Part II
Chapter 4
An Editor for Graphical Objects

In this chapter we will develop a tool for editing graphical objects. The set-up is two-fold. Firstly, the presented application elaborates on the window handling as shown in the previous chapter. Secondly, and even more important, the editor is an example of object-oriented programming in Clean using existential types.

4.1 User view of the graphical editor

From a user's point of view, the basic structure of the graphical editor is as follows. The editor distinguishes two modes: drawing and selecting. During drawing mode, the user can create basic figures like lines, rectangles and ellipses, depending on the drawing tool that has been selected. During selecting mode, compound figures can be build, or existing figures can be changed, for instance one can put a number of selected figures into a group, or change the position/shape of these. The actions in selecting mode are always performed on active or selected figures; the others (inactive) remain unchanged. Of course, it also possible to activate or to deactivate figures.

In selecting mode we distinguish menu driven actions (cut, copy, paste, group and ungroup) from the actions that are mouse driven (activate, deactivate, move and resize). All actions in drawing mode are mouse driven. These two modes more or less prescribe the appearance of the editor to the user. Obviously, there is a window in which drawing takes place, and a toolbar allowing the user to switch between the different drawing tools. Moreover, a menu bar is specified referring to the different menus for performing the menu-driven actions in selecting mode, and for quitting the program. Activating a figure is done by clicking on it. Such a selected figure can be recognised by its selection markers: little black squares place at the corners of its enclosing rectangle. If the user moves the mouse while holding down the mouse key, the position of a selected figure is changed. If the user clicks on the selection marker, the corresponding figure can be resized.

4.2 Structure of the program

We will develop the graphics editor stepwise, taking the simple figure representation from chapter 4.2 as a starting point. By using existentially quantified data structures one can create objects allowing an object oriented style of programming.

```
::  Drawable E.a =  {   state       :: a
                    ,   move        :: a -> Point -> a
```



```
                    ,   draw        :: a Picture -> Picture
                    }
```

A `Drawable` object is an existentially quantified record containing a `state` field (of arbitrary type) and two "methods" (functions on the state) `move` and `draw` for moving and displaying the object on the screen, respectively. For each basic graphical object (e.g. a line, a circle, *etcetera*) an auxiliary function is defined which creates a `Drawable` graphical object. For instance, a `Drawable` line can be created by applying the function `MakeLine`. The type `Line` and the drawing routine `MakeLine` are defined in the standard I/O library `StdPicture`.

```
MakeLine :: Line -> Drawable Void
MakeLine line
        =   { state = line
            , move  = \dist line -> line + (dist,dist)
            , draw  = DrawLine
            }
```

The program state of the editor consists of the editing mode, and a collection of figures split up into selected (or active) and not selected (inactive) ones. Switching between modes is regulated via the mode menu. Initially, the editor is in selecting mode which is indicated by the shape of the cursor and by the fact that none of the drawing tools are marked in the mode menu.

```
::  * ProgState =   {   mode        :: Mode
                    ,   active      :: [Drawable Void]
                    ,   inactive    :: [Drawable Void]
                    }

::  Mode        = SelectMode | DrawMode DrawTool

::  DrawTool    = LineTool | RectangleTool | OvalTool


Start :: * World    ->  * World
Start world = CloseEvents events` world`
where
    (s, events`)        =   StartIO [menu, window] InitState [] events
    (events, world`)    =   OpenEvents world

    menu    =   MenuSystem [file, edit, arrange, mode]

    file    =   PullDownMenu FileId "File" Able
                [   MenuItem  QuitId "Quit" (Key 'Q') Able Quit]

    edit    =   PullDownMenu EditId "Edit" Able
                [   MenuItem  CutId "Cut" (Key 'X') Able Cut,
                    MenuItem  CopyId "Copy" (Key 'C') Able Copy,
                    MenuItem  PasteId "Paste" (Key 'V') Able Paste]

    arrange =   PullDownMenu ArrangeId "Arrange" Able
                [   MenuItem  GroupId "Group" (Key 'G') Able DoGroup,
                    MenuItem  UngroupId "Ungroup" (Key 'U') Able DoUngroup]
    mode    =   PullDownMenu ModeId "Mode" Able
                [   MenuItemGroup ModeGroupId tools_menu ]

    window  =   WindowSystem [ ScrollWindow 1 (0,0) "Picture"
                    (ScrollBar (Thumb 0) (Scroll 10))
                    (ScrollBar (Thumb 0) (Scroll 10))
                    PictDomain MinWinSize MaxWinSize Update
                    [Mouse Able MouseWaits, GoAway Quit]]

    tools_menu  =   [ CheckMenuItem LineToolId "Line" NoKey Able
                            NoMark (AdjustMode (DrawMode LineTool))
                    , CheckMenuItem RectangleToolId "Rectangle" NoKey Able
                            NoMark (AdjustMode (DrawMode RectangleTool))
```

```
                          , CheckMenuItem OvalToolId "Oval" NoKey Able
                                  NoMark (AdjustMode (DrawMode OvalTool))]
```

The auxiliary function `AdjustMode` changes the program state as well as the mode menu in order to indicate the new current mode. As you can see, the drawing mode is indicated by a cross cursor, a marked drawing tool, and moreover, by a (newly added) selection menu item in the mode menu for switching back to the selecting mode.

```
AdjustMode :: Mode ProgState IO -> (ProgState, IO)
AdjustMode  new_mode=:(DrawMode new_tool) state=:{mode = DrawMode old_tool} io
    | new_tool_id == old_tool_id
        = (state, io)
    | otherwise
        = ( { state & mode = new_mode },
            UnmarkMenuItems [old_tool_id] (MarkMenuItems [new_tool_id] io))
where
    old_tool_id = ToolToId old_tool
    new_tool_id = ToolToId new_tool
AdjustMode  new_mode=:(DrawMode tool) state=:{mode = SelectMode} io
    = ( { state & mode = new_mode },
        ChangeActiveWindowCursor CrossCursor
            (MarkMenuItems [ToolToId tool] append_io))
where
    append_io = AppendMenuItems ModeGroupId OvalToolId
                    [MenuItem SelectId "Select" NoKey Able
                        (AdjustMode SelectMode) ] io
AdjustMode  SelectMode state=:{mode = DrawMode tool} io
    = ( { state & mode = SelectMode },
        ChangeActiveWindowCursor StandardCursor
            (UnmarkMenuItems [ToolToId tool] (RemoveMenuItems [SelectId] io)))
```

The rest of the basic IO-system is self-explanatory.

4.3 Mouse Handling

The handling of mouse events is relative complex in a graphical editor because the mouse plays a key role both for making new drawings and for the selection and changing of existing graphical objects. The actions to perform depend on the mode in which the program is and there are many possibilities. We now concentrate on the way mouse events are processed in order to ensure that the behaviour of the program indeed corresponds to the intention of the user.

The initial mouse handling is handled by the function `MouseWaits`. When the mouse button is not pressed, the mouse is in its rest state and nothing happens. When a mouse button is pressed, the function `ActivateMouse` is called. The action that has to be performed depends on the current mode of the editor. The function `ActivateMouse` instals for each possible situation a new the mouse handling function which will deal with subsequential mouse clicks.

```
MouseWaits :: MouseState ProgState IO -> (ProgState, IO)
MouseWaits (pos, ButtonDown, (shift,_,_,_)) state io
    = ActivateMouse pos shift state io
MouseWaits mouse state io =  (state, io)

ActivateMouse :: MousePosition ProgState IO -> (ProgState, IO)
ActivateMouse pos state=:{ mode = DrawMode tool, inactive, active } io
    = ( { state & active = [], inactive = inactive ++ active },
        ChangeActiveMouseFunction (MouseDraws tool (pos, pos))
            (DrawInActiveWindow [ SetPenMode XorMode
                                , Select active
                                ] io))
ActivateMouse pos state=:{ mode = SelectMode, inactive, active } io
    | have_to_resize
        = ( state,
            ChangeActiveMouseFunction (MouseResizes resize_rect pos)
```



```
                (DrawInActiveWindow [ SetPenMode XorMode
                                    , Select active
                                    , SetPenMode OrMode
                                    , DrawBoundingBox resize_rect
                                    , SetPenMode XorMode
                                    ] io))
    | Contains active pos
        = ( state,
            ChangeActiveMouseFunction (MouseWaitsForMoving pos active) io)
    | isEmpty selected
        = ( { state & active = [], inactive = inactive ++ active},
              ChangeActiveMouseFunction (MouseSelects (pos, pos))
                (DrawInActiveWindow [ SetPenMode XorMode
                                    , Select active
                                    , SetPenMode CopyMode
                                    ] io))
    | otherwise
        = ({ state & active = selected, inactive = not_selected ++ active },
             ChangeActiveMouseFunction (MouseWaitsForMoving pos selected)
                (DrawInActiveWindow [ SetPenMode XorMode
                                    , Select (selected ++ active)
                                    , SetPenMode CopyMode
                                    ] io))
where
    (selected, not_selected) = RetrieveSelectedFigures pos inactive
    (have_to_resize, resize_rect) = ResizeAreaIsSelected pos active
```

In drawing mode, the position at which the mouse was pressed determines the starting point of a new figure. The end point is indicated by the moment at which the mouse button is released. In the meanwhile, the new figure in progress is drawn temporarily, to make its new shape visible. As usual, temporary figures are drawn with a pen in `XorMode` making the figure visible independent of its background. All these drawing actions are handled by the function `MouseDraws`. After a new figure has been finished, the mouse re-enters its rest state. The final figure is stored as an active drawable in the program state.

```
MouseDraws :: DrawTool Rectangle MouseState ProgState IO -> (ProgState, IO)
MouseDraws tool rect=:(top_left,bot_right) (pos, ButtonUp, _) state io
    = ({ state & active = [new_fig] },
         ChangeActiveMouseFunction MouseWaits
            (DrawInActiveWindow draw_figure io))
where
    new_fig     = MakeFigure tool (top_left, pos)
    draw_figure = [ DrawFigure tool rect
                  , SetPenMode CopyMode
                  , Draw [new_fig]
                  , SetPenMode XorMode
                  , Select [new_fig]
                  , SetPenMode CopyMode
                  ]
MouseDraws tool rect=:(top_left, bot_right) (pos, _ , _) state io
    | bot_right == pos
        = (state, io)
    | otherwise
        = (state,
           ChangeActiveMouseFunction (MouseDraws tool new_rect)
                (DrawInActiveWindow draw_figure io))
where
    new_rect = (top_left, pos)
    draw_figure = [DrawFigure tool rect, DrawFigure tool new_rect]
```

Mouse handling during selecting mode is much more complicated because there are several different situations that have to be dealt with. If the mouse is activated, by pressing its key, the following three cases are distinguished.

1) The user clicks on one of the selected figures. Then again there are several possibilities. If the user hits a selection marker and exactly one figure is active, that figure is going to be resized, taken care of by the function `MouseResizes`.

```
MouseResizes :: Rectangle  Point MouseState  ProgState IO -> (ProgState, IO)
MouseResizes resize_rect=:(top_left,bot_right) orig_bot_right
        (pos, ButtonUp, _ ) state=:{active, inactive} io
    = ( { state & active = new_figs },
          ChangeActiveMouseFunction MouseWaits
                (DrawInActiveWindow draw_figure io))
where
    new_figs    = Resize active top_left
                       (DetermineMultiplicationFactor top_left orig_bot_right pos)
    draw_figure = [ DrawResizeRectangle top_left bot_right orig_bot_right
                  , SetPenMode CopyMode
                  , ClearSelectedFigures active inactive
                  , Draw new_figs
                  , SetPenMode XorMode
                  , Select new_figs
                  , SetPenMode CopyMode
                  ]
MouseResizes resize_rect=:(top_left,bot_right) orig_bot_right
            (pos, _, _) state io
    | bot_right == pos
        = (state, io)
    | otherwise
        = ( state,
            ChangeActiveMouseFunction
                (MouseResizes new_resize_rect orig_bot_right)
                (DrawInActiveWindow draw_figure io))
    where
        new_resize_rect = (top_left, pos)
        draw_figure = if (orig_bot_right == pos)
                          [DrawResizeRectangle top_left pos orig_bot_right]
                          [DrawResizeRectangle top_left bot_right orig_bot_right
                          ,DrawResizeRectangle top_left pos orig_bot_right]
```

Otherwise, all selected figures are moved which is handled by the function `MouseMoves`.

```
MouseMoves :: Point [Drawable Void] MouseState ProgState IO -> (ProgState, IO)
MouseMoves prev_pos selected (pos, ButtonUp, _) state=:{active, inactive} io
    = ( { state & active = new_figs },
        ChangeActiveMouseFunction MouseWaits
            (DrawInActiveWindow draw_figures io))
    where
        new_figs        = Move selected (pos - prev_pos)
        draw_figures    = [ SetPenMode XorMode
                          , Draw selected
                          , SetPenMode CopyMode
                          , ClearSelectedFigures active inactive
                          , Draw new_figs
                          , SetPenMode XorMode
                          , Select new_figs
                          , SetPenMode CopyMode
                          ]
MouseMoves prev_pos selected (pos, _, _) state io
    | prev_pos == pos
        = (state, io)
    | otherwise
        = (state, ChangeActiveMouseFunction (MouseMoves pos moved_figures)
                        (DrawInActiveWindow [SetPenMode XorMode,
                            Draw (selected ++ moved_figures),
                            SetPenMode CopyMode] io))
    where
        moved_figures = Move selected (pos - prev_pos)
```

2) The user clicks on one or more inactive figures. Then these newly indicated figures become active (and the previously active ones become inactive) and again the mouse enters its moving state.

3) Otherwise, the mouse enters the global selection mode (handled by `MouseSelects`) in which several figures can be selected at the same time by capturing these in the indicated enclosing rectangle).

```
MouseSelects :: Rectangle MouseState ProgState IO -> (ProgState, IO)
MouseSelects  rect=:(top_left,bot_right) (pos, ButtonUp, _)
                state=:{active,inactive} io
    = ({ state & active = sel ++ active, inactive = not_sel },
         ChangeActiveMouseFunction MouseWaits
               (DrawInActiveWindow draw_figure io))
where
    (sel, not_sel) = RetrieveEnclosedFigures (top_left, pos)inactive
    draw_figure = [ SetPenMode XorMode
                  , DrawRectangle rect
                  , Select selected
                  , SetPenMode CopyMode
                  ]
MouseSelects rect=:(top_left, bot_right) (pos, buttondown, _) state io
    | bot_right == pos
        = (state, io)
    | otherwise
        = (state, ChangeActiveMouseFunction (MouseSelects new_rect)
                (DrawInActiveWindow draw_tmp_rectangle io))
where
    new_rect            = (top_left, pos)
    draw_tmp_rectangle  = [ SetPenMode XorMode
                          , DrawRectangle rect
                          , DrawRectangle new_rect
                          ]
```

To be able to perform all these actions (selecting, moving and resizing) correctly, the interface of Drawable objects is extended. This is done as follows.

```
::  Drawable E.a =  {   state       :: a
                    ,   move        :: a -> Point -> a
                    ,   resize      :: a Point (Real,Real) -> a
                    ,   draw        :: a Picture -> Picture
                    ,   bounds      :: a -> Rectangle
                    ,   contains    :: a Point -> Bool
                    }
```

The function `resize` is used to change the figure's shape. The result of this function is a two-dimensional multiplication with respect to the indicated point. The function `bounds` returns the smallest enclosing rectangle of the figure. Lastly, the function `contains` yields whether or not a given point lies on the figure. For the basic line figure, these extension have the following consequences.

```
MakeLine :: Line -> Drawable Void
MakeLine line
    =   {   state       = line
        ,   draw        = DrawLine
        ,   move        = \line dist -> line + (dist, dist)
        ,   resize      = ResizeRectangle
        ,   bounds      = \s -> normalize s
        ,   contains    = on_line
        }
    where
        on_line line=:((x1,y1),(x2,y2)) (x3,y3)
            = InRectangle (x3,y3) (tl_bound - (LineMargin,LineMargin),
                                   br_bound + (LineMargin,LineMargin)) &&
             abs (y_diff * (x3 - x1) - x_diff * (y3 - y1)) <=
                    max (abs (x_diff * LineMargin)) (abs (y_diff * LineMargin))
        where
            (tl_bound, br_bound) = normalize line

            x_diff = x2 - x1
            y_diff = y2 - y1

    normalize ((x1,y1),(x2,y2)) = ((min x1 x2,min y1 y2),(max x1 x2,max y1 y2))
```

```
InRectangle p (tl, br) = tl leq p && p leq br
```

The definition of the `contains` function is a bit subtle. We do not require that the user has to click exactly on a drawing he wants to indicate, but instead we have introduced a "clicking tolerance" determined by the constant `LineMargin`.

For convenience, we define lifted versions working on a list of drawables for all methods in the interface of an drawable object.

```
Move :: [Drawable Void]  Point -> [Drawable Void]
Move draws dist = map (\draw=:{move,state} ->
                        {draw & state = move state dist}) draws

Resize :: [Drawable Void]  Point  (Real,Real) -> [Drawable Void]
Resize draws point fact
    =  map (\draw=:{resize,state} ->
              {draw & state = resize state point fact}) draws

Draw :: [Drawable Void] Picture -> Picture
Draw drawables pict =  foldl (\p {draw,state} -> draw state p) pict drawables

Bounds :: [Drawable Void] -> Rectangle
Bounds drawables = foldl combine_bounds bound rest_bounds
where
    combine_bounds ((r1tlx,r1tly),(r1brx,r1bry)) ((r2tlx,r2tly),(r2brx,r2bry))
        = ((min r1tlx r2tlx,min r1tly r2tly),(max r1brx r2brx,max r1bry r2bry))

    [bound:rest_bounds] = map (\{bounds,state} ->  bounds state) drawables

Contains :: [Drawable Void] Point -> Bool
Contains drawables point = foldr ((||) o \{contains,state} ->
                                  contains state point) False drawables
```

Here again the elegance of our figure representation becomes apparent: the lifted methods be can expressed almost entirely as one-liners in terms of predefined list functions, like map and foldr.

4.4 Grouping

A collection of selected figures can be combined to form new group. The implementation of the grouping command is very easy and can be expressed in few lines of Clean. As you can see below, the only subtlety is the adjustment of the selection markers.

```
DoGroup state=:{active,inactive} io
    = ({ state & active = group }, DrawInActiveWindow draw_group io)
where
    draw_group = [ SetPenMode XorMode
                 , Select active
                 , Select group
                 , SetPenMode CopyMode
                 ]
    group = [MakeGroup active]
```

Ungrouping, however, takes some more doing since one cannot determine `from the outside' whether an object forms a group or not (for, its internal state is invisible). Therefore the task of ungrouping is delegated to the object itself, by adding an ungroup method to its interface, yielding the list of drawables with which the object has been created. By convention, this list is empty for basic objects. The instances of compound for basic as well as for compound figures is very simple.

```
::  Drawable E.a =  {   state       :: a
                    ,   ...
                    ,   ungroup     :: a -> [Drawable Void]
                    ,   ...
                    }
```



```
Ungroup :: [Drawable Void] -> [Drawable Void]
Ungroup [] = []
Ungroup [drawable=:{ungroup,state} : drawables]
    = case ungroup state of
        []      -> [drawable : Ungroup drawables]
        list    -> list ++  Ungroup drawables
```

4.5 Editing

The common clipboard actions, cut, copy and paste are implemented straightforwardly. We first add a clipboard representation to our program state.

```
::  * ProgState =   {   mode        :: Mode
                    ,   active      :: [Drawable Void]
                    ,   inactive    :: [Drawable Void]
                    ,   clipboard   :: [Drawable Void]
                    }
```

The handlers corresponding to the cupt, copy and past menu items are defined as follows.

```
Copy  state=:{active} io =  ({ state & clipboard = active (20,20)}, io)

Cut state=:{active,inactive} io
    =  ({ state & clipboard = Move active (20,20), active = []},
        DrawInActiveWindow [ClearSelectedFigures active inactive] io)

Paste state=:{clipboard, active, inactive} io
    = ({ state & active = move_clip, inactive = not_sel}, draw_window)
where
    not_sel      = active ++ inactive
    move_clip    = Move clipboard (20,20)
    draw_window  = DrawInActiveWindow [ ClearSelectedFigures active not_sel
                                      , Draw move_clip
                                      , SetPenMode XorMode
                                      , Select move_clip
                                      , SetPenMode CopyMode
                                      ] io
```

The functions `Copy` and `Cut` both copy all active figures to the clipboard. In constrast to `Copy`, `Cut` also removes the original figures from the program state. The function `Paste` copies the contents of the clipboard. These copied fugure immediately become active, while the old active figures are deactivated. To avoid that new figures coincide with existing ones, the former are moved slightly (to `(20,20)`). The auxiliary function `ClearSelectedFigures` is called to erase the selection markers of the figures that are deactivated.

4.6 Exercises

To be written.