Part II
Chapter 1
A Simple Database

In this chapter we describe how a simple database application can be programmed in Clean. The database being maintained can be seen as a pile of cards, e.g. of a Rolex system. All cards in the database system are assumed to contain the same kind of information. This application is interesting since the kind of information handled can be changed dynamically. The information can be stored in and retrieved from a file.

This Chapter not only shows how such a database program can be written in Clean, the application is also a good demonstration of how dynamically changing dialogues and windows can be defined in a straightforward manner. The kind of dynamic behaviour offered in Clean provides much more flexibility than the static dialogues generated by many special purpose (visual) dialogue editors.

A more complex database system (a relational database) which can have different kinds of cards stored in different files is described in the next chapter.

1.1 The database program state

The program state of the database application contains an administration (a record of type `TableAdm`) in which all relevant information of the database is remembered.

```
:: *IO              :== IOState SimpleDataBase
:: *SimpleDataBase :== (TableAdm, Files)

:: TableAdm    =   {   name         :: TableName
                   ,   descriptor   :: Descriptor
                   ,   records      :: Table
                   ,   selection    :: Int
                   ,   query        :: Record
                   ,   editinfoid   :: DialogItemId
                   ,   fw           :: Int                // field width
                   ,   dw           :: Int                // descriptor width
                   }

:: TableName   :== String
```

The field `name` in the table administration record refers to the name of the database which is being manipulated by the program. Instead of one record containing the entire program state we use a tuple containing the state of the database and the file system. The

distinction between the file system and the database state appears to be convenient in this program. There is no fundamental objection the use of a unified program state record.

All cards contain the same kind of information, a collection of *attribute values*. Each attribute value has a certain type. An *attribute name* (of type `String`) is used to identify an attribute value of a card. In the `descriptor` in the table administration the attribute names and the type of the corresponding attribute values are stored. For the time being it is assumed that an attribute value can only be of type `String`, indicated by the constructor `STRING`.

```
:: Descriptor      :== [(AttributeName,TypeCode)]
:: AttributeName   :== String
:: TypeCode        =   STRING
```

The attribute values of one card are by tradition called a *record*. In this case they are not stored in a Clean record but in a Clean list. This makes it possible to change the number and the order of the attributes dynamically (see 1.7). Each attribute value is preceded by a constructor indicating the type of the attribute value (e.g. `AS` to indicate a value of type String). The order in which the attribute values are stored in the list must correspond with the order in which the attribute names and types are described in the descriptor. All records together are called the *table* and they are stored in the table administration as well, again in a list.

```
:: Table           :== [ Record ]
:: Record          :== [ AttributeValue ]
:: AttributeValue  =   AS String
```

The cards (records) in the database are numbered, the first card has the number zero. The contents of the database can be displayed in the *database window*, the card being selected is highlighted (see 1.4). With help of the *edit dialogue* one can change the card under selection (see 1.5). One can also search for a specific record. The card to search for is specified in a query (see 1.6). A query is defined by filling in the known information of the card to look for. A query is therefore also of type `Record`. When a card is found it will be selected and high-lighted in the database window.

The number of the selected card is stored in the table administration (field `selection`).

In the table administration it is furthermore indicated whether an edit window or query window is chosen (`editinfoid`) and also some layout information (`fx` and `dx`) is stored in it.

1.2 At the start and the end of the application

The database application starts and stops with the usual actions.

```
Start :: *World -> *World
Start world = seq [ CloseEvents finalevents, closefiles finalfiles] world2
where
    (events,world1)             = OpenEvents world
    (files,world2)              = openfiles world1
    ((_,finalfiles),finalevents)
            =   StartIO devicesystems (initTableAdm,files) initIO events
```

The table administration is initialized as follows:

```
initTableAdm = {   records     =   []
               ,   descriptor  =   []
               ,   selection   =   0
               ,   query       =   []
               ,   name        =   ""
               ,   editinfoid  =   0
               ,   fw          =   0
               ,   dw          =   0
               }
```

In this application we only need a menu device which is defined as described below (the appearance of the corresponding menu on the screen is shown in figure 1.1).

```
devicesystems   = [ MenuSystem [menu] ]
menu =
    PullDownMenu DontCareId "Commands" Able
    [ MenuItem DontCareId "Show Table"  (Key 't') Able ShowTable,
    , MenuItem DontCareId "Edit..."     (Key 'e') Able ShowEditDialog,
    , MenuItem DontCareId "Change Set Up..." (Key 'u') Able ShowAttributeDialog,
    , MenuSeparator,
    , MenuItem DontCareId "Open new..." (Key 'o') Able (warn warning restart),
    , MenuItem DontCareId "Save As..."  (Key 's') Able SaveTable,
    , MenuItem DontCareId "Quit"        (Key 'q') Able (warn warning Quit)
    ]

DontCareId :== 0
```

Figure 1.1 How the menu looks on the screen (Macintosh platform).

The interesting aspects in this definition are of course the call-back functions. The call-back function `ShowTable` shows the attribute names and values in a scrollable window (see section 1.4), the function `ShowEditDialog` displays a dialogue for changing the selected card (see section 1.5) or for specifying a query (see section 1.6), while the function `ShowAttributeDialog` is displaying a dialogue for changing the attribute names (see section 1.7). The call-back function `SaveTable` saves the table in the file (see section 1.3). The call-back function `warn` is a handy function which is used to display a `warning` on the screen (see also figure 1.2).

```
warn message func s io
| choiceId == cancelId = (s,nio)              // do nothing
| otherwise            = func s nio           // OK: apply call-back function
where
    (choiceId,nio)      = OpenNotice warningdef io
    warningdef          = Notice message
                            (NoticeButton cancelId "Cancel") // default button
                               [NoticeButton okId "OK"]      // other buttons
    [cancelId,okId:_]   = [0..]

warning         = [ "If you have not saved, you may loose information."
                  , "Are you sure you want to continue?"
                  ]
```

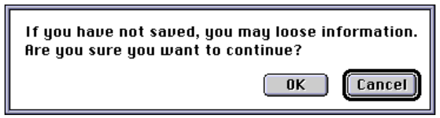

Figure 1.2 A warning before quitting or restarting with a new database.

When the **Cancel** button is chosen, nothing will happen. When the **OK** button is pressed the call-back function given to `warn` is applied. In the case of a **Quit** chosen from the menu the program execution is ended (`Quit` is applied). When **Open new...** is selected from the menu (`restart` is applied) all windows are closed and the program starts all over from scratch.

```
Quit :: SimpleDataBase IO -> (SimpleDataBase, IO)
Quit db io = (db, QuitIO io)
```



```
restart :: SimpleDataBase IO -> (SimpleDataBase, IO)
restart (s,files) io = seqIO initIO ((initTableAdm,files),seq closeIO io)

seqIO fs = seq (map uncurry fs)

closeIO = [ CloseWindows [TableWindowId], closeDbDialogs ]

closeDbDialogs io = seq (map CloseDialog [AttributeDialogId,EdDialogId]) io

TableWindowId       :== 0
EdDialogId          :== 0
AttributeDialogId   :== 1
```

The program will (re)start with performing the actions as described by `initIO`. It will first read the database information from a file (`ReadSimpleDataBase`, see section 1.3) after which it shows the content of the table in a window (`ShowTable`, see section 1.4). Hereafter it displays the edit dialogue (`ShowEditDialog`, see section 1.5).

```
initIO = [ ReadSimpleDataBase, ShowTable, ShowEditDialog ]
```

1.3 File handling

When the application starts it assumes that the database to be manipulated is already stored somewhere in a file. The function `ReadSimpleDataBase` will open a standard dialogue (using `SelectInputFile`) with which the user can select the file to be opened for reading.

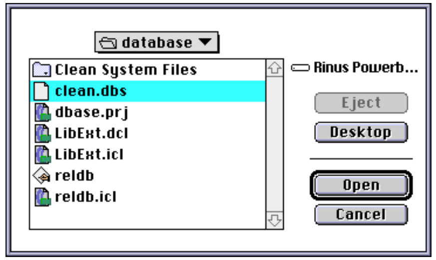

Figure 1.3 Standard selection dialogue for opening a file.

If no file is selected (**Cancel** is pressed) nothing will happen and the program will start with a fresh but empty database. If a file is opened (**Open** is pressed) the information will be read from the selected file and the information is stored in the table administration.

The information stored in a database file should of course obey a certain format (see figure 1.4). In this case it is chosen that on each line exactly one kind of information is stored. The file starts with the information used for filling the descriptor in the table administration: the number of attributes on one card followed by the attribute names. This is followed by the attribute values. The attribute values of one card are preceded by an empty line. In this way the information in the file also stays readable if it is inspected with a simple editor. Remember that in this simple system all attribute values are assumed to be of type `String`. To make it easier to pretty print the attribute names and values their maximum width (`dx` and `fx`) is stored in the table administration.

```
3
Name
E-mail
WWW

van eekelen, marko
marko@cs.kun.nl
http://www.cs.kun.nl/~marko

plasmeijer, rinus
rinus@cs.kun.nl
http://www.cs.kun.nl/~rinus
```

Figure 1.4 The contents of a very small file: 3 attributes names and the attribute values of two cards.

Reading the database contents from a file is straight forward. First the line containing the number of attributes is read. Then this number of lines is interpreted as attibut names. All records start with one additional line that separates the records in the file. This additional line is removed by the pattern match `[_:record]`. The file containing the database is opened as a unique file and closed again in order to be able to use it again to store a new state of the database. The descriptor and list of records are stored in the `state`. The initial `query` is empty (see below). The first record is the initial selection. The layout information files `fw` and `dw` are set to the maximum width of fields and descriptos respectively. Storing these values in the state avoids recomputation.

```
ReadSimpleDataBase :: SimpleDataBase IO -> (SimpleDataBase, IO)
ReadSimpleDataBase (state, files) io
| not done  = ((state,nfiles),nio)
| not open  = ((state,nfiles1),Beep nio)
| not close = ((state,nfiles2),Beep nio)
| otherwise =
    (   ({state & records    =  recs
                , descriptor =  descr
                , query      =  repeatn (size descr) (AS "")
                , selection  =  0
                , name       =  dbname
                , fw =  MaxWidth DbFont.font (map toString (flatten recs))
                , dw =  MaxWidth DbFont.font (map fst descr)}
        ,nfiles2)
    ,nio)
where
    (done, dbname, nfiles, nio) = SelectInputFile files io
    (open, dbfile, nfiles1)     = fopen dbname FReadText nfiles
    (descr,dbfile1, nrofatts)   = FReadDescr dbfile
    (recs, dbfile2)             = FReadTable (inc nrofatts) dbfile1
    (close,nfiles2)             = fclose dbfile2 nfiles1

    FReadDescr file = (map (\d->(d,STRING)) descr,file1,nrofattributes)
    where
     (line1,nfile)  = FReadStrippedLine file
     (descr,file1)  = seqList (repeatn nrofattributes FReadStrippedLine) nfile
     nrofattributes = toInt line1

    FReadTable nroflines file
    |   endOfFile   = ([], file1)
    |   otherwise   = ([map (\s->AS s) record : records], file3)
    where
        (endOfFile, file1) = fend file
        ([_:record],file2) = seqList (repeatn nroflines FReadStrippedLine) file1
        (records,   file3) = FReadTable nroflines file2

    FReadStrippedLine file = (line%(0,maxindex line - 1), file1)
    where
        (line, file1) = freadline file

MaxWidth font []   = 0
MaxWidth font list = maxList (FontStringWidths list font)
```


The current contents of the table administration can be stored again in a(nother) file by choosing **Save as...** from the menu which will lead to a call of the function `SaveTable`. This function uses `SelectOutputFile` which will open a standard dialogue with which the user can type in a new file name and select a directory in which the file is stored.

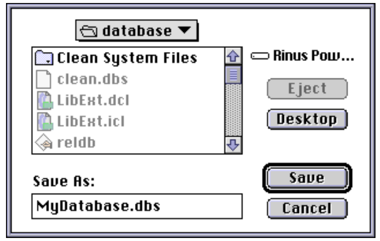

Figure 1.4 Standard selection dialogue for saving a file.

```
SaveTable :: SimpleDataBase IO -> (SimpleDataBase, IO)
SaveTable (state=:{records,descriptor,name}, files) io
|   not done    = ((state, nfiles),nio)
|   not open    = ((state, nfiles1),Beep nio)
|   not close   = ((state, nfiles2),Beep nio)
|   otherwise   = ((state, nfiles2),
                        ChangeWindowTitle TableWindowId (stripDirs dbname) nio)
where
    (done, dbname, nfiles, nio) = SelectOutputFile "Save As: " name files io
    (open, dbfile, nfiles1)     = fopen dbname FWriteText nfiles
    (close,nfiles2)             = fclose (seq ( writedescriptor ++
                                                writerecords) dbfile) nfiles1
    writedescriptor     = [ fwritei (size descriptor),
                                FWriteRecord (map ((\s->AS s) o fst)
descriptor)]
    writerecords        = [ FWriteRecord rec \\ rec <- records ]
    FWriteRecord rec    = fwrites (foldl (+++) "\n"
                              (map (\(AS attribute) -> attribute +++ "\n") rec))
```

1.4 Displaying the database contents

The function `ShowTable` which will be applied when **Show Table** is chosen from the menu opens a scrollable window (the database window, see Figure 1.6). The contents of the cards (the attribute values) preceded by the corresponding attribute names are shown in it. Although one would expect that this is rather simple it involves a lot of tedious work caused by the fact that one has to calculate what to draw where (this depends on the font being used). One also needs to calculate which part of the picture corresponds to which card in the database (for redrawing) and the other way around (to handle mouse clicks in the window).

Figure 1.6. A window on the database, the selected card is high-lighted

```
ShowTable :: SimpleDataBase IO -> (SimpleDataBase, IO)
ShowTable db=:(state=:{records,name},_) io = (db,OpenWindows [window] io)
where
    window = ScrollWindow TableWindowId (5,5) (stripDirs name)
                (ScrollBar (Thumb left) (Scroll DbFont.width))
                    (ScrollBar (Thumb top) (Scroll DbFont.height))
                        domain MinDomainSize (domSize domain)
                            UpdateRecordWindow [Mouse Able MouseSelectItem]
    ((left,top),_)  = domain
    domain          = DbPictureDomain state 0 (size records)

stripDirs namewithdirectories               // remove directory path
    = toString (last (splitby DirSeparator (fromString namewithdirectories)))

splitby :: a .[a] -> [.[a]] | Eq a
splitby x ys = case rest of [] -> [firstpart]; [r:rs] -> [firstpart:splitby x
rs]
where
    (firstpart,rest) = span (\y -> x <> y) ys
```

The window constructor (`ScrollWindow`) requires an outrageous number of arguments: an identification (`TableWindowId`), the initial position of the window (`(0,0)` is the top left corner of the screen), the name of the window (`stripDirs` removes the path name from the stored file name), the initial position of the horizontal (`left`) and vertical (`top`) scroll bars, the size of the picture behind the window (`domain`), minimum size of the window (`MinDomainSize`), initial size of the window (`domSize domain`), the call-back function to be called when the picture has to be redrawn (`UpdateRecordWindow`), and finally a call-back function is defined (`MouseSelectItem`) to handle mouse clicks in the window.

The fixed font which is used to display the database information (in this case `courier`), the maximum font width together with the font height is defined in a global constant record (`DbFont`). It is used to define suitable scroll steps and the information is also used e.g. to determine the size of the picture and the initial size of the window.

```
DbFont   =: {font = f, width = maxwidth, height = ascent+descent+leading}
where
    (ascent, descent, maxwidth, leading) = FontMetrics f
    (_, f)                               = SelectFont "courier" [] 10

domSize ((left,top),(right,bottom)) = (abs (right-left),abs (bottom-top))
```

The function `DbPictureDomain` calculates the size of the picture in pixels. This size depends on the number of cards to display (indicated by the arguments `from` and `to`), the chosen font, the maximum length of the attribute names and values using this font.


```
DbPictureDomain :: TableAdm Int Int -> PictureDomain
DbPictureDomain {descriptor=d,dw,fw} fr to
| top > MaxDomainInt || bottom > MaxDomainInt
            = abort "Error: Database too large to handle"
| dbwidth < minwidth || dbheight < minheight
            = ((~whiteMargin,0),(~whiteMargin+minwidth,minheight))
| otherwise = dbdomain
where
    whiteMargin           = DbFont.width
    (minwidth,minheight)  = MinDomainSize
    (dbwidth,dbheight)    = domSize dbdomain
    top      = toPicCo d fr
    bottom   = toPicCo d to
    dbdomain = ((~whiteMargin,top),
                    (dw + MaxWidth DbFont.font [Separator] +
                        fw + whiteMargin,bottom))

Separator :== ": "
```

When the contents of the window has to be redrawn (e.g. because the window becomes the active front window, because the window size is changed, the window is scrolled, or because the picture is changed) the function `UpdateRecordWindow` is called automatically by the Clean I/O system. For reasons of efficiency it is better to redraw only those parts of the picture that are visible. Parts of the picture which are drawn outside the window will be clipped automatically.

```
UpdateRecordWindow::UpdateArea SimpleDataBase -> (SimpleDataBase,
[DrawFunction])
UpdateRecordWindow domains
                        db=:(state=:{records=recs,descriptor=descr,selection},_)
 = (db, [ SetFont DbFont.font
        : flatten (map Update domains) ] ++
        (if (isEmpty recs) [] (HighLight state selection)))
where
    Update domain=:((_,top),(_,bottom))
     | isEmpty recs = [ EraseRectangle domain ]
     | otherwise    = [ EraseRectangle domain
                      , MovePenTo (0,topofvisiblerecs)
                      : map (DrawCard (map fst descr)) (recs%(toprec,botrec))
                      ]
    where
        topofvisiblerecs = toPicCo descr toprec
        toprec           = toCardCo descr top
        botrec           = toCardCo descr (bottom - 1)

    DrawCard descr rec
     = seq ( drawLine "" ++
           flatten [ drawLine (d +++ Separator +++ f)
                        \\ d<-normwidth descr & AS f<-rec ] )
    where
        normwidth descr = [ f +++
                                toString (spaces ((maxList (map size descr)) -
                                    size f)) \\ f <- descr ]
        drawLine s      = [ DrawString s,
                          , MovePen (~(FontStringWidth s Db
                                Font.font),DbFont.height)
                          ]
```

The system will tell you which part of the picture is visible or where on the picture a mouse is clicked. This is done in pixel co-ordinates (the point (0,0) is the top-left corner of a picture). One needs to calculate which corresponding card is meant. This can be calculated by converting the y-co-ordinate of a pixel to the index of the corresponding card in the table (`toPicCo`). When one wants to redraw a certain card one also needs to know to which part of the picture it corresponds (`toCardCo`).

```
toPicCo:: Descriptor Int -> Int
toPicCo descr n = n * (size descr * DbFont.height + 1)
```

```
toCardCo:: Descriptor Int -> Int
toCardCo descr n = n / (size descr * DbFont.height + 1)
```

The card which is selected (by a mouse click in the window or a search from a query) is highlighted (see Figure 1.6) by the function `HighLight`.

```
HighLight :: TableAdm Int -> [Picture -> Picture]
HighLight state i = [ SetPenMode HiLiteMode
                    , FillRectangle (DbPictureDomain state i (i+1))
                    , SetPenNormal
                    ]
```

A mouse click in the window will have the effect that the call-back function `MouseSelectItem` is called which will change the current card being selected accordingly. Notice that the old card being selected is high-lighted again. High-lighting an area twice will remove the high-light. The information in the Edit window is changed accordingly (`SetCardTextattributes`).

```
MouseSelectItem :: MouseState SimpleDataBase IO -> (SimpleDataBase, IO)
MouseSelectItem ((_,mvpos),ButtonDown,_)
                (state=:{records,descriptor,selection},files) io
| isEmpty records   = ((state, files), io)
| otherwise         = (({state & selection=index},files),
                          ChangeSelection state selection index io)
where
    index = toCardCo descriptor mvpos
    MouseSelectItem _ database io = (database, io)

ChangeSelection:: TableAdm Int Int IO -> IO
ChangeSelection state=:{descriptor=d,records,editinfoid} old new io
 =  seq [ DrawInWindow TableWindowId
            (HighLight state old ++ HighLight state new)
        , SetCardTextattributes state new
        ] io
```

1.5 Changing the database contents

The function `ShowEditDialog` will be applied e.g. when **Edit...** is chosen from the menu. It displays a multi-purpose dialogue. The dialogue has two modes. In one mode (`SetCardTextattributes`) it displays by default the current card (record) being selected (`SetTextattributes`). By pressing **Delete** the selected card can be deleted (`DeleteRecord`). One can also edit the displayed attribute values and **Replace** the contents of the selected card with the new values (`AddRecord Replace`). It is also possible to **Add** a complete new card (`AddRecord (not Replace)`). Finally the database can be sorted (`Sort`) by pressing **Sort**. In the other mode the same dialogue can also be used to fill in a query to search for a certain card in the database (see 1.6).

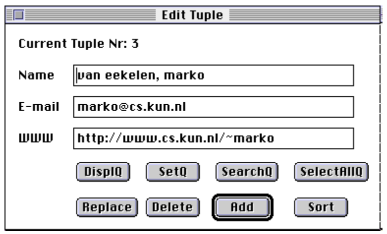

Figure 1.7. The edit dialogue to change the selected card.

```
ShowEditDialog  :: SimpleDataBase IO -> (SimpleDataBase, IO)
ShowEditDialog (s=:{descriptor=descr,records=recs,selection=sel},files) io
 = ((nstate,files), seq [   OpenDialog editDialog,
                            SetCardTextattributes nstate sel] io)
where
 nstate      = {s & editinfoid=infoid}
```



```
 editDialog  = CommandDialog EdDialogId "Edit Record" [] addId dialogitems
 dialogitems =
  [ DynamicText infoid Left InputBoxWidth ""    ] ++
  flatten [ inputattribute sid eid attribute \\ attribute <- map fst descr
                                              &  eid  <- [0..]
                                              & sid <- [size descr..]] ++
  [ DialogButton dispQId (Below (maxindex descr)) "DisplQ" Able DisplQuery
  , DialogButton setQId  (RightTo dispQId) "SetQ"       Able SetQuery
  , DialogButton srchQId (RightTo setQId)  "SearchQ"    Able Search
  , DialogButton slctQId (RightTo srchQId) "SelectAllQ" Able SelectAll
  , DialogButton replId  (Below dispQId)   "Replace"    Able (AddRecord Replace)
  , DialogButton delId   (Below setQId)    "Delete"     Able DeleteRecord
  , DialogButton addId   (Below srchQId)   "Add"        Able
                                                    (AddRecord (not Replace))
  , DialogButton sortId  (Below slctQId)   "Sort"       Able Sort
  ]

 inputattribute sid eid attribute = [ StaticText sid Left attribute
                                    , EditText eid pos InputBoxWidth 1 ""
                                    ]
 where
    pos = case eid of
          0    -> XOffset sid offset
          else -> Below (dec eid)
    offset = Pixel (DfFont.width +
                MaxWidth DfFont.font (map fst descr) -
                    MaxWidth DfFont.font [attribute])

  [infoid,dispQId,setQId,srchQId,slctQId,replId,delId,addId,sortId:_]
        = [2*(size descr)..]

InputBoxWidth   :== Pixel (20*DfFont.width)     // DefaultWidth boxes (20 chars)
Replace         :== True

SetCardTextattributes :: TableAdm Int IO -> IO
SetCardTextattributes {editinfoid,descriptor=d,records=recs} index io
| isEmpty recs  = SetTextattributes editinfoid
                    "No Table: Empty Database!" d (repeatn (size d) (AS "")) io
| otherwise     = SetTextattributes editinfoid
                    ("Current Record Nr: " +++ toString index) d (recs!index) io

SetTextattributes:: Int String Descriptor Record IO -> IO
SetTextattributes id s d rec io
    = ChangeDialog EdDialogId
        [   ChangeDynamicText id s
        :   [ ChangeEditText id f \\ id <- [0..maxindex d] & AS f<-rec ]
        ] io

DfFont  =: {font = f, width = maxwidth, height = ascent+descent+leading}
where                           // Global graph def: default font (in dialogs)
    (ascent, descent, maxwidth, leading) = FontMetrics f
    (_, f)                               = SelectFont name styles size
    (name,styles,size)                   = DefaultFont
```

The current card being selected is deleted by removing the corresponding record from the list of records. The next card in the database becomes selected. Now also the picture showing the table has to be redrawn (see 1.4) accordingly.

```
DeleteRecord :: DialogInfo SimpleDataBase IO -> (SimpleDataBase, IO)
DeleteRecord dialogInfo
db=:(state=:{records=oldrecs,selection=index,descriptor,fw},files) io
| isEmpty oldrecs = (db,Beep io)
| otherwise       = UpdateDbDomain (nstate,files)
                        (ChangeSelection nstate index nindex io)
where
 newrecs         = remove index oldrecs
 attributewidth  = if fw == MaxWidth DbFont.font (map toString (oldrecs!index))
                      (MaxWidth DbFont.font (map toString (flatten newrecs)))
                      fw
 nindex  = if (isEmpty newrecs) 0 (index mod (size newrecs))
```

```
 nstate = { state & records     =   newrecs,
                    selection   =   nindex,
                    fw          =   attributewidth }
```

The function `AddRecord` reads the changed or new attribute values from the editable text boxes in the edit dialogue (`GetTextattributes`). Depending on the Boolean argument it either adds a complete new card or it replaces the selected one.

```
AddRecord :: Bool DialogInfo SimpleDataBase IO -> (SimpleDataBase, IO)
AddRecord replace dialogInfo
                   db=:(state=:{descriptor,selection,records=recs,fw},files) io
| isEmpty recs && replace   = (db,Beep io)
| otherwise                 = UpdateDbDomain (nstate,files)
                                      (ChangeSelection nstate selection index io1)
where
    (newrec,io1)    = GetTextattributes descriptor io
    (index,newrecs) = insertindex (\a b -> a <= b) newrec
                                      (if replace (remove selection recs) recs)
    attributewidth  = if recalc
                         (MaxWidth DbFont.font (map toString (flatten newrecs)))
                         (max (MaxWidth DbFont.font (map toString newrec)) fw)
    recalc          = replace && MaxWidth DbFont.font
                                           (map toString (recs!selection)) < fw
    nstate          = { state & records     =   newrecs,
                                selection   =   index,
                                fw          =   attributewidth }

insertindex :: (a -> .(a -> .Bool)) a u:[a] -> (Int,u:[a])
insertindex r x ls = inserti r 0 x ls
where
    inserti r i x ls=:[y : ys]
    | r x y          = (    i,[x : ls])
    | otherwise      = (index,[y : list])
    with
        (index,list) = inserti r (inc i) x ys
    inserti r i x [] = (    i,[x])

GetTextattributes :: Descriptor IO -> (Record,IO)
GetTextattributes descr io
    = ([AS (GetEditText id dialogInfo) \\ id <- [0..maxindex descr]],nio)
where
    (_,dialogInfo,nio) = GetDialogInfo EdDialogId io
```

To sort the database one can simply call the library function `sort` from Cleans `StdEnv` under the condition that the class `<` is instantiated for attribute values.

```
Sort :: DialogInfo SimpleDataBase IO -> (SimpleDataBase, IO)
Sort dialogInfo (state=:{records=recs,selection},files) io
 = UpdateDbDomain (nstate,files) (SetCardTextattributes nstate selection io)
where
    nstate = {state & records = sort recs}

instance < AttributeValue
where
    (<) (AS a) (AS b) = a < b
```

1.6 Handling database queries

The dialogue shown by the function `ShowEditDialog` (see the specification in 1.5) can also be used to fill in a query to search for certain cards in the database. When **DisplQ** is pressed (`DisplQuery` is called) a query previously set is displayed. The dialogue box now starts with **Query:**, set by `SetQueryTextattributes`). A new query can be typed in the editable boxes and set by pressing **SetQ** (`SetQuery`). Hereafter one can search for the next card matching the previously set query by pressing **SearchQ** (calling `Search`). It is also possible to find all cards matching the query set by pressing **SelectAllQ** (`SelectAll`).

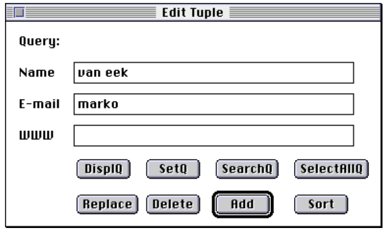

Figure 1.8 The edit dialogue now used to define a query.

```
DisplQuery ::DialogInfo SimpleDataBase IO -> (SimpleDataBase, IO)
DisplQuery info db=:({descriptor=d,query=q,editinfoid},_) io
    = (db,SetQueryTextattributes editinfoid d q io)

SetQueryTextattributes:: Int Descriptor Record IO -> IO
SetQueryTextattributes editinfoid d q io
                = SetTextattributes editinfoid "Query: " d q io

SetQuery ::DialogInfo SimpleDataBase IO -> (SimpleDataBase, IO)
SetQuery info (state, files) io = (({state & query = nquery},files), nio)
where
    (nquery,nio) = GetTextattributes state.descriptor io
```

When a query is set one can search for a card in the database matching this query. Each query value typed in is compared with the corresponding attribute value of a card. They are considered equal when the query value is a prefix sub-string of the card value. A card is found if all attribute values match the query (`QueryRecord`). A search for a card starts with the card following the card currently being selected. Cards are inspected in a round robin way until a matching card is found or the searching is stopped because none of the cards turn out to match the query. If a card is found it will be selected and its contents is both displayed in the edit window as well as in the database window (`MakeSelectionVisible`).

```
Search ::DialogInfo SimpleDataBase IO -> (SimpleDataBase, IO)
Search  info database=:(state=:{records,query,selection=sel},files) io
| isEmpty found = (database, Beep io)
| otherwise
    = MakeSelectionVisible ({state & selection=nsel},files)
                                   (ChangeSelection state sel nsel io)
where
    nsel    = hd found
    found   = [ i \\ e <- el ++ bl
                   &  i <- [sel+1 .. maxindex records] ++ [0..]
                   |  QueryRecord query e
              ]
    (bl,el) = splitAt (sel+1) records

QueryRecord:: Record Record -> Bool
QueryRecord query e =   and [ EqPref qf f \\ AS f <- e & AS qf <- query ]
where
    EqPref pref name
    |   size pref > size name   =   False
    |   otherwise               =   pref == name%(0,maxindex pref)

MakeSelectionVisible:: SimpleDataBase IO -> (SimpleDataBase,IO)
MakeSelectionVisible db=:({records,selection,descriptor},_) io
| isEmpty records     = (db,io)
| selection_invisible = ChangeScrollBar TableWindowId
                                          (ChangeVThumb selthumb) db io1
| otherwise           = (db,io1)
where
    selection_invisible = selthumb < visibletop || selthumb >= visiblebot
    selthumb            = toPicCo descriptor selection
    (((_,visibletop),(_,visiblebot)), io1) = WindowGetFrame TableWindowId io
```

The function `SelectAll` filters out all cards which do not satisfy the query set. A new database (named `SelectedTable`) is formed by those cards who pass the test.

```
SelectAll ::DialogInfo SimpleDataBase IO -> (SimpleDataBase, IO)
SelectAll info db=:(state=:{records,query,selection,descriptor},files) io
| isEmpty recs = (db, Beep io)
| otherwise    = UpdateDbDomain (nstate,files)
                   (seq [  ChangeSelection nstate selection 0,
                           ChangeWindowTitle TableWindowId nstate.name] io)
where
    nstate  = { state & selection   =   0,
                        records     =   recs,
                        name        =   "SelectedTable",
                        fw          =   MaxWidth DbFont.font
                                            (map toString (flatten recs)) }
    recs    = filter (QueryRecord query) records
```

1.7 Changing the format of the database

The function `ShowAttributeDialog` which will be applied when choosing **Change Set Up…** from the menu (see 1.2) opens a dialogue (see Figure 1.9.a) in which all the attribute names of the database are displayed. One can click **Delete** to delete the selected attribute (`Deleteattribute`), click **Move** to move the order in which the attributes are stored (`Moveattribute`), click **Rename** to give a new name to an existing attribute (`Renameattribute`), and click **Add New** to add a new attribute name (`Addattribute`). If the database is empty a dialogue is opened (see Figure 1.9.b) using the auxiliary function `inputdialog` prompting for a first attribute name to be added (see 1.7.4).

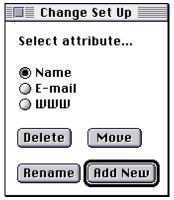

Figure 1.9.a The change set up dialogue showing the attribute names of a non-empty database.

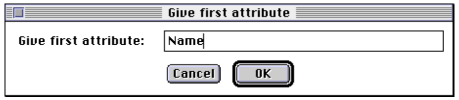

Figure 1.9.b The change set up dialogue in case of an empty database

```
ShowAttributeDialog :: SimpleDataBase IO -> (SimpleDataBase, IO)
ShowAttributeDialog db=:({descriptor=d},_) io
| isEmpty d = inputdialog "Give first attribute" InputBoxWidth
                (\input->AttributeChangeIO (adda (-1) input)) db nio
| otherwise = (db,OpenDialog attributedialog nio)
where
    nio             = CloseDialog EdDialogId io
    attributedialog
     =  CommandDialog AttributeDialogId "Change Set Up" [] addId
        [   StaticText DontCareId Left "Select attribute..."
        ,   RadioButtons selectId Left (Columns 1) firstRadioId
                                (radioitems firstRadioId (map fst d))
        ,   DialogButton deleteId Left "Delete"  Able
                                (Deleteattribute getselectedattribute)
        ,   DialogButton moveId (RightTo deleteId) "Move" Able
                                (Moveattribute getselectedattribute)
        ,   DialogButton renameId Left "Rename"  Able
                                (Renameattribute getselectedattribute)
```



```
        ,   DialogButton addId (Below moveId) "Add New" Able
                                      (Addattribute getselectedattribute)
        ]

    getselectedattribute dialoginfo
        = GetSelectedRadioItemId selectId dialoginfo - firstRadioId

    [deleteId,moveId,renameId,addId,selectId,firstRadioId:_] = [0..]

radioitems firstid titles
    = [RadioItem id t Able selectdummy \\ id <- [firstid..] & t <- titles]
where
    selectdummy dialoginfo dialogstate = dialogstate

inputdialog name width fun s io = (s,OpenDialog dialogdef io)
where
    dialogdef
     =  CommandDialog dlgId name [] okId
        [   StaticText nameId Left (name+++": ")
        ,   EditText inputId (RightTo nameId) width 1 ""
        ,   DialogButton cancelId (Below inputId) "Cancel" Able (cancel dlgId)
        ,   DialogButton okId (RightTo cancelId) "OK" Able (ok fun)
        ]

    ok fun dlginfo s io
     = fun (GetEditText inputId dlginfo) s (CloseDialog dlgId io)
    cancel id dialoginfo s io              = (s, CloseDialog id io)
    [dlgId,nameId,inputId,cancelId,okId:_]  = [0..]
```

Any change in the attribute names will close the change set up and edit dialogue while the database window is redrawn taking the changes into account (`AttributeChangeIO`). When the change set up and edit dialogue are re-opened the changes are taken into account. This shows the dynamic behaviour of these kind of dialogues specification in contrast with the static description often generated by many dedicated (visual) dialogue editors (see e.g. Figure 1.12).

```
AttributeChangeIO :: (TableAdm -> TableAdm) SimpleDataBase IO ->
(SimpleDataBase,IO)
AttributeChangeIO changefun (state,files) io
    = UpdateDbDomain (changefun state,files) (closeDbDialogs io)
```

1.7.1 Deleting an attribute field

The function `Deleteattribute` will delete the indicated attribute name and *all* the corresponding attribute values from the database administration. Which attribute is deleted is depending on which radio button is set in the change set up dialogue (see Figure 1.9.b). A dialogue (Figure 1.10) warns the user that the a complete attribute will be deleted if **OK** is pressed.

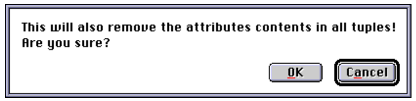

Figure 1.10 The change set up dialogue showing the attribute names of a non-empty database.

```
Deleteattribute :: (DialogInfo -> Int) DialogInfo SimpleDataBase IO
                                                    -> (SimpleDataBase, IO)
Deleteattribute getattribute dialoginfo db io
    = warn warning (AttributeChangeIO (deletea (getattribute dialoginfo))) db io
where
    warning = [ "This will also remove the attributes contents in all records!",
                "Are you sure?" ]
```

```
deletea i s=:{records=rs,descriptor=d,query=q}
    = {s &      records     =   newrs,
                descriptor  =   newdescr,
                query       =   remove i q,
                dw          =   MaxWidth DbFont.font (map fst newdescr),
                fw          =   nfw }
where
    newrs    = map (remove i) rs
    newdescr = remove i d
    nfw      = MaxWidth DbFont.font (map toString (flatten newrs))
```

1.7.2 Moving an attribute field

The function `Moveattribute` will move the indicated attribute name and *all* the corresponding attribute values from the database administration to a new position. Which attribute is moved is depending on which radio button is set in the change set up dialogue (see Figure 1.9.b). A move attribute dialogue (Figure 1.12) will ask the user to which new position the attribute has to be moved. When the change set up dialogue or edit dialogue is re-opened one can inspect that the move indeed has taken place.

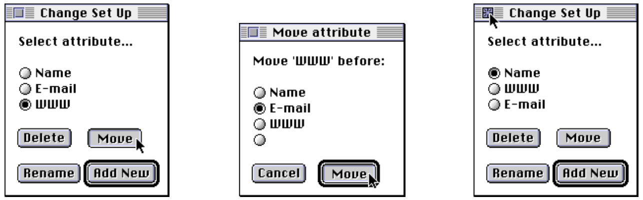

Figure 1.12 Moving an attribute from position effects the appearance of the change set up dialogue.

```
Moveattribute :: (DialogInfo -> Int) DialogInfo SimpleDataBase IO
                                                             ->
(SimpleDataBase,IO)
Moveattribute getattribute dialoginfo db=:({descriptor=d},_) io
    = (db,OpenDialog movedialog io)
where
 attributetomove = getattribute dialoginfo
 movedialog
    = CommandDialog moveDialogId "Move attribute" [] okId
      [ StaticText   infoId   Left
                     ("Move '" +++ fst (d!attributetomove) +++ "' before: ")
      , RadioButtons selectId Left (Rows (inc (size d))) firstRadioId
                     (radioitems firstRadioId (map fst d++[""]))
      , DialogButton cancelId Left "Cancel" Able (cancel moveDialogId),
      , DialogButton okId     (RightTo cancelId) "Move" Able
                     (ok (movea attributetomove))
      ]

 [moveDialogId,cancelId,okId,infoId, selectId,firstRadioId:_] = [0..]

 ok mvf dlginfo s io
  = AttributeChangeIO (mvf destinationattribute) s (CloseDialog moveDialogId io)
 where
   destinationattribute = GetSelectedRadioItemId selectId dlginfo - firstRadioId
```

1.7.3 Renaming an attribute field

The function `Renameattribute` allows to rename the chosen attribute name. An input dialogue (Figure 1.13) will prompt the user for the new name. The administration (the descriptor) is changed accordingly.

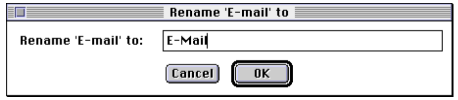

Figure 1.13 Defining the new field attribute to be added to a card

```
Renameattribute :: (DialogInfo -> Int) DialogInfo SimpleDataBase IO
                                                   -> (SimpleDataBase,IO)
Renameattribute getattribute dialoginfo db=:(state,files) io
 = inputdialog infotext InputBoxWidth
        (\input->AttributeChangeIO (renamea attributetorename input)) db io
where
 infotext = "Rename '" +++ fst (state.descriptor!attributetorename) +++ "' to"
 attributetorename  = getattribute dialoginfo

 renamea selectedattribute newattributename s=:{descriptor=d}
    = {s &  descriptor  =   newdescr,
            dw          =   MaxWidth DbFont.font (map fst newdescr) }
 where
    newdescr = updateAt selectedattribute (newattributename,STRING) d

updateAt :: .Int .a [.a] -> [.a]
updateAt i x ys = before ++ [ x : case at of [] -> []; [r:rs] -> rs ]
where
    (before,at) = splitAt i ys
```

1.7.4 Adding an attribute field

The function `Addattribute` will add a new attribute name after the selected old attribute name. An input dialogue (Figure 1.13) will ask the user to for the new attribute name. The database administration is changed accordingly. A corresponding attribute value initialized with a default value is added to each card. In the database window the new field is added (see Figure 1.15).

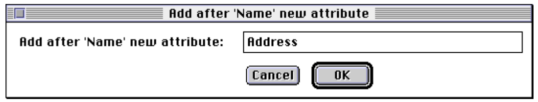

Figure 1.14 Defining the new field attribute to be added to a card

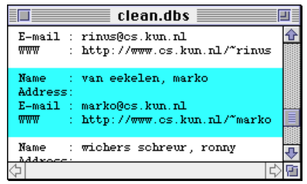

Figure 1.15 The attribute has been added to each card, the empty string has been taken as default value.

```
Addattribute :: (DialogInfo -> Int) DialogInfo SimpleDataBase IO
                                                             ->
(SimpleDataBase,IO)
Addattribute getattribute dialoginfo db=:(state,files) io
 = inputdialog infotext InputBoxWidth
        (\input->AttributeChangeIO (adda attributename input)) db io
where
    infotext        =   "Add after '"   +++
                        fst (state.descriptor!attributename) +++
                        "' new attribute"
    attributename   =   getattribute dialoginfo
```

```
adda :: Int String TableAdm -> TableAdm
adda afterattribute attributename state=:{records=rs,descriptor=d,query=q,dw}
 =  {state &    records     =   map (ins (AS "")) rs,
                descriptor  =   ins (attributename,STRING) d,
                query       =   ins (AS "") q,
                dw          =   descrwidth  }
where
    ins x ys   = insertAt (inc afterattribute) x ys
    descrwidth = max (MaxWidth DbFont.font [attributename]) dw

insertAt :: .Int .a u:[.a] -> u:[.a]
insertAt i x ys = before ++ [ x : at ]
where
    (before,at) = splitAt i ys

movea :: Int Int TableAdm -> TableAdm
movea sf df s=:{records=rs,descriptor=d,query=q}
 = {s &         records         =   map (moveinlist sf df) rs,
                descriptor  =   moveinlist sf df d,
                query       =   moveinlist sf df q  }

moveinlist :: .Int .Int .[a] -> [a]
moveinlist src dest l
 | src < dest = remove src beforedest ++ [l!src : atdest]
 | src > dest = beforedest ++ [l!src : remove (src - dest) atdest]
 | otherwise  = l
where
    (beforedest,atdest) = splitAt dest l
```

1.8 Exercises

1.1 Assume that the database can also contain attribute values of other type, say `Int` and `Real`. Modify the program (the table administration, the `ReadSimpleDataBase`, the `SaveTable` and all other functions which manipulate the database) such that the database application can deal with these other types.

1.2 Make it possible for the user to choose anyone of the other fonts and font sizes available on the system to be used to write the information in the database window. Add a new item to the menu which opens a standard dialogue to make the font selection. Safe the information of the chosen font in the database state.

1.3 The query one can define is rather rigid. Only if the prefix of an attribute value exactly matches the query a match is possible. Change the function `QueryRecord` such that the query string is considered equal when the string can be found anywhere in the corresponding attribute value.

1.4 Using one and the same dialogue for editing and defining queries is not so user friendly. Split the dialogue up in two parts.

1.5 Define an overloaded `==`, `<`, `<=`, `>`, `>=`, `||` and `&&` on all attribute and query values, also for values of type `Int` and `Real`. Define a query expression language in which these operators can be used. Make a parser for this language. It should be possible to specify a search for an integer value `>= 0 && <= 10`.

1.6 Make it possible to define several query alternatives. For each of the alternatives a separate query dialogue has to be filled in. A search for a card should succeed when the card matches any of the query alternatives specified in this way.

1.7 Make it possible to display the database contents in the database windows in a way as defined by the user. Open a window in which the user can add, delete and drag any of the attribute names or values. Make a menu from which the available attribute names and values can be chosen. Also allow to draw lines and boxes to give the user a possibility to create his own lay-out. Store the lay-out in the database file.