClipboard


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdClipboard specifies all functions on the clipboard.
//  ********************************************************************************


import  StdMaybe
from    iostate import  PSt, IOSt


//  Clipboard data items:

::  ClipboardItem

class Clipboard item where
    toClipboard     :: !item            -> ClipboardItem
    fromClipboard   :: !ClipboardItem   -> Maybe item
/*  toClipboard
        makes an item transferable to the clipboard.
    fromClipboard
        attempts to retrieve an item of the instance type from the clipboard item.
        If this fails, the result is Nothing, otherwise it is (Just item).
*/

instance Clipboard {#Char}


//  Access to the current content of the clipboard:

setClipboard :: ![ClipboardItem] !(PSt .l .p) -> PSt .l .p
getClipboard :: !(PSt .l .p) -> (![ClipboardItem],!PSt .l .p)
/*  setClipboard
        replaces the current content of the clipboard with the argument list.
        Of the list only the first occurence of a ClipboardItem of the same type
        will be stored in the clipboard.
        Note that setClipboard [] erases the clipboard.
    getClipboard
        gets the current content of the clipboard without changing the content.
*/


clipboardHasChanged :: !(PSt .l .p) -> (!Bool,!PSt .l .p)
/*  clipboardHasChanged holds if the current content of the clipboard is different
    from the last access to the clipboard.
*/
```

A.3 StdControl

```
definition module StdControl


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdControl specifies all control operations.
//  Changing controls in a window/dialogue requires a *WState.
//  Reading the status of controls requires a WState.
//  ********************************************************************************


import  StdControlDef, StdMaybe
from    iostate     import IOSt


::  WState


getWindow               :: !Id !(IOSt .l .p) -> (!Maybe WState, !IOSt .l .p)
/*  getWindow returns a read-only WState for the indicated window.
    In case the indicated window does not exist Nothing is returned.
*/

setWindow               :: !Id ![IdFun *WState] !(IOSt .l .p) -> IOSt .l .p
/*  Apply the control changing functions to the current state of the indicated
    window.
    In case the indicated window does not exist nothing happens.
*/


/*  Functions that change the state of controls.
    When applied to unknown Ids these functions have no effect.
*/
showControls            :: ![Id]                            !*WState -> *WState
hideControls            :: ![Id]                            !*WState -> *WState
/*  (show/hide)Controls makes the indicated controls visible/invisible.
    Hiding a control overrides the visibility of its elements, which become
        invisible.
    Showing a hidden control re-establishes the visibility state of its elements.
*/

enableControls          :: ![Id]                            !*WState -> *WState
disableControls         :: ![Id]                            !*WState -> *WState
/*  (en/dis)ableControls (en/dis)ables the indicated controls.
    Disabling a control overrides the SelectStates of its elements, which become
        unselectable.
    Enabling a disabled control re-establishes the SelectStates of its elements.
*/

markCheckControlItems   :: !Id ![Index]                     !*WState -> *WState
unmarkCheckControlItems :: !Id ![Index]                     !*WState -> *WState
/*  (unm/m)arkCheckControlItems unmarks/marks the indicated check items of the given
    CheckControl. Indices range from 1 to the number of check items. Illegal indices
    are ignored.
*/

selectRadioControlItem  :: !Id  !Index                      !*WState -> *WState
/*  selectRadioControlItem marks the indicated radio item of a RadioControl, causing
    the mark of the previously marked radio item to disappear. The item is given by
    the Id of the RadioControl and its index position (counted from 1).
*/

selectPopUpControlItem  :: !Id  !Index                      !*WState -> *WState
/*  selectPopUpControlItem marks the indicated popup item of a PopUpControl, causing
```

```
    the mark of the previously marked popup item to disappear. The item is given by
    the Id of the PopUpControl and its index position (counted from 1).
*/

moveControlViewFrame    :: !Id Vector                        !*WState -> *WState
/*  moveControlViewFrame moves the orientation of the CompoundControl over the given
    vector, and updates the control if necessary. The control frame is not moved
    outside the ViewDomain of the control. MoveControlViewFrame has no effect if the
    indicated control has no ControlDomain attribute.
*/

setControlTexts         :: ![(Id,String)]                    !*WState -> *WState
/*  setControlTexts sets the text of the indicated (Text/Edit/Button)Controls.
    If the indicated control is a (Text/Button)Control, then AltKey are interpreted
    by the system.
    If the indicated control is an EditControl, then the text is taken as it is.
*/

setEditControlCursor    :: !Id !Int                          !*WState -> *WState
/*  setEditControlCursor sets the cursor at position @2 of the current content of
    the EditControl.
    In case @2<0, then the cursor is set at the start of the current content.
    In case @2>size content, then the cursor is set at the end of the current
    content.
*/

setControlLooks         :: ![(Id,Bool,Look)]                 !*WState -> *WState
/*  setControlLooks applied to a CompoundControl turns it into a non-transparant
    CompoundControl.
    Setting the Look only redraws the indicated controls if the corresponding
    Boolean is True.
*/

setSliderStates         :: ![(Id,SliderState->SliderState)] !*WState -> *WState
setSliderThumbs         :: ![(Id,Int)]                       !*WState -> *WState
/*  setSliderStates
        applies the function to the current SliderState of the indicated
        SliderControl and redraws the settings if necessary.
    setSliderThumbs
        sets the new thumb value of the indicated SliderControl and redraws the
        settings if necessary.
*/


drawInControl           :: !Id ![DrawFunction]               !*WState -> *WState
/*  Draw in a (Custom(Button)/Compound)Control. If the CompoundControl is
    transparant then this operation has no effect.
*/


getControlTypes         ::          !WState -> [(ControlType,Maybe Id)]
getCompoundTypes        :: !Id      !WState -> [(ControlType,Maybe Id)]
/*  getControlTypes
        yields the list of ControlTypes of the component controls of this window.
    getCompoundTypes
        yields the list of ControlTypes of the component controls of this
        CompoundControl.
    For both functions (Just id) is yielded if the component control has a
    (ControlId id) attribute, and Nothing otherwise. Component controls are not
    collected recursively through CompoundControls.
    If the indicated CompoundControl is not a CompoundControl, then [] is yielded.
*/

getControlLayouts       :: ![Id] !WState -> [(Bool,(Maybe ItemPos,ItemOffset))]
                                                        // (Nothing,zero)
getControlViewSizes     :: ![Id] !WState -> [(Bool,Size)]              // zero
getControlSelectStates  :: ![Id] !WState -> [(Bool,SelectState)]       // Able
```

```
getControlShowStates    :: ![Id] !WState -> [(Bool,Bool)]              // False
getControlTexts         :: ![Id] !WState -> [(Bool,Maybe String)]      // Nothing
getControlNrLines       :: ![Id] !WState -> [(Bool,Maybe NrLines)]     // Nothing
getControlLooks         :: ![Id] !WState -> [(Bool,Maybe Look)]        // Nothing
getControlMinimumSizes  :: ![Id] !WState -> [(Bool,Maybe Size)]        // Nothing
getControlResizes       :: ![Id] !WState -> [(Bool,Maybe ControlResizeFunction)]
                                                                       // Nothing
getRadioControlItems    :: ![Id] !WState -> [(Bool,Maybe [TextLine])]  // Nothing
getRadioControlSelection:: ![Id] !WState -> [(Bool,Maybe Index)]       // Nothing
getCheckControlItems    :: ![Id] !WState -> [(Bool,Maybe [TextLine])]  // Nothing
getCheckControlSelection:: ![Id] !WState -> [(Bool,Maybe [Index])]     // Nothing
getPopUpControlItems    :: ![Id] !WState -> [(Bool,Maybe [TextLine])]  // Nothing
getPopUpControlSelection:: ![Id] !WState -> [(Bool,Maybe Index)]       // Nothing
getSliderDirections     :: ![Id] !WState -> [(Bool,Maybe Direction)]   // Nothing
getSliderStates         :: ![Id] !WState -> [(Bool,Maybe SliderState)] // Nothing
getControlViewFrames    :: ![Id] !WState -> [(Bool,Maybe ViewFrame)]   // Nothing
getControlViewDomains   :: ![Id] !WState -> [(Bool,Maybe ViewDomain)]  // Nothing
getControlItemSpaces    :: ![Id] !WState -> [(Bool,Maybe (Int,Int))]   // Nothing
getControlMargins       :: ![Id] !WState -> [(Bool,Maybe ((Int,Int),(Int,Int)))]
                                                                       // Nothing

/*  Functions that return the current state of controls.
    The result list is of equal length as the argument Id list. Each result list
    element corresponds in order with the argument Id list. Of each element the
    first Boolean result is False in case of invalid Ids (if so dummy values are
    returned - see comment).
    Important: controls with no ControlId attribute, or illegal ids, can not be
    found in the WState!

    getControlLayouts
        Yields (Just ControlPos) if the indicated control had a ControlPos attribute
        and Nothing otherwise. The ItemOffset offset is the exact current location
        of the indicated control (LeftTop,offset).
    getControlViewSizes
        Yields the current view frame size of the indicated control. Note that for
        any control other than the CompoundControl this is the exact size of the
        control.
    getControlSelectStates
        Yields the current SelectState of the indicated control.
    getControlShowStates
        Yields True if the indicated control is visible, and False otherwise.
    getControlTexts
        Yields (Just text) of the indicated (Text/Edit/Button)Control.
        If the control is not such a control, then Nothing is yielded.
    getControlNrLines
        Yields (Just nrlines) of the indicated EditControl.
        If the control is not such a control, then Nothing is yielded.
    getControlLooks
        Yields the Look of the indicated (Custom/CustomButton/Compound)Control.
        If the control is not such a control, or is a transparant CompoundControl,
        then Nothing is yielded.
    getControlMinimumSizes
        Yields (Just minimumsize) if the indicated control had a ControlMinimumSize
        attribute and Nothing otherwise.
    getControlResizes
        Yields (Just resizefunction) if the indicated control had a ControlResize
        attribute and Nothing otherwise.
    getRadioControlItems
        Yields the TextLines of the items of the indicated RadioControl.
        If the control is not such a control, then Nothing is yielded.
    getRadioControlSelection
        Yields the index of the selected radio item of the indicated RadioControl.
        If the control is not such a control, then Nothing is yielded.
    getCheckControlItems
        Yields the TextLines of the items of the indicated CheckControl.
        If the control is not such a control, then Nothing is yielded.
    getCheckControlSelection
```

```
        Yields the indices of the selected checkitems of the indicated CheckControl.
        If the control is not such a control, then Nothing is yielded.
    getPopUpControlItems
        Yields the TextLines of the items of the indicated PopUpControl.
        If the control is not such a control, then Nothing is yielded.
    getPopUpControlSelection
        Yields the Index of the indicated PopUpControl.
        If the control is not such a control, then Nothing is yielded.
    getSliderDirections
        Yields (Just Direction) of the indicated SliderControl.
        If the control is not such a control, then Nothing is yielded.
    getSliderStates
        Yields (Just SliderState) of the indicated SliderControl.
        If the control is not such a control, then Nothing is yielded.
    getControlViewFrames
        Yields (Just ViewFrame) of the indicated CompoundControl.
        If the control is not such a control, then Nothing is yielded.
    getControlViewDomains
        Yields (Just ViewDomain) of the indicated CompoundControl.
        If the control is not such a control, then Nothing is yielded.
    getControlItemSpaces
        Yields (Just (horizontal space,vertical space)) of the indicated
        CompoundControl.
        If the control is not such a control, then Nothing is yielded.
    getControlMargins
        Yields (Just (ControlHMargin,ControlVMargin)) of the indicated
        CompoundControl.
        If the control is not such a control, then Nothing is yielded.
*/
```

A.4 StdControlClass

```
definition module StdControlClass


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdControlClass define the standard set of controls instances.
//  ********************************************************************************


import  StdIOCommon, StdControlDef
from    windowhandle    import ControlState
from    StdPSt          import PSt, IOSt


class Controls cdef where
    controlToHandles:: !(cdef .ls (PSt .l .p)) -> [ControlState .ls (PSt .l .p)]
    getControlType  ::  (cdef .ls .ps) -> ControlType

instance Controls (AddLS  c)            | Controls c
instance Controls (NewLS  c)            | Controls c
instance Controls (ListLS c)            | Controls c
instance Controls NilLS
instance Controls ((:+:) c1 c2)         | Controls c1 & Controls c2
instance Controls RadioControl
instance Controls CheckControl
instance Controls PopUpControl
instance Controls SliderControl
instance Controls TextControl
instance Controls EditControl
instance Controls ButtonControl
instance Controls CustomButtonControl
instance Controls CustomControl
instance Controls (CompoundControl c)   | Controls c
```

A.5 StdControlDef

```
definition module StdControlDef


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdControl contains the types to define the standard set of controls.
//  ********************************************************************************


import  StdIOCommon
from    StdPicture  import  DrawFunction, Picture


::  RadioControl        ls ps
 =  RadioControl        [RadioControlItem *(ls,ps)] RowsOrColumns Index
                                                [ControlAttribute *(ls,ps)]
::  CheckControl        ls ps
 =  CheckControl        [CheckControlItem *(ls,ps)] RowsOrColumns
                                                [ControlAttribute *(ls,ps)]
::  PopUpControl        ls ps
 =  PopUpControl        [PopUpControlItem *(ls,ps)] Index
                                                [ControlAttribute *(ls,ps)]
::  SliderControl       ls ps
 =  SliderControl       Direction Length SliderState (SliderAction *(ls,ps))
                                                [ControlAttribute *(ls,ps)]
::  TextControl         ls ps
 =  TextControl         TextLine                [ControlAttribute *(ls,ps)]
::  EditControl         ls ps
 =  EditControl         TextLine Width NrLines  [ControlAttribute *(ls,ps)]
::  ButtonControl       ls ps
 =  ButtonControl       TextLine                [ControlAttribute *(ls,ps)]
::  CustomButtonControl ls ps
 =  CustomButtonControl Size Look               [ControlAttribute *(ls,ps)]
::  CustomControl       ls ps
 =  CustomControl       Size Look               [ControlAttribute *(ls,ps)]
::  CompoundControl      c ls ps
 =  CompoundControl     (c ls ps)               [ControlAttribute *(ls,ps)]

::  TextLine                :== String
::  NrLines                 :== Int
::  Width                   :== Int
::  Length                  :== Int
::  RowsOrColumns
    =   Rows    Int
    |   Columns Int
::  RadioControlItem    ps  :== (TextLine,            IOFunction ps)
::  CheckControlItem    ps  :== (TextLine, MarkState, IOFunction ps)
::  PopUpControlItem    ps  :== (TextLine,            IOFunction ps)
::  Look                    :== SelectState -> UpdateState -> [DrawFunction]
::  SliderAction        ps  :== SliderMove  -> ps -> ps
::  SliderMove
    =   SliderIncSmall
    |   SliderDecSmall
    |   SliderIncLarge
    |   SliderDecLarge
    |   SliderThumb Int

::  ControlAttribute ps                         // Default:
    =   ControlId           Id                  // no id
    |   ControlPos          ItemPos             // (RightTo previous,zero)
    |   ControlSize         Size                // system derived/overruled
    |   ControlMinimumSize  Size                // zero
    |   ControlResize       ControlResizeFunction   // no resize
    |   ControlSelectState  SelectState         // control Able
```

```
    |   ControlHide                                // initially visible
    |   ControlFunction     (IOFunction     ps)    // id
    |   ControlModsFunction (ModsIOFunction ps)    // ControlFunction
    |   ControlMouse        MouseStateFilter SelectState (MouseFunction ps)
                                                   // no mouse input/overruled
    |   ControlKeyboard     KeyboardStateFilter SelectState (KeyboardFunction ps)
                                                   // no keyboard input/overruled
//  For CompoundControls only:
    |   ControlItemSpace    Int Int                // system dependent
    |   ControlHMargin      Int Int                // system dependent
    |   ControlVMargin      Int Int                // system dependent
    |   ControlLook         Look                   // control is transparant
    |   ControlViewDomain   ViewDomain             // {zero,max range}
    |   ControlOrigin       Point                  // Left top of ViewDomain
    |   ControlHScroll      ScrollFunction         // no horizontal scrolling
    |   ControlVScroll      ScrollFunction         // no vertical   scrolling

::  ControlResizeFunction
    :== Size ->                                    // current  control size
        Size ->                                    // old      window  size
        Size ->                                    // new      window  size
        Size                                       // new      control size
::  ScrollFunction
    :== ViewFrame       ->                         // current  view
        SliderState     ->                         // current  state of scrollbar
        SliderMove      ->                         // action of the user
        Int                                        // new thumb value of scrollbar
::  ControlType
    :== String
```

A.6 StdControlReceiver

```
definition module StdControlReceiver


//	********************************************************************************
//	Clean Standard Object I/O library, version 1.0.1
//	
//	StdControlReceiver defines Receiver(2) controls instances.
//	********************************************************************************


import	StdReceiverDef, StdControlClass


instance Controls (Receiver  m  )
instance Controls (Receiver2 m r)
```

A.7 StdFileSelect

```
definition module StdFileSelect


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdFileSelect defines the standard file selector dialogue.
//  ********************************************************************************


import StdMaybe, StdString

class FileSelectEnv env where
    selectInputFile ::                 !*env -> (!Maybe String,!*env)
    selectOutputFile:: !String !String !*env -> (!Maybe String,!*env)
/*  selectInputFile
        opens a dialogue in which the user can browse the file system to select an
        existing file.
        If a file has been selected, the String result contains the complete
        pathname of the selected file.
        If the user has not selected a file, Nothing is returned.
    selectOutputFile
        opens a dialogue in which the user can browse the file system to save a
        file.
        The first argument is the prompt of the dialogue (default: "Save As:")
        The second argument is the suggested filename.
        If the indicated directory already contains a file with the indicated name,
        selectOutputFile opens a new dialogue to confirm overwriting of the existing
        file.
        If either this dialogue is not confirmed or browsing is cancelled then
        Nothing is returned, otherwise the String result is the complete pathname of
        the selected file.
*/

instance FileSelectEnv World
```

A.8 StdFont

```
definition module StdFont


//	**************************************************************************************
//	Clean Standard Object I/O library, version 1.0.1
//
//	StdFont contains the font definitions.
//	**************************************************************************************


from	osfont	import Font
import	StdFontDef


class FontEnv env where
	openFont            ::            !FontDef   !*env -> (!(!Bool,!Font),!*env)
	openDefaultFont     ::                       !*env -> (!Font,         !*env)
	openDialogFont      ::                       !*env -> (!Font,         !*env)

	getFontNames        ::                       !*env -> (![FontName],   !*env)
	getFontStyles       ::            !FontName  !*env -> (![FontStyle],  !*env)
	getFontSizes        :: !Int !Int !FontName   !*env -> (![FontSize],   !*env)

	getFontCharWidth    :: ! Char     !Font      !*env -> (!Int,          !*env)
	getFontCharWidths   :: ![Char]    !Font      !*env -> (![Int],        !*env)
	getFontStringWidth  :: ! String   !Font      !*env -> (!Int,          !*env)
	getFontStringWidths :: ![String]  !Font      !*env -> (![Int],        !*env)

	getFontMetrics      ::            !Font      !*env -> (!FontMetrics,  !*env)
/*	openFont
		creates the font as specified by the name, stylistic variations, and size.
		The Boolean result is True only if the font is available and need not be
		scaled.
		In all other cases, an existing font is returned (depending on the system).
	openDefaultFont
		returns the font used by default by applications.
	openDialogFont
		returns the font used by default by the system.

	getFontNames
		returns the FontNames  of all available fonts.
	getFontStyles
		returns the FontStyles of all available styles of a particular FontName.
	getFontSizes
		returns all FontSizes in increasing order of a particular FontName that are
		available without scaling. The sizes inspected are inclusive between the two
		Integer arguments. (Negative values are set to zero.)
		In case the requested font is unavailable, the styles or sizes of the
		default font are returned.

	getFont(Char/String)Width(s)
		return the width(s) in terms of pixels of given character(s) (string(s)) for
		a particular Font.

	getFontMetrics
		returns the metrics of a given Font in terms of pixels.
*/

instance FontEnv World

getFontDef :: !Font -> FontDef
/*	getFontDef returns the name, stylistic variations and size of the argument Font.
*/
```

A.9 StdFontDef

```
definition module StdFontDef


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdFontDef contains the font definitions.
//  ********************************************************************************


import StdInt, StdString


::  FontDef
    =   {   fName       :: !FontName    // Name of the font
        ,   fStyles     :: ![FontStyle] // Stylistic variations
        ,   fSize       :: !FontSize    // Size in points
        }
::  FontMetrics
    =   {   fAscent     :: !Int         // Distance between top    and base line
        ,   fDescent    :: !Int         // Distance between bottom and base line
        ,   fLeading    :: !Int         // Distance between two text lines
        ,   fMaxWidth   :: !Int         // Maximum character width including spacing
        }
::  FontName            :== String
::  FontStyle           :== String
::  FontSize            :== Int


//  Font constants:

SerifFontDef            :== {fName="Times",      fStyles=[],fSize=10}
SansSerifFontDef        :== {fName="Arial",      fStyles=[],fSize=10}
SmallFontDef            :== {fName="Small Fonts",fStyles=[],fSize=7 }
NonProportionalFontDef  :== {fName="Courier",    fStyles=[],fSize=10}
SymbolFontDef           :== {fName="Symbol",     fStyles=[],fSize=10}


//  Style constants:

ItalicsStyle            :== "Italic"
BoldStyle               :== "Bold"
UnderlinedStyle         :== "Underline"


//  Standard lineheight of a font is the sum of its leading, ascent and descent:
fontLineHeight fMetrics :== fMetrics.fLeading + fMetrics.fAscent + fMetrics.fDescent
```

A.10 StdId

```
definition module StdId


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdId specifies the generation functions for identification values.
//  ********************************************************************************


from    id      import  Id, RId, R2Id, RIdtoId, R2IdtoId, toString, ==
from    iostate import  IOSt

class Ids env where
    openId      ::      !*env -> (!Id,          !*env)
    openIds     :: !Int !*env -> (![Id],        !*env)

    openRId     ::      !*env -> (!RId  m,      !*env)
    openRIds    :: !Int !*env -> (![RId m],     !*env)

    openR2Id    ::      !*env -> (!R2Id  m r,   !*env)
    openR2Ids   :: !Int !*env -> (![R2Id m r],  !*env)
/*  There are three types of identification values:
    -   RId  m:     for uni-directional message passing (see StdReceiver)
    -   R2Id m r:   for bi-directional  message passing (see StdReceiver)
    -   Id:         for all other Object I/O library components
    Of each generation function there are two variants:
    -   to create exactly one identification value.
    -   to create a number of identification values.
            If the integer argument <=0, then an empty list of identification values
            is generated.
*/

instance Ids World
instance Ids (IOSt .l .p)
```

A.11 StdIO

```
definition module StdIO

//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdIO contains all definition modules of the Object I/O library.
//  ********************************************************************************

import
    StdId,                  // The operations that generate identification values
    StdIOCommon,            // Function and type definitions used in the library
    StdMaybe,               // The Maybe data type
    StdPSt,                 // Operations on PSt that are not device related
    StdSystem,              // System dependent operations

    StdFileSelect,          // File selector dialogues

    StdFontDef,             // Type definitions for font handling
    StdFont,                // Font handling operations

    StdPictureDef,          // Type definitions for picture handling
    StdPicture,             // Picture handling operations
    StdBitmap,              // Defines an instance for drawing bitmaps

    StdProcessDef,          // Type definitions for process handling
    StdProcess,             // Process handling operations

    StdClipboard,           // Clipboard handling operations

    StdControlDef,          // Type definitions for controls
    StdControlClass,        // Standard controls class instances
    StdControlReceiver,     // Receiver controls class instances
    StdControl,             // Control handling operations

    StdMenuDef,             // Type definitions for menus
    StdMenuElementClass,    // Standard menus class instances
    StdMenuReceiver,        // Receiver menus class instances
    StdMenuElement,         // Menu element handling operations
    StdMenu,                // Menu handling operations

    StdReceiverDef,         // Type definitions for receivers
    StdReceiver,            // Receiver handling operations

    StdTimerDef,            // Type definitions for timers
    StdTimerElementClass,   // Standard timer class instances
    StdTimerReceiver,       // Receiver timer class instances
    StdTimer,               // Timer handling operations
    StdTime,                // Time related operations

    StdWindowDef,           // Type definitions for windows
    StdWindow               // Window handling operations
```

A.12 StdIOCommon

```
definition module StdIOCommon


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdIOCommon defines common types and access functions for the
//  Object I/O library.
//  ********************************************************************************


import  StdOverloaded
import  StdString
from    id      import  Id, RId, R2Id, RIdtoId, R2IdtoId, toString, ==
from    key     import  SpecialKey,
                            BeginKey,
                            ClearKey,
                            DeleteKey, DownKey,
                            EndKey, EnterKey, EscapeKey,
                            F1Key,  F2Key,  F3Key,  F4Key,  F5Key,
                            F6Key, F7Key, F8Key, F9Key, F10Key,
                            F11Key, F12Key, F13Key, F14Key, F15Key,
                            HelpKey,
                            LeftKey,
                            PgDownKey, PgUpKey,
                            RightKey,
                            UpKey


/*  General type constructors for composing context-independent data structures.
*/
::  :^:     t1 t2           = (:^:) infixr 9 t1 t2


/*  General type constructors for composing context-dependent data structures.
*/
::  :~:     t1 t2       cs  = (:~:) infixr 9 (t1 cs) (t2 cs)
::  ListCS      t       cs  = ListCS [t cs]
::  NilCS               cs  = NilCS


/*  General type constructors for composing local and context-dependent
    data structures.
*/
::  :+:     t1 t2   ls  cs  = (:+:) infixr 9 (t1 ls cs) (t2 ls cs)
::  ListLS      t   ls  cs  = ListLS [t ls cs]
::  NilLS           ls  cs  = NilLS
::  NewLS       t   ls  cs  = E..new: {newLS::new, newDef:: t   new     cs}
::  AddLS       t   ls  cs  = E..add: {addLS::add, addDef:: t *(add,ls) cs}

noLS ::     (.a->.b)    (.c,.a) -> (.c,.b)  // Lift function    a  ->  b
                                            // to            (c,a)->(c,b)
noLS1:: (.x->.a->.b) .x (.c,.a) -> (.c,.b)  // Lift function x->  a  ->  b
                                            // to           x->(c,a)->(c,b)

::  Index           :== Int
::  Title           :== String


::  Vector          =   {vx::!Int,vy::!Int}

instance            ==          Vector      // @1-@2==zero
instance            +           Vector      // {vx=@1.vx+@2.vx,vy=@1.vy+@2.vy}
instance            -           Vector      // {vx=@1.vx-@2.vx,vy=@1.vy-@2.vy}
instance            zero        Vector      // {vx=0,vy=0}
```

```
instance            ~           Vector      // zero-@1

class toVector      x :: !x ->  Vector


::  Size            =   {w ::!Int,h ::!Int}

instance            ==          Size        // @1.w==@2.w && @1.h==@2.h
instance            zero        Size        // {w=0,h=0}
instance            toVector    Size        // {w,h}->{vx=w,vy=h}

::  SelectState     =   Able | Unable
::  MarkState       =   Mark | NoMark

enabled     :: !SelectState -> Bool         // @1 == Able
marked      :: !MarkState   -> Bool         // @1 == Mark

instance    == SelectState                  // Constructor equality
instance    == MarkState                    // Constructor equality
instance    ~  SelectState                  // Able <-> Unable
instance    ~  MarkState                    // Mark <-> NoMark

::  KeyboardState
    =   CharKey     Char        KeyState    // ASCII character input
    |   SpecialKey  SpecialKey  KeyState Modifiers
                                            // Special key input
::  KeyState
    =   KeyDown     IsRepeatKey             // Key is down
    |   KeyUp                               // Key goes up
::  IsRepeatKey                             // Flag on key down:
    :== Bool                                // True iff key is repeating
::  Key
    =   IsCharKey    Char
    |   IsSpecialKey SpecialKey
::  KeyboardStateFilter                     // Predicate on KeyboardState:
    :== KeyboardState -> Bool               // evaluate KeyFunction only if True

getKeyboardStateKeyState:: !KeyboardState -> KeyState
getKeyboardStateKey     :: !KeyboardState -> Key

instance    == KeyState                     // Equality on KeyState


::  MouseState
    =   MouseMove   Point Modifiers         // Mouse is up     (position,modifiers)
    |   MouseDown   Point Modifiers Int     // Mouse goes down (and nr down)
    |   MouseDrag   Point Modifiers         // Mouse is down   (position,modifiers)
    |   MouseUp     Point Modifiers         // Mouse goes up   (position,modifiers)
::  ButtonState
    =   ButtonStillUp                       // MouseMove
    |   ButtonDown                          // MouseDown _ _ 1
    |   ButtonDoubleDown                    //           _ _ 2
    |   ButtonTripleDown                    //           _ _ >2
    |   ButtonStillDown                     // MouseDrag
    |   ButtonUp                            // MouseUp
::  MouseStateFilter                        // Predicate on MouseState:
    :== MouseState -> Bool                  // evaluate MouseFunction only if True

getMouseStatePos        :: !MouseState  -> Point
getMouseStateModifiers  :: !MouseState  -> Modifiers
getMouseStateButtonState:: !MouseState  -> ButtonState

instance    == ButtonState                  // Constructor equality


::  SliderState
    =   {   sliderMin   :: !Int
```

```
        ,   sliderMax   :: !Int
        ,   sliderThumb :: !Int
        }

instance == SliderState                    // @1.sliderMin   == @2.sliderMin
                                           // @1.sliderMax   == @2.sliderMax
                                           // @1.sliderThumb == @2.sliderThumb


::  UpdateState
    =   {   oldFrame    :: !ViewFrame
        ,   newFrame    :: !ViewFrame
        ,   updArea     :: !UpdateArea
        }
::  ViewDomain          :== Rectangle
::  ViewFrame           :== Rectangle
::  UpdateArea          :== [ViewFrame]


::  Point
    =   {   x       :: !Int
        ,   y       :: !Int
        }
::  Rectangle
    =   {   corner1 :: !Point
        ,   corner2 :: !Point
        }

instance            ==          Point       // @1-@2==zero
instance            +           Point       // {x=@1.x+@2.x,y=@1.y+@2.y}
instance            -           Point       // {x=@1.x-@2.x,y=@1.y-@2.y}
instance            zero        Point       // {x=0,y=0}
instance            toVector    Point       // {x,y}->{vx=x,vy=y}

instance            ==          Rectangle   //    @1.corner1==@2.corner1
                                            // && @1.corner2==@2.corner2
instance            zero        Rectangle   // {corner1=zero,corner2=zero}

rectangleSize       :: !Rectangle -> Size   // {w=abs (@1.corner1-@1.corner2).x,
                                            //  h=abs (@1.corner1-@1.corner2).y}


::  Modifiers
    =   {   shiftDown   :: !Bool            // True iff shift   down
        ,   optionDown  :: !Bool            // True iff option  down
        ,   commandDown :: !Bool            // True iff command down
        ,   controlDown :: !Bool            // True iff control down
        ,   altDown     :: !Bool            // True iff alt     down
        }

//  Constants to check which of the Modifiers are down.

NoModifiers :== {shiftDown  = False
                ,optionDown = False
                ,commandDown= False
                ,controlDown= False
                ,altDown    = False
                }
ShiftOnly   :== {NoModifiers & shiftDown    = True}
OptionOnly  :== {NoModifiers & optionDown   = True}
CommandOnly :== {NoModifiers & commandDown  = True}
ControlOnly :== {NoModifiers & controlDown  = True}
AltOnly     :== {NoModifiers & altDown      = True}


/*  The layout language used for windows and controls.
*/
```

```
::  ItemPos
    :== (   ItemLoc
        ,   ItemOffset
        )
::  ItemLoc
    //  Absolute:
    =   Fix     Point
    //  Relative to corner:
    |   LeftTop
    |   RightTop
    |   LeftBottom
    |   RightBottom
    //  Relative in next line:
    |   Left
    |   Center
    |   Right
    //  Relative to other item:
    |   LeftOf  Id
    |   RightTo Id
    |   Above   Id
    |   Below   Id
    //  Relative to previous item:
    |   LeftOfPrev
    |   RightToPrev
    |   AbovePrev
    |   BelowPrev
::  ItemOffset
    :== Vector

instance    == ItemLoc                      // Constructor and value equality


/*  The Direction type.
*/
::  Direction
    =   Horizontal
    |   Vertical

instance    == Direction                    // Constructor equality


/*  Document interface type of interactive processes.
*/
::  DocumentInterface
    =   NDI                                 // No       Document Interface
    |   SDI                                 // Single   Document Interface
    |   MDI                                 // Multiple Document Interface

instance    == DocumentInterface            // Constructor equality


/*  Process attributes.
*/
::  ProcessAttribute ps                     // Default:
    =   ProcessWindowPos    ItemPos         // Platform dependent
    |   ProcessWindowSize   Size            // Platform dependent
    |   ProcessWindowResize (ProcessWindowResizeFunction ps)
                                            // Platform dependent
    |   ProcessHelp         (IOFunction ps) // No Help  facility
    |   ProcessAbout        (IOFunction ps) // No About facility
    |   ProcessActivate     (IOFunction ps) // No action on activate
    |   ProcessDeactivate   (IOFunction ps) // No action on deactivate
    |   ProcessClose        (IOFunction ps) // Process is closed
    |   ProcessShareGUI                     // Process does not share parent GUI
//  Attributes for MDI processes only:
    |   ProcessNoWindowMenu                 // Process has WindowMenu
```

```
::  ProcessWindowResizeFunction ps
    :== Size                                // Old ProcessWindow size
     -> Size                                // New ProcessWindow size
     -> ps -> ps


/*  Frequently used function types.
*/
::  IdFun               ps  :==                     ps -> ps
::  IOFunction          ps  :==                     ps -> ps
::  ModsIOFunction      ps  :== Modifiers       ->  ps -> ps
::  MouseFunction       ps  :== MouseState      ->  ps -> ps
::  KeyboardFunction    ps  :== KeyboardState   ->  ps -> ps


/*  Common error report types.
*/
::  ErrorReport                             // Usual cause:
    =   NoError                             // Everything went allright
    |   ErrorViolateDI                      // Violation against DocumentInterface
    |   ErrorIdsInUse                       // Object contains Ids that are bound
    |   ErrorUnknownObject                  // Object can not be found

instance    ==          ErrorReport         // Constructor equality
```

A.13 StdMaybe

```
definition module StdMaybe

//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdMaybe defines the Maybe type.
//  ********************************************************************************

::  Maybe x
    =   Just x
    |   Nothing

isJust      :: !(Maybe .x) -> Bool      // case @1 of (Just _) -> True; _ -> False
isNothing   :: !(Maybe .x) -> Bool      // not o isNothing
fromJust    :: !(Maybe .x) -> .x        // \(Just x) -> x
```

A.14 StdMenu

```
definition module StdMenu


//	********************************************************************************
//	Clean Standard Object I/O library, version 1.0.1
//
//	StdMenu defines functions on menus.
//	********************************************************************************


import	StdMenuDef, StdMenuElementClass
from	iostate		import PSt, IOSt


//	Operations on unknown Ids are ignored.

class Menus mdef where
	openMenu	:: .ls !(mdef .ls (PSt .l .p)) !(PSt .l .p)
						  -> (!ErrorReport,!PSt .l .p)
	getMenuType ::	(mdef .ls .ps)	-> MenuType
/*	Open the given menu definition for this interactive process.
	openMenu may not be permitted to open a menu depending on its DocumentInterface
		(see the comments at the shareProcesses instances in module StdProcess).
	In case a menu with the same Id is already open then nothing happens. In case
		the menu has the WindowMenuId Id then nothing happens. In case the menu does
		not have an Id, it will obtain an Id which is fresh with respect to the
		current set of menus. The Id can be reused after closing this menu. In case
		menu elements are opened with duplicate Ids, the menu will not be opened.
	In case the menu definition does not have a MenuIndex attribute (see StdMenuDef)
		it will be opened behind the last menu. In case the menu definition has a
		MenuIndex attribute it will be placed behind the menu indicated by the
		integer index.
		The index of a menu starts from one for the first present menu. If the index
		is negative or zero, then the new menu is added before the first menu. If
		the index exceeds the number of menus, then the new menu is added behind the
		last menu.
*/

instance Menus	(Menu m)	| MenuElements m


closeMenu :: !Id !(IOSt .l .p) -> IOSt .l .p
/*	closeMenu closes the indicated Menu and all of its elements.
	The WindowMenu can not be closed by closeMenu (in case the Id argument equals
	WindowMenuId).
*/


openMenuElements	:: !Id	   !Index .ls (m .ls (PSt .l .p))	!(IOSt .l .p)
						 -> (!ErrorReport,!IOSt .l .p)
						 |  MenuElements m
openSubMenuElements :: !Id !Id !Index .ls (m .ls (PSt .l .p))	!(IOSt .l .p)
						 -> (!ErrorReport,!IOSt .l .p)
						 |  MenuElements m
openRadioMenuItems  :: !Id !Id !Index ![MenuRadioItem (PSt .l .p)] !(IOSt .l .p)
						 -> (!ErrorReport,!IOSt .l .p)
/*	Add menu elements to the indicated Menu, SubMenu, or RadioMenu.
	openMenuElements:
		adds menu elements to the Menu identified by the Id argument.
	openSubMenuElements:
		adds menu elements to the SubMenu identified by the second Id argument,
		which must be contained in the Menu identified by the first Id argument.
	openRadioMenuItems:
		adds menu radio items to the RadioMenu identified by the second Id argument,
		which must be contained in the Menu identified by the first Id argument.
```

```
        If the RadioMenu was empty, then the first item in the list will be checked.
    Menu elements are added after the item with the specified index. The index of a
        menu element starts from one for the first menu element in the indicated
        menu.
        If the index is negative or zero, then the new menu elements are added
        before the first menu element of the indicated menu.
        If the index exceeds the number of menu elements in the indicated menu, then
        the new menu elements are added behind the last menu element of the
        indicated menu.
    No menu elements are added if the indicated menu does not exist.
    open(Sub)MenuElements have no effect in case menu elements with duplicate Ids
    are opened.
*/


closeMenuElements :: !Id ![Id] !(IOSt .l .p) -> IOSt .l .p
/*  closeMenuElements
        closes menu elements of the Menu identified by the first Id argument by
        their Ids. The elements of (Sub/Radio)Menus will be removed first.
*/


closeMenuIndexElements      :: !Id     ![Index] !(IOSt .l .p) -> IOSt .l .p
closeSubMenuIndexElements   :: !Id !Id ![Index] !(IOSt .l .p) -> IOSt .l .p
closeRadioMenuIndexElements :: !Id !Id ![Index] !(IOSt .l .p) -> IOSt .l .p
/*  Close menu elements of the indicated Menu, SubMenu, or RadioMenu by their Index
    position.
    closeMenuIndexElements:
        closes menu elements of the Menu identified by the Id argument.
    closeSubMenuIndexElements:
        closes menu elements of the SubMenu identified by the second Id argument,
        which must be contained in the Menu identified by the first Id argument.
    closeRadioMenuIndexElements:
        closes menu items of the RadioMenu identified by the second Id argument,
        which must be contained in the Menu identified by the first Id argument.
    Analogous to openMenuElements and openRadioMenuItems indices range from one to
        the number of menu elements in a menu. Invalid indices (less than one or
        larger than the number of menu elements of the menu) are ignored.
    If the currently checked element of a RadioMenu is closed, the first remaining
        element of that RadioMenu will be checked.
    Closing a (Sub/Radio)Menu closes the indicated (Sub/Radio)Menu and all of its
    elements.
*/


enableMenuSystem :: !(IOSt .l .p) -> IOSt .l .p
disableMenuSystem:: !(IOSt .l .p) -> IOSt .l .p
/*  Enable/disable the menu system of this interactive process. When the menu system
    is re-enabled the previously selectable menus and elements will become
    selectable again.
    Enable/disable operations on the menu(element)s of a disabled menu system take
    effect when the menu system is re-enabled.
    enableMenuSystem has no effect in case the interactive process has a (number of)
    modal dialogue(s).
*/


enableMenus :: ![Id] !(IOSt .l .p) -> IOSt .l .p
disableMenus:: ![Id] !(IOSt .l .p) -> IOSt .l .p
/*  Enable/disable individual menus.
    The WindowMenu can not be enabled/disabled.
    Disabling a menu overrules the SelectStates of its elements, which become
    unselectable.
    Enabling a disabled menu re-establishes the SelectStates of its elements.
    Enable/disable operations on the elements of a disabled menu take effect when
    the menu is re-enabled.
*/
```

```
getMenuSelectState :: !Id !(IOSt .l .p) -> (!Maybe SelectState,!IOSt .l .p)
/*  getMenuSelectState yields the current SelectState of the indicated menu. In case
    the menu does not exist, Nothing is returned.
*/


getMenus :: !(IOSt .l .p) -> (![(Id,MenuType)],!IOSt .l .p)
/*  getMenus yields the Ids and MenuTypes of the current set of menus of this
    interactive process.
*/


getMenuPos :: !Id !(IOSt .l .p) -> (!Maybe Index,!IOSt .l .p)
/*  getMenuPos yields the index position of the indicated menu in the current list
    of menus.
    In case the menu does not exist, Nothing is returned.
*/


setMenuTitle :: !Id !Title !(IOSt .l .p) -> IOSt .l .p
getMenuTitle :: !Id        !(IOSt .l .p) -> (!Maybe Title,!IOSt .l .p)
/*  setMenuTitle sets the title of the indicated menu.
        In case the menu does not exist or refers to the WindowMenu, nothing
        happens.
    getMenuTitle retrieves the current title of the indicated menu.
        In case the menu does not exist, Nothing is returned.
*/
```

A.15 StdMenuDef

```
definition module StdMenuDef


//  ****************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdMenu contains the types to define the standard set of menus and their
//  elements.
//  ****************************************************************************


import  StdIOCommon, StdMaybe


/*  Menus:              */
::  Menu        m ls ps = Menu          Title         (m ls ps)
                                        [MenuAttribute *(ls,ps)]

/*  Menu elements:      */
::  SubMenu     m ls ps = SubMenu       Title         (m ls ps)
                                        [MenuAttribute *(ls,ps)]
::  RadioMenu     ls ps = RadioMenu     [MenuRadioItem *(ls,ps)] Index
                                        [MenuAttribute *(ls,ps)]
::  MenuItem      ls ps = MenuItem      Title
                                        [MenuAttribute *(ls,ps)]
::  MenuSeparator ls ps = MenuSeparator [MenuAttribute *(ls,ps)]

::  MenuRadioItem ps  :== (Title,Maybe Id,Maybe Char,IOFunction ps)

::  MenuAttribute       ps                      // Default:
//  Attributes for Menus and MenuElements:
    =   MenuId              Id                  // no Id
    |   MenuSelectState     SelectState         // menu(item) Able
//  Attributes only for Menus:
    |   MenuIndex           Int                 // end of current menu list
//  Attributes ignored by (Sub)Menus:
    |   MenuShortKey        Char                // no ShortKey
    |   MenuMarkState       MarkState           // NoMark
    |   MenuFunction        (IOFunction     ps) // \x->x
    |   MenuModsFunction    (ModsIOFunction ps) // MenuFunction

::  MenuType        :== String
::  MenuElementType :== String
```

A.16 StdMenuElement

```
definition module StdMenuElement


//	********************************************************************************
//	Clean Standard Object I/O library, version 1.0.1
//	
//	StdMenuElement specifies all functions on menu elements.
//	Changing the status of menu elements requires a *MState.
//	Reading  the status of menu elements requires a  MState.
//	********************************************************************************


import	StdMenuDef
from	iostate import IOSt


::	MState


getMenu :: !Id !(IOSt .l .p) -> (!Maybe MState, !IOSt .l .p)
/*	getMenu returns a read-only MState for the indicated menu.
	In case the indicated menu does not exist Nothing is returned.
*/


setMenu :: !Id ![IdFun *MState] !(IOSt .l .p) -> IOSt .l .p
/*	Apply the menu element changing functions to the current state of the indicated
	menu.
	In case the indicated menu does not exist nothing happens.
*/


/*	Functions that change the state of menu elements.
	When applied to unknown Ids none of these functions have effect.
*/

enableMenuElements  :: ![Id] !*MState -> *MState
disableMenuElements :: ![Id] !*MState -> *MState
/*	(en/dis)ableMenuElements set the SelectState of the indicated menu elements.
	Disabling a (Sub/Radio)Menu overrules the SelectStates of its elements, which
		become unselectable.
	Enabling a disabled (Sub/Radio)Menu re-establishes the SelectStates of its
		elements.
	(En/Dis)able operations on the elements of a disabled (Sub/Radio)Menu take
		effect when the (Sub/Radio)Menu is re-enabled.
*/

setMenuElementTitles :: ![(Id,Title)] !*MState -> *MState
/*	setMenuElementTitles sets the titles of the indicated menu elements.
*/

markMenuItems   :: ![Id] !*MState -> *MState
unmarkMenuItems :: ![Id] !*MState -> *MState
/*	(un)markMenuItems sets the MarkState of the indicated MenuItems.
*/

selectRadioMenuItem      :: !Id !Id    !*MState -> *MState
selectRadioMenuIndexItem :: !Id !Index !*MState -> *MState
/*	selectRadioMenu(Index)Item
		selects the indicated MenuRadioItem of a RadioMenu, causing the mark of the
		previously marked MenuRadioItem to disappear.
	selectRadioMenuItem
		indicates the MenuRadioItem by the Id of its parent RadioMenu and its Id.
	selectRadioMenuIndexItem
		indicates the MenuRadioItem by the Id of its parent RadioMenu and its index
```

```
        position (counted from 1).
*/


/*  Functions that read the state of menu elements.
*/

getMenuElementTypes         ::     !MState -> [(MenuElementType,Maybe Id)]
getCompoundMenuElementTypes :: !Id !MState -> [(MenuElementType,Maybe Id)]
/*  getMenuElementTypes
        yields the list of MenuElementTypes of all menu elements of this menu.
    getCompoundMenuElementTypes
        yields the list of MenuElementTypes of all menu elements of this
        (Sub/Radio)Menu.
    Both functions return (Just id) if the element has a MenuId attribute, and
    Nothing otherwise.
    Ids are not collected recursively through (Sub/Radio)Menus.
*/

getSelectedRadioMenuItem    :: !Id   !MState -> (!Index,!Maybe Id)
/*  getSelectedRadioMenuItem
        returns the Index and Id, if any, of the currently selected MenuRadioItem of
        the indicated RadioMenu.
    If the RadioMenu does not exist or is empyt, the Index is zero and the Id is
    Nothing.
*/

getMenuElementSelectStates  :: ![Id] !MState -> [(Bool,SelectState)]    // Able
getMenuElementMarkStates    :: ![Id] !MState -> [(Bool,MarkState)]      // NoMark
getMenuElementTitles        :: ![Id] !MState -> [(Bool,Maybe String)]   // Nothing
getMenuElementShortKey      :: ![Id] !MState -> [(Bool,Maybe Char)]     // Nothing
/*  The  result list is of equal length as the argument Id list.
    Each result list element corresponds in order with the argument Id list.
    Of each element the first Boolean result is False in case of invalid id
        (if so dummy values are returned - see comment).
    -   getMenuElementSelectStates
            yield the SelectState of the indicated elements.
    -   getMenuElementMarkStates
            yield the MarkState of the indicated elements.
    -   getMenuElementTitles
            yields (Just title) of the indicated (SubMenu/MenuItem),
            Nothing otherwise.
    -   getMenuElementShortKey
            yields (Just key) of the indicated MenuItem, Nothing otherwise.
*/
```

A.17 StdMenuElementClass

```
definition module StdMenuElementClass


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdMenuElementClass defines the standard set of menu element instances.
//  ********************************************************************************


import  StdMenuDef
from    menuhandle  import MenuElementState


class MenuElements m
where
    menuElementToHandles    :: !(m .ls .ps) -> [MenuElementState .ls .ps]
    getMenuElementType      ::  (m .ls .ps) -> MenuElementType

instance MenuElements (AddLS    m)  | MenuElements m    // getMenuElementType==""
instance MenuElements (NewLS    m)  | MenuElements m    // getMenuElementType==""
instance MenuElements (ListLS   m)  | MenuElements m    // getMenuElementType==""
instance MenuElements NilLS                             // getMenuElementType==""
instance MenuElements ((:+:) m1 m2) | MenuElements m1
                                    & MenuElements m2   // getMenuElementType==""
instance MenuElements (SubMenu  m)  | MenuElements m
instance MenuElements RadioMenu
instance MenuElements MenuItem
instance MenuElements MenuSeparator
```

A.18 StdMenuReceiver

```
definition module StdMenuReceiver


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdMenuReceiver defines Receiver(2) menu element instances.
//  ********************************************************************************


import  StdReceiverDef, StdMenuElementClass


//  Receiver components:
instance MenuElements (Receiver  m  )
instance MenuElements (Receiver2 m r)
```

A.19 StdPicture

```
definition module StdPicture


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdPicture contains the drawing operations and access to Pictures.
//  ********************************************************************************


from    ospicture   import  Picture
import  StdPictureDef


//  Attribute functions.

getPicture              ::                       !*Picture -> (!Picture,!*Picture)
getPictureAttributes    ::                       ! Picture -> [PictureAttribute]
setPictureAttributes    :: ![PictureAttribute]  !*Picture -> *Picture

//  Pen position attributes:
setPenPos               :: !Point               !*Picture -> *Picture
getPenPos               ::                       ! Picture -> Point

class movePenPos figure :: !figure              !*Picture -> *Picture
//  Move the pen position as much as when drawing the figure.
instance movePenPos Vector
instance movePenPos Curve

//  PenSize attributes:
setPenSize              :: !Int                 !*Picture -> *Picture
getPenSize              ::                       ! Picture -> Int

setDefaultPenSize       ::                       !*Picture -> *Picture
//  setDefaultPenSize = setPenSize 1

//  Colour attributes:
setPenColour            :: !Colour              !*Picture -> *Picture
getPenColour            ::                       ! Picture -> Colour

setDefaultPenColour     ::                       !*Picture -> *Picture
//  setDefaultPenColour = setPenColour BlackColour

//  Font attributes:
setPenFont              :: !Font                !*Picture -> *Picture
getPenFont              ::                       ! Picture -> Font

setDefaultPenFont       ::                       !*Picture -> *Picture

/*  Picture is an environment instance of the FontEnv class.    */
instance FontEnv Picture


//  Drawing functions.

::  DrawFunction
    :== *Picture -> *Picture

drawPicture     :: ![DrawFunction] !*Picture -> *Picture
drawseqPicture  :: ![DrawFunction] !*Picture -> *Picture
xorPicture      :: ![DrawFunction] !*Picture -> *Picture
xorseqPicture   :: ![DrawFunction] !*Picture -> *Picture
/*  draw(seq)Picture
        applies the list of drawing functions to the argument picture in left to
        right order.
```

```
        After drawing, drawPicture restores the picture attributes of the resulting
        picture to those of the argument picture.
    xor(seq)Picture
        applies the list of drawing functions to the argument picture in left to
        right order in the appropriate platform xor mode.
        After drawing, xorPicture restores the picture attributes of the resulting
        picture to those of the argument picture.
*/


//  Drawing points:

drawPoint       ::               !*Picture -> *Picture
drawPointAt     :: !Point        !*Picture -> *Picture
/*  drawPoint
        plots a point at the current  pen position p and moves to p+{vx=1,vy=0}
    drawPointAt
        plots a point at the argument pen position, but retains the pen position.
*/

//  Drawing lines:

drawLineTo      :: !Point        !*Picture -> *Picture
drawLine        :: !Point !Point !*Picture -> *Picture
/*  drawLineTo
        draws a line from the current pen position to the argument point which
        becomes the new pen position.
    drawLine
        draws a line between the two argument points, but retains the pen position.
*/


//  Hiliting figures:

class Hilites figure where
    hilite  ::        !figure !*Picture -> *Picture
    hiliteAt:: !Point !figure !*Picture -> *Picture
/*  hilite
        draws figures in the appropriate 'hilite' mode at the current  pen position.
    hiliteAt
        draws figures in the appropriate 'hilite' mode at the argument pen position.
    Both functions reset the 'hilite' mode after drawing.
*/

instance Hilites Box        // Hilite a box
instance Hilites Rectangle  // Hilite a rectangle (note: hiliteAt pos r = hilite r)


//  Drawing within in a (list of) clipping area(s):

class Clips area where
    clip    ::        !area ![DrawFunction] !*Picture -> *Picture
    clipAt  :: !Point !area ![DrawFunction] !*Picture -> *Picture
/*  clip
        takes the base point of the clipping area to be the current  pen position.
    clipAt
        takes the base point of the clipping area to be the argument pen position.
*/

instance Clips Box                      // Clip within a box
instance Clips Rectangle                // Clip within a rectangle
instance Clips Polygon                  // Clip within a polygon
instance Clips [figure] | Clips figure  // Clip within the union of figures


/*  Drawing and filling operations.
    These functions are divided into the following classes:
```

```
    Drawables:  draw     'line-oriented' figures at the current  pen position.
                drawAt   'line-oriented' figures at the argument pen position.
    Fillables:  fill     'area-oriented' figures at the current  pen position.
                fillAt   'area-oriented' figures at the argument pen position.
*/
class Drawables figure where
    draw    ::        !figure !*Picture -> *Picture
    drawAt  :: !Point !figure !*Picture -> *Picture

class Fillables figure where
    fill    ::        !figure !*Picture -> *Picture
    fillAt  :: !Point !figure !*Picture -> *Picture


//  Text drawing operations:
//  Text is always drawn with the baseline at the y coordinate of the pen.

instance Drawables  Char
instance Drawables  {#Char}
/*  draw     text:
        draws the text starting at the current pen position.
        The new pen position is directly after the drawn text including spacing.
    drawAt p text:
        draws the text starting at p.
*/


//  Vector drawing operations:
instance Drawables Vector
/*  draw     v:
        draws a line from the current pen position pen to pen+v.
    drawAt p v:
        draws a line from p to p+v.
*/


/*  Oval drawing operations:
    An Oval o is a transformed unit circle
    with    horizontal radius rx    o.oval_rx
            vertical   radius ry    o.oval_ry
    Let (x,y) be a point on the unit circle:
        then (x`,y`) = (x*rx,y*ry) is a point on o.
    Let (x,y) be a point on o:
        then (x`,y`) = (x/rx,y/ry) is a point on the unit circle.
*/
instance Drawables Oval
instance Fillables Oval
/*  draw     o:
        draws an oval with the current pen position being the center of the oval.
    drawAt p o:
        draws an oval with p being the center of the oval.
    fill     o:
        fills an oval with the current pen position being the center of the oval.
    fillAt p o:
        fills an oval with p being the center of the oval.
    None of these functions change the pen position.
*/


/*  Curve drawing operations:
    A Curve c is a slice of an oval o
    with    start angle a   c.curve_from
            end   angle b   c.curve_to
            direction   d   c.curve_clockwise
    The angles are taken in radians (counter-clockwise).
    If d holds then the drawing direction is clockwise, otherwise drawing occurs
    counter-clockwise.
```

```
*/
instance Drawables Curve
instance Fillables Curve
/*  draw     c:
        draws a curve with the starting angle a at the current pen position.
        The pen position ends at ending angle b.
    drawAt p c:
        draws a curve with the starting angle a at p.
    fill     c:
        fills the figure obtained by connecting the endpoints of the drawn curve
        (draw c) with the center of the curve oval.
        The pen position ends at ending angle b.
    fillAt p c:
        fills the figure obtained by connecting the endpoints of the drawn curve
        (drawAt p c) with the center of the curve oval.
*/


/*  Box drawing operations:
    A Box b is a horizontally oriented rectangle
    with    width  w        b.box_w
            height h        b.box_h
    In case w==0 (h==0),   the Box collapses to a vertical (horizontal) vector.
    In case w==0 and h==0, the Box collapses to a point.
*/
instance Drawables  Box
instance Fillables  Box
/*  draw     b:
        draws a box with left-top corner at the current pen position p and
        right-bottom corner at p+(w,h).
    drawAt p b:
        draws a box with left-top corner at p and right-bottom corner at p+(w,h).
    fill     b:
        fills a box with left-top corner at the current pen position p and
        right-bottom corner at p+(w,h).
    fillAt p b:
        fills a box with left-top corner at p and right-bottom corner at p+(w,h).
    None of these functions change the pen position.
*/


/*  Rectangle drawing operations:
    A Rectangle r is always horizontally oriented
    with    width  w        abs (r.corner1.x-r.corner2.x)
            height h        abs (r.corner1.y-r.corner2.y)
    In case w==0 (h==0),   the Rectangle collapses to a vertical (horizontal) vector.
    In case w==0 and h==0, the Rectangle collapses to a point.
*/
instance Drawables Rectangle
instance Fillables Rectangle
/*  draw     r:
        draws a rectangle with diagonal corners r.corner1 and r.corner2.
    drawAt p r:
        draw r
    fill     r:
        fills a rectangle with diagonal corners r.corner1 and r.corner2.
    fillAt p r:
        fill r
    None of these functions change the pen position.
*/


/*  Polygon drawing operations:
    A Polygon p is a figure
    with    shape   p.polygon_shape
    A polygon p at a point base is drawn as follows:
        drawPicture [setPenPos base:map draw shape]++[drawToPoint base]
```

```
*/
instance Drawables Polygon
instance Fillables Polygon
/*  None of these functions change the pen position.
*/
```

A.20 StdPictureDef

```
definition module StdPictureDef


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdPictureDef contains the predefined figures that can be drawn.
//  ********************************************************************************

import  StdIOCommon, StdFont


::  Box                                     // A box is a rectangle
    =   {   box_w           :: !Int         // The width  of the box
        ,   box_h           :: !Int         // The height of the box
        }
::  Oval                                    // An oval is a stretched unit circle
    =   {   oval_rx         :: !Int         // The horizontal radius (stretch)
        ,   oval_ry         :: !Int         // The vertical   radius (stretch)
        }
::  Curve                                   // A curve is a slice of an oval
    =   {   curve_oval      :: !Oval        // The source oval
        ,   curve_from      :: !Real        // Starting angle (in radians)
        ,   curve_to        :: !Real        // Ending   angle (in radians)
        ,   curve_clockwise :: !Bool        // Direction: True iff clockwise
        }
::  Polygon                                 // A polygon is an outline shape
    =   {   polygon_shape   :: ![Vector]    // The shape of the polygon
        }


//  The picture attributes:
::  PictureAttribute                        // Default:
    =   PicturePenSize      Int             // 1
    |   PicturePenPos       Point           // zero
    |   PicturePenColour    Colour          // Black
    |   PicturePenFont      Font            // DefaultFont
::  Colour
    =   RGB RGBColour
    |   Black       | White
    |   DarkGrey    | Grey      | LightGrey // 75%, 50%, and 25% Black
    |   Red         | Green     | Blue
    |   Cyan        | Magenta   | Yellow
::  RGBColour
    =   {   r   :: !Int                     // The contribution of red
        ,   g   :: !Int                     // The contribution of green
        ,   b   :: !Int                     // The contribution of blue
        }
BlackRGB    :== {r=MinRGB,g=MinRGB,b=MinRGB}
WhiteRGB    :== {r=MaxRGB,g=MaxRGB,b=MaxRGB}
MinRGB      :== 0
MaxRGB      :== 255

PI          :== 3.1415926535898
```

A.21 StdProcess

```
definition module StdProcess


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdProcess contains the process creation and manipulation functions.
//  ********************************************************************************


import  StdProcessDef, StdWindow


/*  General process topology creation functions:
*/

class Processes pdef where
    startProcesses          :: !pdef      !*World      -> *World
    openProcesses           :: !pdef      !(PSt .l .p) -> PSt .l .p

class shareProcesses pdef   :: !(pdef .p) !(PSt .l .p) -> PSt .l .p

/*  (start/open/share)Processes creates an interactive process topology specified by
        the pdef argument.
        All interactive processes can communicate with each other by means of  the
        file system or by message passing.
        By default, processes obtain a private ProcessWindow. However, if a process
        has the ProcessShareGUI attribute, then the process will share the
        ProcessWindow of the current interactive process. Every interactive process
        can create processes in this way. This results in a tree of processes (see
        also the notes of termination at closeProcess).
    startProcesses aborts the application if the argument world does not contain a
        file system.
        startProcesses terminates as soon as all interactive processes that are
        created by startProcesses and their child processes have terminated. It
        returns the final world, consisting of the final file system and event
        stream.
    shareProcesses adds the interactive processes specified by the pdef argument to
        the process group of the current interactive process. The new interactive
        processes can communicate with all interactive processes of the current
        process group by means of the public process state component.
*/

instance Processes      (ProcessGroup pdef) | shareProcesses pdef
instance Processes      [pdef]              | Processes      pdef
instance Processes      (:^: pdef1 pdef2)   | Processes      pdef1
                                            & Processes      pdef2

instance shareProcesses  NDIProcess
instance shareProcesses (SDIProcess wdef)   | Windows        wdef
instance shareProcesses  MDIProcess
instance shareProcesses (ListCS    pdef )   | shareProcesses pdef
instance shareProcesses (:~: pdef1 pdef2)   | shareProcesses pdef1
                                            & shareProcesses pdef2


//  Convenience process creation functions:

startIO :: !.l !.p !(ProcessInit (PSt .l .p)) ![ProcessAttribute (PSt .l .p)]
           !*World -> *World
/*  startIO creates one process group of one interactive MDI process which is
    initialised with the ProcessInit argument.
*/
```

```
//  Process access operations:

closeProcess    :: !(PSt .l .p) -> PSt .l .p
/*  closeProcess removes all abstract devices that are held in the interactive
    process.
    If the interactive process has processes that share its GUI then these will also
    be closed recursively. As a result evaluation of this interactive process
    including GUI sharing processes will terminate.
*/


hideProcess     :: !(IOSt .l .p) -> IOSt .l .p
showProcess     :: !(IOSt .l .p) -> IOSt .l .p
/*  If the interactive process is active, hideProcess hides the interactive process,
    and showProcess makes it visible. Note that hiding an interactive process does
    NOT disable the process but simply makes it invisible.
*/

getProcessWindowPos :: !(IOSt .l .p) -> (!Point,!IOSt .l .p)
/*  getProcessWindowPos returns the current position of the ProcessWindow.
*/


getProcessWindowSize:: !(IOSt .l .p) -> (!Size,!IOSt .l .p)
/*  getProcessWindowSize returns the current size of the ProcessWindow.
*/
```

A.22 StdProcessDef

```
definition module StdProcessDef


//	********************************************************************************
//	Clean Standard Object I/O library, version 1.0.1
//	
//	StdProcessDef contains the types to define interactive processes.
//	********************************************************************************


import	StdIOCommon
from	iostate import PSt, IOSt


::	ProcessGroup pdef
	=	E..p:ProcessGroup p (pdef p)
::	NDIProcess      p                                   // DocumentInterface: NDI
	=	E..l:   NDIProcess  l
	                        (ProcessInit      (PSt l p))
	                        [ProcessAttribute (PSt l p)]
::	SDIProcess wdef p                                   // DocumentInterface: SDI
	=	E..l ls:SDIProcess  l ls
	                        (wdef          ls (PSt l p))
	                        (ProcessInit      (PSt l p))
	                        [ProcessAttribute (PSt l p)]
::	MDIProcess      p                                   // DocumentInterface: MDI
	=	E..l:   MDIProcess  l
	                        (ProcessInit      (PSt l p))
	                        [ProcessAttribute (PSt l p)]

/*	NDIProcesses can't open windows and menus.
	SDIProcesses can't open windows, except for their argument window.
	MDIProcesses can open an arbitrary number of device instances.
*/

::	ProcessInit ps
	:== [IdFun ps]
```

A.23 StdPSt

```
definition module StdPSt


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdPSt defines operations on PSt and IOSt that are not abstract device related.
//  ********************************************************************************


import  StdFunc, StdFile
import  StdFileSelect, StdFont, StdIOCommon, StdTime
from    iostate import  PSt, IOSt


/*  PSt is an environment instance of the following classes:
    - FileEnv       (see StdFile)
    - FileSelectEnv (see StdFileSelect)
    - FontEnv       (see StdFont)
    - TimeEnv       (see StdTime)
*/
instance FileEnv        (PSt .l .p)
instance FileSelectEnv  (PSt .l .p)
instance FontEnv        (PSt .l .p)
instance TimeEnv        (PSt .l .p)


beep :: !(IOSt .l .p) -> IOSt .l .p
/*  beep emits the alert sound.
*/


//  Operations on the global cursor:

/* RWS ---
setCursor       :: !CursorShape !(IOSt .l .p) -> IOSt .l .p
resetCursor     ::              !(IOSt .l .p) -> IOSt .l .p
obscureCursor   ::              !(IOSt .l .p) -> IOSt .l .p
/*  setCursor       overrules the shape of the cursor of all windows.
    resetCursor     removes the overruled cursor shape of all windows.
    obscureCursor   hides the cursor until the mouse is moved.
*/


//  Operations on the DoubleDownDistance:

setDoubleDownDistance :: !Int !(IOSt .l .p) -> IOSt .l .p
/*  setDoubleDownDistance sets the maximum distance the mouse is allowed to move to
    generate a ButtonDouble(Triple)Down button state. Negative values are set to
    zero.
*/
--- RWS */

//  Operations on the DocumentInterface of an interactive process:

getDocumentInterface :: !(IOSt .l .p) -> (!DocumentInterface, !IOSt .l .p)
/*  getDocumentInterface returns the DocumentInterface of the interactive process.
*/


//  Operations on the attributes of an interactive process:

setProcessActivate  :: !(IdFun (PSt .l .p)) !(IOSt .l .p) -> IOSt .l .p
setProcessDeactivate:: !(IdFun (PSt .l .p)) !(IOSt .l .p) -> IOSt .l .p
setProcessHelp      :: !(IdFun (PSt .l .p)) !(IOSt .l .p) -> IOSt .l .p
```

```
setProcessAbout     :: !(IdFun (PSt .l .p)) !(IOSt .l .p) -> IOSt .l .p
/*  These functions set the ProcessActivate, ProcessDeactivate, ProcessHelp, and
    ProcessAbout attribute of the interactive process respectively.
*/


//  Coercing PSt component operations to PSt operations.

appListPIO  :: ![.IdFun (IOSt .l .p)]   !(PSt .l .p) ->          PSt .l .p
appListPLoc :: ![.IdFun .l]             !(PSt .l .p) ->          PSt .l .p
appListPPub :: ![.IdFun .p]             !(PSt .l .p) ->          PSt .l .p

appPIO      :: !.(IdFun (IOSt .l .p))   !(PSt .l .p) ->          PSt .l .p
appPLoc     :: !.(IdFun .l)             !(PSt .l .p) ->          PSt .l .p
appPPub     :: !.(IdFun .p)             !(PSt .l .p) ->          PSt .l .p

//  Accessing PSt component operations.

accListPIO  :: ![.St (IOSt .l .p) .x]   !(PSt .l .p) -> (![.x], !PSt .l .p)
accListPLoc :: ![.St .l           .x]   !(PSt .l .p) -> (![.x], !PSt .l .p)
accListPPub :: ![.St .p           .x]   !(PSt .l .p) -> (![.x], !PSt .l .p)

accPIO      :: !.(St (IOSt .l .p) .x)   !(PSt .l .p) -> (!.x,   !PSt .l .p)
accPLoc     :: !.(St .l           .x)   !(PSt .l .p) -> (!.x,   !PSt .l .p)
accPPub     :: !.(St .p           .x)   !(PSt .l .p) -> (!.x,   !PSt .l .p)
```

A.24 StdReceiver

```
definition module StdReceiver


//	**************************************************************************************
//	Clean Standard Object I/O library, version 1.0.1
//
//	StdReceiver specifies all receiver operations.
//	**************************************************************************************


import	StdReceiverDef, StdMaybe
from	iostate	import	PSt, IOSt
from	id		import	RId, R2Id, RIdtoId, R2IdtoId, ==


//	Operations on the ReceiverDevice.

//	Open uni- and bi-directional receivers:

class Receivers rdef where
	openReceiver	:: .ls !(rdef .ls (PSt .l .p)) !(PSt .l .p)
					-> (!ErrorReport,!PSt .l .p)
	reopenReceiver	:: .ls !(rdef .ls (PSt .l .p)) !(PSt .l .p)
					-> (!ErrorReport,!PSt .l .p)
	getReceiverType ::	.(rdef .ls .ps)  -> ReceiverType
/*	openReceiver
		opens the given receiver if no receiver currently exists with the given
		R(2)Id. The R(2)Id has to be used to send messages to this receiver.
	reopenReceiver
		first closes the receiver with the same R(2)Id if present, and then opens a
		new receiver with the given receiver definition. The R(2)Id has to be used
		to send messages to the new receiver.
	getReceiverType
		returns the type of the receiver (see also getReceivers).
*/

instance Receivers (Receiver  msg)
instance Receivers (Receiver2 msg resp)


closeReceiver		:: !Id !(IOSt .l .p) -> IOSt .l .p
/*	closeReceiver closes the indicated uni- or bi-directional receiver.
	Invalid Ids have no effect.
*/


getReceivers		:: !(IOSt .l .p) -> (![(Id,ReceiverType)], !IOSt .l .p)
/*	getReceivers returns the Ids and ReceiverTypes of all currently open uni- or
	bi-directional receivers of this interactive process.
*/


enableReceivers		:: ![Id] !(IOSt .l .p) ->			IOSt .l .p
disableReceivers	:: ![Id] !(IOSt .l .p) ->			IOSt .l .p
getReceiverSelectState	:: ! Id  !(IOSt .l .p) -> (!Maybe SelectState,!IOSt .l .p)
/*	(en/dis)ableReceivers
		(en/dis)able the indicated uni- or bi-directional receivers.
		Note that this implies that in case of synchronous message passing messages
		can fail (see the comments of syncSend and syncSend2 below). Invalid Ids
		have no effect.
	getReceiverSelectState
		yields the current SelectState of the indicated receiver. In case the
		receiver does not exist, Nothing is returned.
*/
```

```
//  Inter-process communication:

//  Message passing status report:
::  SendReport
    =   SendOk
    |   SendUnknownProcess
    |   SendUnknownReceiver
    |   SendUnableReceiver
    |   SendDeadlock

instance toString SendReport

asyncSend :: !(RId msg) msg !(PSt .l .p) -> (!SendReport, !PSt .l .p)
/*  asyncSend posts a message to the receiver indicated by the argument RId. In case
    the indicated receiver belongs to this process, the message is simply buffered.
    asyncSend is asynchronous: the message will at some point be received by the
    indicated receiver.
    The SendReport can be one of the following alternatives:
    -   SendOk: No exceptional situation has occurred. The message has been sent.
                Note that even though the message has been sent, it cannot be
                guaranteed that the message will actually be handled by the
                indicated receiver because it might become closed, forever disabled,
                or flooded with synchronous messages.
    -   SendUnknownProcess:
                The indicated interactive process does not exist, therefore the
                receiver also does not exist.
    -   SendUnknownReceiver:
                The indicated receiver does not exist, although the interactive
                process that created it does exist.
    -   SendUnableReceiver:
                Does not occur: the message is always buffered, regardless whether
                the indicated receiver is Able or Unable. Note that in case the
                receiver never becomes Able, the message will not be handled.
    -   SendDeadlock:
                Does not occur.
*/

syncSend :: !(RId msg) msg !(PSt .l .p) -> (!SendReport, !PSt .l .p)
/*  syncSend posts a message to the receiver indicated by the argument RId. In case
    the indicated receiver belongs to the current process, the corresponding
    ReceiverFunction is applied directly to the message argument and current process
    state.
    syncSend is synchronous: this interactive process blocks evaluation until the
    indicated receiver has received the message.
    The SendReport can be one of the following alternatives:
    -   SendOk: No exceptional situation has occurred. The message has been sent and
                handled by the indicated receiver.
    -   SendUnknownProcess:
                The indicated interactive process does not exist, therefore the
                receiver also does not exist.
    -   SendUnknownReceiver:
                The indicated receiver does not exist, although the interactive
                process that created it does exist.
    -   SendUnableReceiver:
                The addressee exists, but its ReceiverSelect attribute is Unable.
                Message passing is halted. The message is not sent.
    -   SendDeadlock:
                The addressee is involved in a synchronous, cyclic communication
                with the current process. Blocking the current process would result
                in a deadlock situation. Message passing is halted to circumvent the
                deadlock. The message is not sent.
*/

syncSend2 :: !(R2Id msg resp) msg  !(PSt .l .p)
    -> (!(!SendReport,!Maybe resp), !PSt .l .p)
/*  syncSend2 posts a message to the receiver indicated by the argument R2Id. In
```

```
    case the indicated receiver belongs to the current process, the corresponding
    Receiver2Function is applied directly to the message argument and current
    process state.
    syncSend2 is synchronous: this interactive process blocks until the indicated
    receiver has received the message.
    The SendReport can be one of the following alternatives:
    -   SendOk: No exceptional situation has occurred. The message has been sent and
                handled by the indicated receiver. The response of the receiver is
                returned as well as (Just response).
    -   SendUnknownProcess:
                The indicated interactive process does not exist, therefore the
                receiver also does not exist.
    -   SendUnknownReceiver:
                The indicated receiver does not exist, although the interactive
                process that created it does exist.
    -   SendUnableReceiver:
                The addressee exists, but its ReceiverSelect attribute is Unable.
                Message passing is halted. The message is not sent.
    -   SendDeadlock:
                The addressee is involved in a synchronous, cyclic communication
                with the current process. Blocking the current process would result
                in a deadlock situation. Message passing is halted to circumvent the
                deadlock. The message is not sent.
    In all other cases than SendOk, the optional response is Nothing.
*/
```

A.25 StdReceiverDef

```
definition module StdReceiverDef


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdReceiverDef contains the types to define the standard set of receivers.
//  ********************************************************************************


import  StdIOCommon


::  Receiver  m   ls ps = Receiver  (RId  m)   (ReceiverFunction  m   *(ls,ps))
                                              [ReceiverAttribute     *(ls,ps)]
::  Receiver2 m r ls ps = Receiver2 (R2Id m r) (Receiver2Function m r *(ls,ps))
                                              [ReceiverAttribute     *(ls,ps)]

::  ReceiverFunction  m   ps :== m -> ps ->    ps
::  Receiver2Function m r ps :== m -> ps -> (r,ps)

::  ReceiverAttribute   ps                          // Default:
    =   ReceiverSelectState SelectState             // receiver Able
::  ReceiverType
    :== String
```

A.26 StdSystem

```
definition module StdSystem


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdSystem defines platform dependent constants and functions.
//  ********************************************************************************


import  StdIOCommon


//  System dependencies concerning the file system.

dirseparator    :: Char     // Separator between folder- and filenames in a pathname
homepath        :: !String -> String
applicationpath :: !String -> String
/*  dirseparator
        is the separator symbol used between folder- and filenames in a file path.
    homepath
        prefixes the 'home' directory file path to the given file name.
    applicationpath
        prefixes the 'application' directory file path to the given file name.
    Use these directories to store preference/options/help files of an application.
*/


//  System dependencies concerning the time resolution

ticksPerSecond  :: Int
/*  ticksPerSecond returns the maximum timer resolution per second.
*/


//  System dependencies concerning the screen resolution.

mmperinch       :== 25.4

hmm             :: !Real -> Int
vmm             :: !Real -> Int
hinch           :: !Real -> Int
vinch           :: !Real -> Int
/*  h(mm/inch) convert millimeters/inches into pixels, horizontally.
    v(mm/inch) convert millimeters/inches into pixels, vertically.
*/

maxScrollWindowSize :: Size
maxFixedWindowSize  :: Size
/*  maxScrollWindowSize
        yields the range at which scrollbars are inactive.
    maxFixedWindowSize
        yields the range at which a window still fits on the screen.
*/
```

A.27 StdTime

```
definition module StdTime


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdTime contains time related operations.
//  ********************************************************************************


::  Time
    =   {   hours   :: !Int     // hours        (0-23)
        ,   minutes :: !Int     // minutes      (0-59)
        ,   seconds :: !Int     // seconds      (0-59)
        }
::  Date
    =   {   year    :: !Int     // year
        ,   month   :: !Int     // month        (1-12)
        ,   day     :: !Int     // day          (1-31)
        ,   dayNr   :: !Int     // day of week  (1-7, Sunday=1, Saturday=7)
        }

wait                :: !Int .x -> .x

/*  wait n x suspends the evaluation of x modally for n ticks.
    If n<=0, then x is evaluated immediately.
*/

class TimeEnv env where
    getBlinkInterval:: !*env -> (!Int,  !*env)
    getCurrentTime  :: !*env -> (!Time, !*env)
    getCurrentDate  :: !*env -> (!Date, !*env)
/*  getBlinkInterval
        returns the time interval in ticks that should elapse between blinks of
        e.g. a cursor. This interval may be changed by the user while the
        interactive process is running!
    getCurrentTime
        returns the current Time.
    getCurrentDate
        returns the current Date.
*/

instance TimeEnv World
```

A.28 StdTimer

```
definition module StdTimer


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdTimer specifies all timer operations.
//  ********************************************************************************


import  StdTimerDef, StdTimerElementClass, StdMaybe
from    StdSystem   import ticksPerSecond
from    iostate     import PSt, IOSt


class Timers tdef where
    openTimer   :: .ls !(tdef .ls (PSt .l .p)) !(PSt .l .p)
                              -> (!ErrorReport,!PSt .l .p)
    getTimerType::      (tdef .ls .ps) -> TimerType
/*  Open a new timer.
    This function has no effect in case the interactive process already contains a
    timer with the same Id. In case TimerElements are opened with duplicate Ids, the
    timer will not be opened. Negative TimerIntervals are set to zero.
    In case the timer does not have an Id, it will obtain an Id which is fresh with
    respect to the current set of timers. The Id can be reused after closing this
    timer.
*/

instance Timers (Timer t)   | TimerElements t


closeTimer :: !Id !(IOSt .l .p) -> IOSt .l .p
/*  closeTimer closes the timer with the indicated Id.
*/


getTimers  ::     !(IOSt .l .p) -> ([(Id,TimerType)],!IOSt .l .p)
/*  getTimers returns the Ids and TimerTypes of all currently open timers.
*/


enableTimer         :: !Id !(IOSt .l .p) -> IOSt .l .p
disableTimer        :: !Id !(IOSt .l .p) -> IOSt .l .p
getTimerSelectState :: !Id !(IOSt .l .p) -> (!Maybe SelectState,!IOSt .l .p)
/*  (en/dis)ableTimer (en/dis)ables the indicated timer.
    getTimerSelectState yields the SelectState of the indicated timer. If the timer
    does not exist, then Nothing is yielded.
*/


setTimerInterval    :: !Id !TimerInterval   !(IOSt .l .p) -> IOSt .l .p
getTimerInterval    :: !Id                  !(IOSt .l .p)
                    -> (!Maybe TimerInterval,!IOSt .l .p)
/*  setTimerInterval
        sets the TimerInterval of the indicated timer.
        Negative TimerIntervals are set to zero.
    getTimerInterval
        yields the TimerInterval of the indicated timer.
        If the timer does not exist, then Nothing is yielded.
*/
```

A.29 StdTimerDef

```
definition module StdTimerDef


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdTimerDef contains the types to define the standard set of timers.
//  ********************************************************************************


import  StdIOCommon


::  Timer t ls ps   = Timer TimerInterval (t ls ps) [TimerAttribute *(ls,ps)]

::  TimerInterval
    :== Int

::  TimerAttribute  ps                              // Default:
    =   TimerId             Id                      // no Id
    |   TimerSelectState    SelectState             // timer Able
    |   TimerFunction       (TimerFunction ps)      // \_ x->x

::  TimerFunction   ps  :== NrOfIntervals->ps->ps
::  NrOfIntervals       :== Int

::  TimerType           :== String
::  TimerElementType    :== String
```

A.30 StdTimerElementclass

```
definition module StdTimerElementClass


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdTimerElementClass define the standard set of timer element instances.
//  ********************************************************************************


import  StdIOCommon, StdTimerDef
from    timerhandle import TimerElementState


class TimerElements t where
    timerElementToHandles   :: !(t .ls .ps) -> [TimerElementState .ls .ps]
    getTimerElementType     ::  (t .ls .ps) -> TimerElementType

instance TimerElements (NewLS   t)   | TimerElements t  // getTimerElementType==""
instance TimerElements (AddLS   t)   | TimerElements t  // getTimerElementType==""
instance TimerElements (ListLS  t)   | TimerElements t  // getTimerElementType==""
instance TimerElements NilLS                            // getTimerElementType==""
instance TimerElements ((:+:) t1 t2) | TimerElements t1
                                     & TimerElements t2 // getTimerElementType==""
```

A.31 StdTimerReceiver

```
definition module StdTimerReceiver


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdTimerReceiver defines Receiver(2) timer element instances.
//  ********************************************************************************


import  StdReceiverDef, StdTimerElementClass


//  Receiver components for timers:
instance TimerElements  (Receiver  m  )
instance TimerElements  (Receiver2 m r)
```

A.32 StdWindow

```
definition module StdWindow


//	********************************************************************************
//	Clean Standard Object I/O library, version 1.0.1
//
//	StdWindow defines functions on windows and dialogues.
//	********************************************************************************


import	StdMaybe, StdWindowDef
from	StdPSt		import PSt, IOSt
from	StdControlClass	import Controls, ControlState


//	Functions applied to non-existent windows or unknown ids have no effect.
class Windows wdef
where
	openWindow	:: .ls !(wdef .ls (PSt .l .p)) !(PSt .l .p)
					-> (!ErrorReport,!PSt .l .p)
	getWindowType	::	(wdef .ls .ps)	-> WindowType

class Dialogs wdef
where
	openDialog	:: .ls !(wdef .ls (PSt .l .p)) !(PSt .l .p)
					-> (!ErrorReport,!PSt .l .p)
	openModalDialog :: .ls !(wdef .ls (PSt .l .p)) !(PSt .l .p)
					-> (!ErrorReport,!PSt .l .p)
	getDialogType	::	(wdef .ls .ps)	-> WindowType

/*	open(Window/Dialog) opens the given window(dialog).
	If the Window(Dialog) has no WindowIndex attribute (see StdWindowDef), then the
	new window is opened frontmost.
	If the Window(Dialog) has a WindowIndex attribute, then the new window is
	opened behind the window indicated by the integer index:
		Index value 1 indicates the top-most window.
		Index value M indicates the bottom-most modal window, if there are M modal
			windows.
		Index value N indicates the bottom-most window, if there are N windows.
		If index<M, then the new window is added behind the bottom-most modal window
			(at index M).
		If index>N, then the new window is added behind the bottom-most window
			(at index N).
	openModalDialog always opens a window at the front-most position.
	openWindow may not be permitted to open a window depending on its
		DocumentInterface (see the comments at the ShareProcesses instances in
		module StdProcess).
	In case the window does not have an Id, it will obtain an Id which is fresh with
	respect to the current set of windows. The Id can be reused after closing this
	window.
	In case a window with the same Id is already open the window will not be opened.
	In case controls are opened with duplicate Ids, the window will not be opened.
	openModalDialog terminates when:
		the window has been closed (by means of closeWindow), or the process has
		been terminated (by means of closeProcess).
*/

instance Windows (Window c) | Controls c
instance Dialogs (Dialog c) | Controls c


closeWindow :: !Id !(PSt .l .p) -> PSt .l .p
/*	If the indicated window is not an inactive modal window, then closeWindow closes
	the window.
	In case the Id of the window was generated by open(Window/Dialog), it will
```

```
    become reusable for new windows/dialogues.
    In case of unknown Id, closeWindow does nothing.
*/

closeControls :: !Id [Id] !Bool !(IOSt .l .p) -> IOSt .l .p
/*  closeControls removes the indicated controls (second argument) from the
    indicated window (first argument) and recalculates the layout iff the Boolean
    argument is True.
*/


openControls        :: !Id     .ls (cdef .ls (PSt .l .p)) !(IOSt .l .p)
                                          -> (!ErrorReport,!IOSt .l .p)
                                          |  Controls cdef
openCompoundControls:: !Id !Id .ls (cdef .ls (PSt .l .p)) !(IOSt .l .p)
                                          -> (!ErrorReport,!IOSt .l .p)
                                          |  Controls cdef
/*  openControls
        adds the given controls argument to the indicated window.
    openCompoundControls
        adds the given controls argument to the indicated compound control.
    Both functions have no effect in case the indicated window/dialog/compound
    control could not be found (ErrorUnknownObject) or if controls are opened with
    duplicate Ids (ErrorIdsInUse).
*/


setControlPos :: !Id !Id !ItemPos   !(IOSt .l .p) -> (!Bool,!IOSt .l .p)
/*  setControlPos changes the current layout position of the indicated control to
    the new position.
    If there are relatively laynout controls, then their layout also changes. The
    window is not resized.
    The Boolean result is False iff the window or control id are unknown, or if the
    new ItemPos refers to an unknown control.
*/


controlSize :: !(cdef .ls (PSt .l .p))
                     !(Maybe (Int,Int)) !(Maybe (Int,Int)) !(Maybe (Int,Int))
                     !(IOSt .l .p)
            -> (!Size,!IOSt .l .p)
            |  Controls cdef
/*  controlSize calculates the size of the given control definition as it would be
    opened as an element of a window/dialog.
    The Maybe arguments are the prefered horizontal margins, vertical margins, and
    item spaces (see also the (Window/Control)(H/V)Margin and
    (Window/Control)ItemSpace attributes). If Nothing is specified, their default
    values are used.
*/


hideWindows     :: ![Id]    !(IOSt .l .p) -> IOSt .l .p
showWindows     :: ![Id]    !(IOSt .l .p) -> IOSt .l .p
getHiddenWindows::          !(IOSt .l .p) -> (![Id],!IOSt .l .p)
getShownWindows ::          !(IOSt .l .p) -> (![Id],!IOSt .l .p)
/*  (hide/show)Windows hides/shows the indicated modeless windows (modal dialogues
    can not be hidden).
    get(Hidden/Shown)Windows yields the list of currently visible/invisible windows.
*/


activateWindow  :: !Id      !(IOSt .l .p) -> IOSt .l .p
/*  activateWindow makes the indicated window the active window.
    If the window was hidden, then it will become shown.
    If there are modal dialogues, then the window will be placed behind the last
    modal dialog.
    activateWindow has no effect in case the window is unknown or is a modal dialog.
```

```
*/

getActiveWindow ::          !(IOSt .l .p) -> (!Maybe Id,!IOSt .l .p)
/*  getActiveWindow returns the Id of the window that currently has the input focus
    of the interactive process.
    Nothing is returned if there is no such window.
    Note that hidden windows never are active windows, and that modal windows never
    are hidden.
*/


stackWindow     :: !Id !Id  !(IOSt .l .p) -> IOSt .l .p
/*  stackWindow id1 id2 places the window with id1 behind the window with id2.
    If id1 or id2 is unknown, or id1 indicates a modal window, stackWindow does
    nothing.
    If id2 indicates a modal window, then the window with id1 is placed behind the
    last modal window.
*/

getWindowStack  :: !(IOSt .l .p) -> (![(Id,WindowType)],!IOSt .l .p)
getWindowsStack :: !(IOSt .l .p) -> (![Id],             !IOSt .l .p)
getDialogsStack :: !(IOSt .l .p) -> (![Id],             !IOSt .l .p)
/*  getWindowStack returns the Ids and WindowTypes of all currently open windows
    (including the hidden windows), in the current stacking order starting with the
    active window.
    get(Windows/Dialogs)Stack is equal to getWindowStack, restricted to Windows
    instances and Dialogs instances respectively.
*/


getDefaultHMargin   ::      !(IOSt .l .p) -> ((Int,Int),        !IOSt .l .p)
getDefaultVMargin   ::      !(IOSt .l .p) -> ((Int,Int),        !IOSt .l .p)
getDefaultItemSpace ::      !(IOSt .l .p) -> ((Int,Int),        !IOSt .l .p)
getWindowHMargin    :: !Id  !(IOSt .l .p) -> (!Maybe (Int,Int), !IOSt .l .p)
getWindowVMargin    :: !Id  !(IOSt .l .p) -> (!Maybe (Int,Int), !IOSt .l .p)
getWindowItemSpace  :: !Id  !(IOSt .l .p) -> (!Maybe (Int,Int), !IOSt .l .p)
/*  getDefault((H/V)Margin)/ItemSpace) return the default values for the horizontal
    and vertical window/dialogue margins and item spaces.
    getWindow((H/V)Margin/ItemSpace) return the current horizontal and vertical
    margins and item spaces of the indicated window. These will have the default
    values in case they are not specified.
    In case the window does not exist, Nothing is yielded.
*/


enableWindow            :: !Id  !(IOSt .l .p) -> IOSt .l .p
disableWindow           :: !Id  !(IOSt .l .p) -> IOSt .l .p
enableWindowMouse       :: !Id  !(IOSt .l .p) -> IOSt .l .p
disableWindowMouse      :: !Id  !(IOSt .l .p) -> IOSt .l .p
enableWindowKeyboard    :: !Id  !(IOSt .l .p) -> IOSt .l .p
disableWindowKeyboard   :: !Id  !(IOSt .l .p) -> IOSt .l .p
/*  (en/dis)ableWindow
        (en/dis)ables the indicated window.
    (en/dis)ableWindowMouse
        (en/dis)ables mouse handling of the indicated window.
    (en/dis)ableWindowKeyboard
        (en/dis)ables keyboard handling of the indicated window.
    Disabling a window overrules the SelectStates of its elements, which all become
    Unable.
    Reenabling the window reestablishes the SelectStates of its elements.
    The functions have no effect in case of invalid Ids or Dialogs instances.
    The latter four functions also have no effect in case the Window does not have
    the indicated attribute.
*/


getWindowSelectState        :: !Id  !(IOSt .l .p) ->(!Maybe SelectState,!IOSt .l .p)
```

```
getWindowMouseSelectState   :: !Id  !(IOSt .l .p) ->(!Maybe SelectState,!IOSt .l .p)
getWindowKeyboardSelectState:: !Id  !(IOSt .l .p) ->(!Maybe SelectState,!IOSt .l .p)
/*  getWindowSelectState
        yields the current SelectState of the indicated window.
    getWindow(Mouse/Keyboard)SelectState
        yields the current SelectState of the mouse/keyboard of the indicated
        window.
    The functions return Nothing in case of invalid Ids or Dialogs instances or if
    the Window does not have the indicated attribute.
*/


getWindowMouseStateFilter   :: !Id                          !(IOSt .l .p)
                            -> (!Maybe MouseStateFilter,    ! IOSt .l .p)
getWindowKeyboardStateFilter:: !Id                          !(IOSt .l .p)
                            -> (!Maybe KeyboardStateFilter, ! IOSt .l .p)
setWindowMouseStateFilter   :: !Id !MouseStateFilter        !(IOSt .l .p)
                                                           -> IOSt .l .p
setWindowKeyboardStateFilter:: !Id !KeyboardStateFilter     !(IOSt .l .p)
                                                           -> IOSt .l .p
/*  getWindow(Mouse/Keyboard)StateFilter yields the current
        (Mouse/Keyboard)StateFilter of the indicated window. Nothing is yielded in
        case the window does not exist or has no Window(Mouse/Keyboard) attribute.
    setWindow(Mouse/Keyboard)StateFilter replaces the current
        (Mouse/Keyboard)StateFilter of the indicated window. If the indicated window
        does not exist the function has no effect.
*/


drawInWindow    :: !Id ![DrawFunction]      !(IOSt .l .p) -> IOSt .l .p
/*  drawInWindow applies the list of drawing functions in left-to-right order to the
    picture of the indicated window (behind all controls).
    drawInWindow has no effect in case the window is unknown or is a Dialog.
*/

updateWindow    :: !Id !(Maybe ViewFrame)   !(IOSt .l .p) -> IOSt .l .p
/*  updateWindow applies the WindowLook attribute function of the indicated window.
    The SelectState argument of the Look attribute is the current SelectState of the
    window.
    The UpdateState argument of the Look attribute is
        {oldFrame=frame,newFrame=frame,updArea=[frame]}
    where frame depends on the optional ViewFrame argument:
        in case of (Just rectangle):
            the intersection of the current ViewFrame of the window and rectangle.
        in case of Nothing:
            the current ViewFrame of the window.
    updateWindow has no effect in case of unknown windows, or if the indicated
    window is a Dialog, or the window has no WindowLook attribute, or the optional
    viewframe argument is empty.
*/

setWindowLook   :: !Id !Bool !Look  !(IOSt .l .p) -> IOSt .l .p
getWindowLook   :: !Id              !(IOSt .l .p) -> (!Maybe Look,!IOSt .l .p)
/*  setWindowLook sets the Look of the indicated window.
        The window is redrawn only if the Boolean argument is True.
        setWindowLook has no effect in case the window does not exist, or is a
        Dialog.
    getWindowLook returns the (Just Look) of the indicated window.
        In case the window does not exist, or is a Dialog, or has no WindowLook
        attribute, the result is Nothing.
*/


setWindowPos    :: !Id !ItemPos !(IOSt .l .p) -> IOSt .l .p
getWindowPos    :: !Id          !(IOSt .l .p) -> (!Maybe ItemOffset,!IOSt .l .p)
/*  setWindowPos places the window at the indicated position.
        If the ItemPos argument refers to the Id of an unknown window (in case of
```

```
        LeftOf/RightTo/Above/Below), setWindowPos has no effect.
        If the ItemPos argument is one of (LeftOf/RightTo/Above/Below)Prev, then the
        previous window is the window that is before the window in the current
        stacking order.
        If the window is frontmost, setWindowPos has no effect. setWindowPos also
        has no effect if the window would be moved outside the screen, or if the Id
        is unknown or refers to a modal Dialog.
    getWindowPos returns the current item offset position of the indicated window.
        The corresponding ItemPos is (LeftTop,offset). Nothing is returned in case
        the window does not exist.
*/


moveWindowViewFrame :: !Id Vector !(IOSt .l .p) -> IOSt .l .p
/*  moveWindowViewFrame moves the orientation of the view frame of the indicated
    window over the given vector, and updates the window if necessary. The view
    frame is not moved outside the ViewDomain of the window.
    In case of unknown Id, or of Dialogs, moveWindowViewFrame has no effect.
*/

getWindowViewFrame  :: !Id !(IOSt .l .p) -> (!ViewFrame,!IOSt .l .p)
/*  getWindowViewFrame returns the current view frame of the window in terms of the
    ViewDomain. Note that in case of a Dialog, getWindowViewFrame returns
    {zero,size}.
    In case of unknown windows, the ViewFrame result is zero.
*/

setWindowViewSize   :: !Id Size !(IOSt .l .p) -> IOSt .l .p
getWindowViewSize   :: !Id      !(IOSt .l .p) -> (!Size,!IOSt .l .p)
/*  setWindowViewSize
        sets the size of the view frame of the indicated window as given, and
        updates the window if necessary. The size is fit between the minimum size
        and the screen dimensions.
        In case of unknown Ids, or of Dialogs, setWindowViewSize has no effect.
    getWindowViewSize yields the current size of the view frame of the indicated
        window. If the window does not exist, zero is returned.
*/

setWindowViewDomain :: !Id ViewDomain   !(IOSt .l .p) -> IOSt .l .p
getWindowViewDomain :: !Id              !(IOSt .l .p)
                   -> (!Maybe ViewDomain,!IOSt .l .p)
/*  setWindowViewDomain
        sets the view domain of the indicated window as given. The window view frame
        is moved such that a maximum portion of the view domain is visible. The
        window is not resized.
        In case of unknown Ids, or of Dialogs, setWindowViewDomain has no effect.
    getWindowViewDomain
        returns the current ViewDomain of the indicated window.
        Nothing is returned in case the window does not exist or is a Dialog.
*/


setWindowTitle  :: !Id Title       !(IOSt .l .p) -> IOSt .l .p
setWindowOk     :: !Id Id          !(IOSt .l .p) -> IOSt .l .p
setWindowCancel :: !Id Id          !(IOSt .l .p) -> IOSt .l .p
setWindowCursor :: !Id CursorShape !(IOSt .l .p) -> IOSt .l .p
getWindowTitle  :: !Id             !(IOSt .l .p) -> (!Maybe Title,      !IOSt .l .p)
getWindowOk     :: !Id             !(IOSt .l .p) -> (!Maybe Id,         !IOSt .l .p)
getWindowCancel :: !Id             !(IOSt .l .p) -> (!Maybe Id,         !IOSt .l .p)
getWindowCursor :: !Id             !(IOSt .l .p) -> (!Maybe CursorShape,!IOSt .l .p)
/*  setWindow(Title/Ok/Cancel/Cursor) set the indicated window attributes.
        In case of unknown Ids, these functions have no effect.
    getWindow(Title/Ok/Cancel/Cursor) get the indicated window attributes.
        In case of unknown Ids, the result is Nothing.
*/
```

A.33 StdWindowDef

```
definition module StdWindowDef


//  ****************************************************************************
//  Clean Standard Object I/O library, version 1.0.1
//
//  StdWindowDef contains the types to define the standard set of windows and
//  dialogues.
//  ****************************************************************************


import  StdControlDef


::  Window c ls ps = Window Title (c ls ps) [WindowAttribute *(ls,ps)]
::  Dialog c ls ps = Dialog Title (c ls ps) [WindowAttribute *(ls,ps)]

::  WindowAttribute ps                         // Default:
//  Attributes for Windows and Dialogs:
    =   WindowId            Id                 // system defined id
    |   WindowPos           ItemPos            // system dependent
    |   WindowIndex         Int                // open front-most
    |   WindowSize          Size               // screen size
    |   WindowHMargin       Int Int            // system dependent
    |   WindowVMargin       Int Int            // system dependent
    |   WindowItemSpace     Int Int            // system dependent
    |   WindowOk            Id                 // no default (Custom)ButtonControl
    |   WindowCancel        Id                 // no cancel  (Custom)ButtonControl
    |   WindowHide                             // initially visible
    |   WindowClose         (IOFunction ps)    // user can't close window
    |   WindowInit          [IdFun ps]         // no actions after opening window
//  Attributes for Windows only:
    |   WindowSelectState   SelectState        // Able
    |   WindowLook          Look               // show system dependent background
    |   WindowViewDomain    ViewDomain         // {zero,max range}
    |   WindowOrigin        Point              // left top of picture domain
    |   WindowHScroll       ScrollFunction     // no horizontal scrolling
    |   WindowVScroll       ScrollFunction     // no vertical   scrolling
    |   WindowMinimumSize   Size               // system dependent
    |   WindowResize                           // fixed size
    |   WindowActivate      (IOFunction ps)    // id
    |   WindowDeactivate    (IOFunction ps)    // id
    |   WindowMouse         MouseStateFilter    SelectState (MouseFunction    ps)
                                               // no mouse input
    |   WindowKeyboard      KeyboardStateFilter SelectState (KeyboardFunction ps)
                                               // no keyboard input
    |   WindowCursor        CursorShape        // no change of cursor
::  CursorShape
    =   StandardCursor
    |   BusyCursor
    |   IBeamCursor
    |   CrossCursor
    |   FatCrossCursor
    |   ArrowCursor
    |   HiddenCursor

::  WindowType
    :== String
```