BlackBox Tutorial

Oberon microsystems, Inc.
Technoparkstrasse 1
CH-8005 Zürich
Switzerland

Net: info@oberon.ch
Tel: +41 (0) 44 445 17 51
Fax: +41 (0) 44 445 17 52

Table of Contents

Part I: Design Patterns

Part II: Library

Part I: Design Patterns

Part I of the BlackBox tutorial gives an introduction to the design patterns that are used throughout BlackBox. To know these patterns makes it easier to understand and remember the more detailed design decisions in the various BlackBox modules.

Part II of the BlackBox tutorial demonstrates how the most important library components can be used: control, form, and text components.

Part III of the BlackBox tutorial demonstrates how new views can be developed, by giving a series of examples that gradually become more sophisticated.

1 User Interaction

In this section, we will discuss how a user interface can be made user-friendly. Graphical user interfaces are part of the answer. Unfortunately, this answer leads to another problem: how can such a user interface be implemented at reasonable cost? The design approach which solves this problem is surprisingly fundamental, and has deep consequences on the way we construct modern software. It is called object-oriented programming.

1.1 User-friendliness

There are at least 100 million personal computers in use today. With so many users who are not computer specialists, it is important that computers be easy to use. This implies that operating systems and applications must be user-friendly. What does this mean? There are several aspects.
First, the various applications should be consistent with one another: a function that is available in several programs should have the same user interface in all of them, so that the user does not have to learn different ways to do one and the same thing. One way to achieve this is to publish user interface guidelines that all software developers should adhere to. This approach was championed by Apple when they introduced the Macintosh. A second approach is to implement a function only once, so that it can be reused across different applications. This is the idea of component software. Apple didn't go that far, but they at least provided a number of so-called "toolbox" routines. The toolbox is a comprehensive library built into the Mac OS.

Second, the idea of the desktop metaphor, where icons graphically represent files, directories, programs, and other items, made it possible to replace arcane commands that have to be remembered and typed in by more intuitive direct manipulation: for example, an icon can be dragged with the mouse and dropped into a waste basket, instead of typing in "erase myfile" or something similar. Such a rich user interface requires a vast set of services. Not surprisingly, the Mac OS toolbox included services for windows, menus, mouse, dialog boxes, graphics, and so on. The toolbox made it possible to build applications that really adhered to the user interface guidelines. Unfortunately, programming for such an environment implied a tremendous leap in complexity. As a result, the Macintosh was much easier to use than earlier computers, but it was also much more difficult to program.
Today, the browser metaphor is an increasingly popular addition to traditional graphical user interfaces. Combined with Internet access, it opens a huge universe of documents to a computer user - and makes the job of a developer even more complicated.

Finally, one of the most general and most important aspects of user-friendliness is the avoidance of modes. A modal user interface separates user interactions into different classes, and gives a separate environment to each class. Switching between environments is cumbersome. For example, in a database application, a user may open a window (or "screen", "mask") for data entry. In this window, no records may be deleted, no records may be searched, no notes taken, no email read, etc. For each of these other tasks, the data entry environment would have to be left first. This is cumbersome, and reminds us of the application- rather than the document-centric way of computing discussed in the first part of this book. In complex applications, it is often difficult to remember where you currently are, what commands are available, and where you may go next. Figure 1-1 shows an example of the states and possible state transitions of a hypothetical modal application.

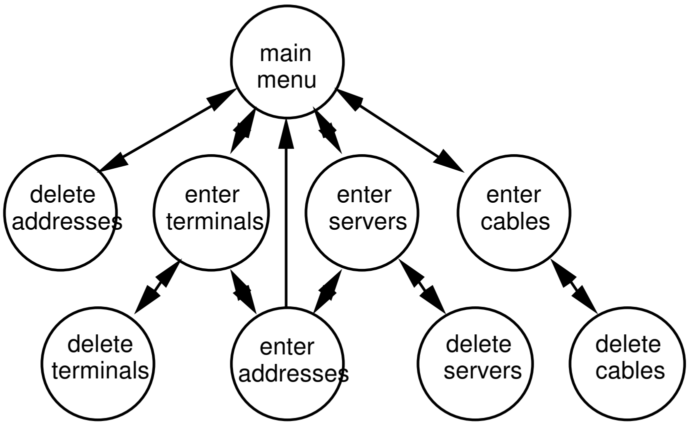


Figure 1-1. Navigation possibilities in a modal user interface

Non-modal user interfaces have no separate working environments, or at least let you switch between environments without losing the state of others. For example, if during data entry you want to look up something with a library browser or text searching tool, you don't lose the partial data that you have already entered. If there are separate working environments, navigation is made simple and obvious by giving explicit feedback about the environment and its capabilities.
A modern graphical user interface is a prime example of a mostly non-modal user interface. There may be several windows (environments) open at the same time, but it is clearly visible with which window you are currently interacting (usually the "top" window). You can work in one window, temporarily work in another one (thereby bringing it to the top), and later switch back to resume your work in the first window.

These user interfaces are a huge improvement over old-style modal interfaces. Users feel more comfortable because they have a better idea of where they are and what they can do. However, even modern GUIs are far from perfect. Most of them still use modal dialog boxes, i.e., windows that you can't put aside to work with another one for a while. For example, a typical modal file dialog box lets you open a file, but doesn't allow you to quickly switch to a file searching tool and back again. The dialog box "modes you in". Even the fact that you have to bring a window to the top before manipulating it constitutes an inconvenient and unnecessary mode, at least if you have a screen large enough that you can lay out all needed windows in a non-overlapping fashion. And many database products don't allow you to have several data entry or data manipulation forms open at the same time. A Web browser is the closest thing to a truly non-modal user interface today: if you have entered some incomplete data into an HTML form, and then switch to another Web page, the browser won't prevent you from doing it.

The BlackBox Component Builder is more radical than other tools in that it simply doesn't support modal dialog boxes. Every dialog box is non-modal, except for a very few built-in operating system dialog boxes such as the standard file dialog boxes. In general, you can always put dialog boxes aside and get back to them later.

1.2 Event loops

Implementing a non-modal application looks deceptively simple: the program waits for a user event, such as a mouse click or a key press; reacts in an appropriate way; waits for the next event; and so on. This programming style, with a central loop that polls the operating system for events and then calls the appropriate handlers for them, is called event-driven programming.

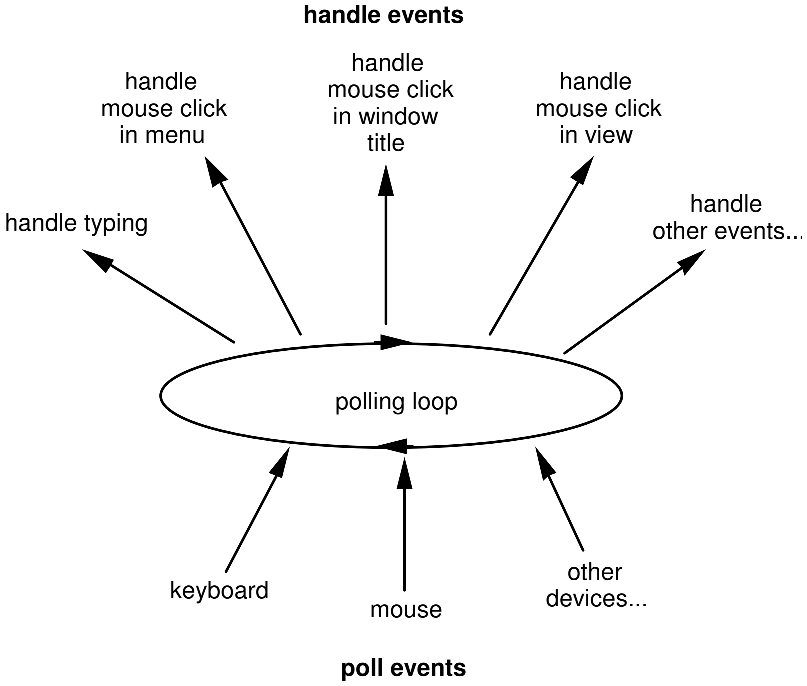


Figure 1-2. Event loop of a non-modal program

The event-polling loop of a typical event-driven program looks similar to the following program fragment:

```
PROCEDURE Main;
   VAR event: OS.Event;
BEGIN
   LOOP
      event := OS.NextEvent();    (* poll the operating system for the next event *)
      IF event IS KeyDownEvent THEN
         HandleKeyDown(event)
      ELSIF event IS MouseDownEvent THEN
         IF MouseInMenu(event) THEN
            HandleMenuEvent(event)
         ELSIF MouseInWindowTitle(event) THEN
            HandleTitleEvent(event)
         ELSIF MouseInWindowBorder(event) THEN
            HandleBorderEvent(event)
         ELSIF MouseInWindowContents(event) THEN
            HandleContentsEvent(event)
         ELSIF...
            ...
         END
      ELSIF ...
         ...
      END
   END Main;
```

Listing 1-3. Event loop with cascaded selection of event handlers

In realistic programs, these cascaded IF-statements can easily span several pages of code, and are always similar. Thus it suggests itself to put this code into a library (or even better, into the operating system).
Such a library would implement the standard user interface guidelines. For example, the *HandleTitleEvent* would further distinguish where in the window's title bar the mouse has been clicked. Depending on the exact location, the window will be dragged to another place, zoomed, closed, etc. This is a generic behavior that can, and for consistency reasons should, only be implemented once. A procedure like *HandleContentsEvent* is different, though. A standard library cannot know what should happen when the user has clicked into the interior (contents) area of a window.
If we require that the library itself must never need adaptation to a particular application, then there results a peculiar kind of system structure, where a library sometimes calls the application. Such a call is called a "call-back". When we visualize an application as sitting on top of the library, it becomes obvious why such calls are also called "up-calls" and the resulting programming style as "inverted programming":

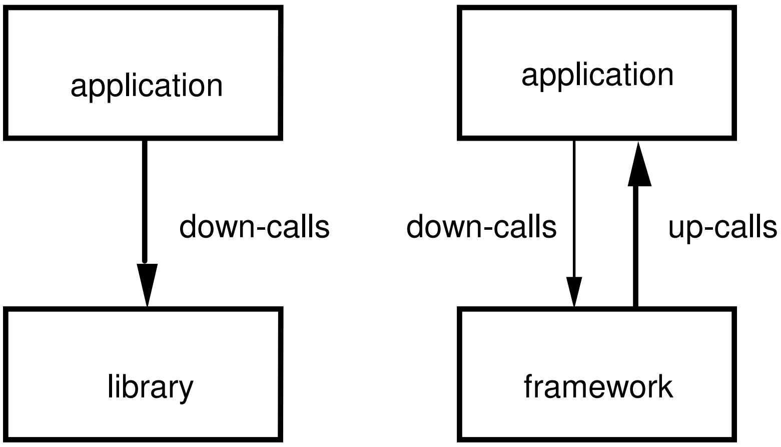


Figure 1-4. Inverted programming design pattern

A library which strongly relies on inverted programming is called a *framework*. It is a semi-finished product that must be customized by plugging in procedures at run-time. For some of the procedures, this can be optional: for procedure *HandleTitleEvent* there can be a default implementation that implements the standard user interface guidlines. Such procedures will only be replaced if unconventional behavior is desired.

Experience shows that a procedure plugged into a framework frequently refers to the same data. For example, the procedure *HandleTitleEvent* will often access the window in which the user has clicked. This makes it convenient to bundle the procedure and its state into a capsule. Such a capsule is called an *object*, and it lifts inverted programming to *object-oriented programming*.

Thinking about user-friendly software led us to the problem of how to reduce the amount of repetitive and complex coding of event loops, which led us straight to the inverted programming design style; to frameworks as a way of casting such a design style into code; and to object-oriented programming as a means to implement frameworks in a convenient way.

The BlackBox Component Framework hides the event loop from application programmers. It goes even further than older frameworks in that it also hides platform-specific user-interface features such as windows and menus. This was achieved by focusing on the abstraction that represents the contents of a window: the so-called *View* type. We will come back to this topic in Chapter 2 in more detail.

1.3 Access by multiple users

Personal computers have made computer users independent from central IT (information technology) departments, their bureaucracies, and their overloaded time-sharing servers. This independence is a good thing, if it can be combined with integration where this is useful and cost-effective. Local area networks, wide area networks, and then the Internet have made integration possible. When two computers cooperate over a network, one of them asks the other to provide a service, for example to deliver a file over the network. The computer which issues the request is called the *client*, the other one is called the *server*. The same machine may act both as a client and as a server, but often these roles are assigned in a fixed manner: the clerk at the counter of a bank's branch office always uses a client machine, and the large box in the air-conditioned vault of the bank's headquarters is always used as a server. A bank's internal network may connect thousands of clients to dozens of servers. But even the smallest network obviously requires that applications are split into two parts: a client and a server part. Consequently, this kind of architecture is called *client/server* computing. It allows to assign processing tasks to the machines which are most suitable. In enterprise environments, the most popular assignment is to put a centralized database on a *database server*, and the remaining functionality on the client machines. This is called a *2-tier* architecture. It requires *fat clients*, i.e., client machines have to perform everything but the actual database accesses. To reduce this burden, high-end database management systems allow to execute some code on the server itself, as an interpreted *stored procedure*. This can greatly increase performance, because it can prevent large amounts of data being shuffled back and forth over the network.

If large numbers of clients are involved, it can become hard to keep all client installations up-to-date and consistent. This problem can be reduced using a *3-tier* architecture, where special application servers are interposed between clients and servers. The "thin" clients are reduced to implementing user interfaces, the database servers handle the databases, while the application servers contain all application-specific knowledge ("business logic").

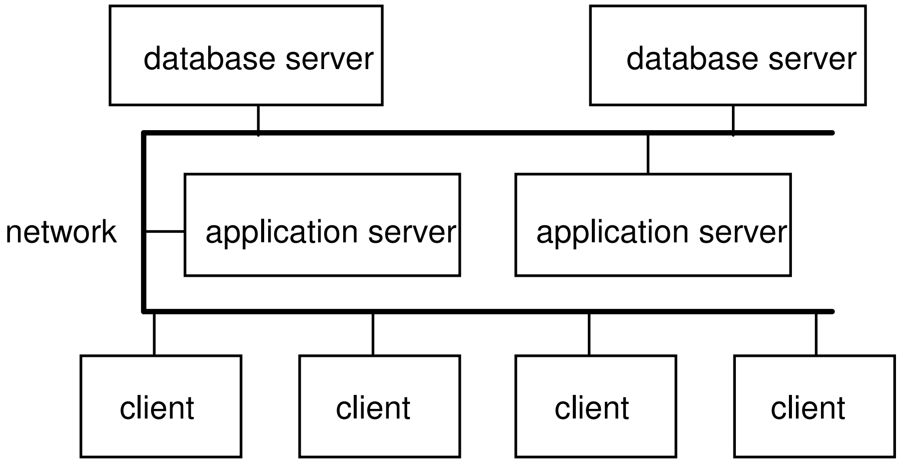


Figure 1-5. 3-tier client/server architecture pattern

A 3-tier architecture is reasonably manageable and scalable in terms of size. Note that a software system separated in this way can be scaled down, in the extreme case by putting the client, application server, and database server on a single machine. The other way around does not work: a monolithic application cannot be easily partitioned to fit a client/server architecture.

Clients and servers are coupled over a network. Communication either operates at the low abstraction level of untyped byte streams (e.g., using a sockets interface for TCP/IP communication) or at the high abstraction level of typed data. In the latter case, either distributed objects (for immediate point-to-point communication) or message objects (for multicast or delayed communication) are used.

The *Comm* subsystem of the BlackBox Component Framework provides a simple byte stream communication abstraction, on top of which messaging services can be built. Among other things, this has been used to implement remote controls, i.e., controls which visualize and manipulate data on a remote machine. For

example, the internal state of an embedded system can be monitored in this way.
The *Sql* subsystem of the BlackBox Component Framework provides a simple distributed object interface specialized for accessing SQL databases. More general DCOM-based [COM] distributed objects can be accessed and implemented with the optional Direct-To-COM Component Pascal compiler.

1.4 Language-specific parameters

The global nature of today's software business often makes it necessary to produce "localized" versions of a software package. At the least, this requires translation of all string constants in the program that the user may see. For example, in a country like Switzerland, where four official languages are spoken, it is often required that the same application is available in German, French, and Italian language versions. This can even mean that the layouts of dialog boxes need to be adapted to the different languages, since the captions and control labels have different lengths in different languages.
If such language-specific aspects are hard-coded into the program sources, language-specific versions require editing and compiling the source code. This is always a sensitive point, since errors and inconsistencies can easily be introduced. It is also very inconvenient: hard-coded layouts of dialog boxes are very unintuitive to modify, and the edit-compile-run cycle is cumbersome if you only want to move or resize a control.
Many tools have been developed which provide more convenient special-purpose editors, in particular layout editors for dialog boxes. One type of tool produces source code out of the interactively produced layout, which then has to be compiled. Since a programmer have to edit the generated source code, it is easy to introduce inconsistencies between the layout and the source code.
Fortunately, there is a much better way to solve this problem, by avoiding the intermediate source-code generator and directly use the editor's output in the program. The editor saves the dialog box layout in a file, a so-called *resource* file. The program then reads these resources when it needs them, e.g., when it opens a dialog box. Resources can be regarded as persistent objects which are only modified if the configuration is changed.

We have seen that the problem of convenient adaptation of a program to different languages can be solved by separating language- or location-specific parameters from the source code. They are put into resource files, which can be modified by convenient special-purpose editors.
Moving user-interface aspects away from the proper application logic is a way to make client software more modular and better adaptable. If taken to the extreme, resources almost *become* the client software: a Web browser on a client is all that is needed to present a user interface; the resources (HTML texts) are downloaded from a server whenever needed. (However, this approach with "ultra-thin" clients becomes less credible the fatter the Web browsers themselves become.)

The BlackBox Component Framework uses its standard document format to store resources. For example, a dialog box and its controls are simply parts of a compound document stored in a file. The same view is used for layouting and for actually using a dialog box - no separate layout editor progam is needed. Resources can be available in several language versions simultaneously, and languages can even be switched at run-time. This is useful for customs applications or other programs used in regions where several languages are spoken.

2 Compound Documents

In the previous sections, we have discussed a variety of problems that occur with interactive systems, and we have seen various architecture and design approaches that help to solve them.
In the following sections we focus specifically on the programming problems posed by compound documents, and the design patterns that help to solve them.

2.1 Persistence

We call the object that is contained in a window a *view*. A view displays itself in the window, and it may interpret user input in this area, such as mouse clicks. In the most general case, a view can be opened and saved as a document. Other views may be created at run-time or are loaded from resource files. The contents of a "Find & Replace" dialog box is an example of the latter case.

In general, a view is a persistent object. This presumes a mechanism for the externalization (saving) and the internalization (opening) of a view. Using object-oriented programming, we can model a view as a specialization of a generic persistent object, which we call a *store*. A store can externalize and internalize its persistent state. A view is a specialized store, i.e., a subtype. In addition to externalizing and internalizing themselves, views can display themselves and can handle user input. These generic views can be further specialized, for example to text views, picture views, and so on (see Figure 2-1).

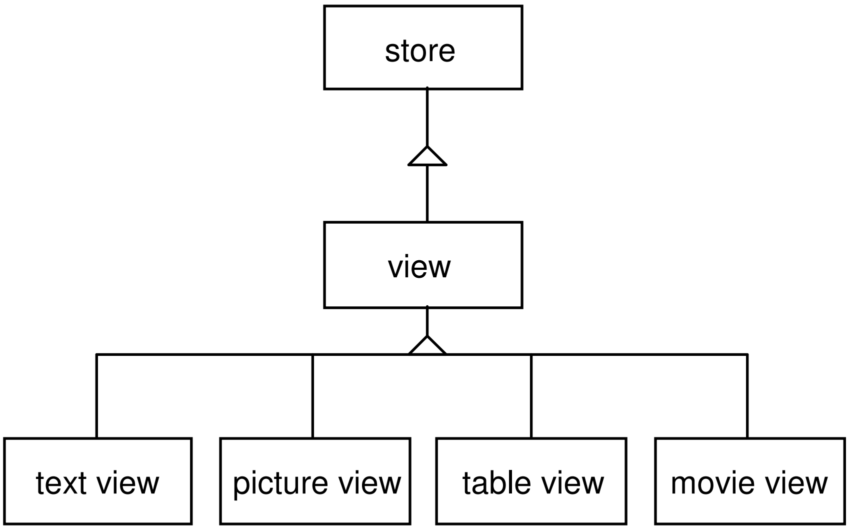


Figure 2-1. Subtype relations between views and stores

In the BlackBox Component Framework, stores are represented as type *Store* in module *Stores*; views are represented as type *View* in module *Views*; and there are various specific views such as type *View* in module *TextViews* or type *View* in module *FormViews* or type *Control* in module *Controls* (see Figure 2-2). This is only a small selection, actually there are many more view types. The store mechanism's file format is platform-independent, i.e., differences in byte ordering are compensated for.

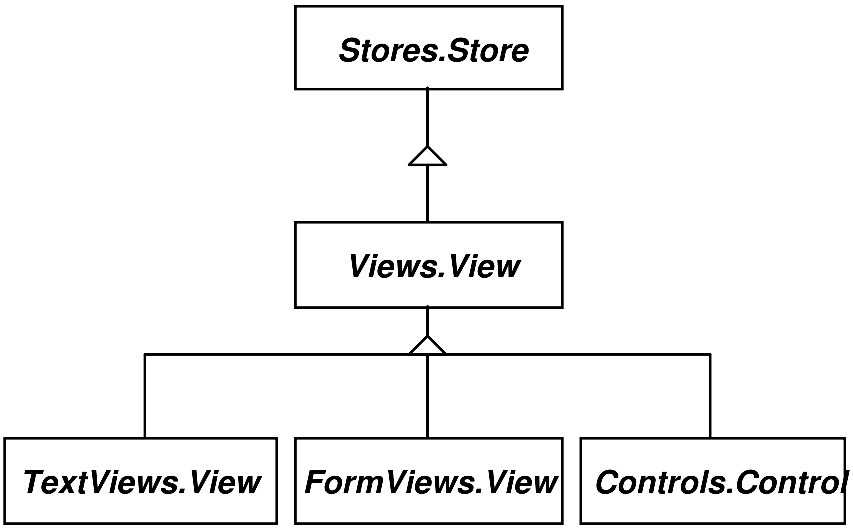


Figure 2-2. Subtype relations between views and stores in BlackBox

Often, copying of a store is equivalent to writing it to a temporary file, and then reading it in again (externalize old object to file, then allocate and internalize new object from file). This makes it convenient to combine persistence and copying support in the same store abstraction. However, BlackBox stores are able to change their identities or to "dissolve" upon externalization to a file, in which case copying is not equivalent to externalize/internalize anymore. For example, error marker views which indicate syntax errors remove themselves upon externalization - yet a visible error marker can be copied with drag & drop like any other view. To allow for such flexibility, BlackBox doesn't automatically treat copying as an externalize/internalize pair. Stores have explicit *Externalize*, *Internalize* and *CopyFrom* methods. Upon externalization, a store gets the opportunity to substitute another store for itself, possibly *NIL*. This is the mechanism which allows to change its identity to another one, a so-called *proxy*.

2.2 Display independence

The computing world is heterogeneous, with different hardware and operating system platforms on the market. The Internet, with its millions of platforms of all flavors, has finally made it clear how important platform-independence of code and data is. This is not easy to achieve, because software and hardware can vary widely between platforms. Display devices, such as monitors and printers, are a good example. Their spatial and color resolutions can differ enormously.
To achieve platform-independent code, views and their persistent state must not refer to specific devices. Instead, the *document space* and the *display space* must be clearly separated and a suitable mapping between the two must be provided by a framework. For example, in the document space, distances are measured in well-defined units such as inches or millimeters, while in the display space, distances are measured in pixels, whatever their actual sizes may be.

To model the separation between display and document space, where the document space is populated by persistent views, we need an abstraction of a display device. We call such an object a *frame*. A frame is a rectangular drawing surface which provides a set of drawing operations. When it is made visible, a view obtains a frame, and via this frame it can draw on the screen (or printer). The frame performs the mapping between the document space units and the pixels of the display device, so that views never need to deal with pixel sizes directly.

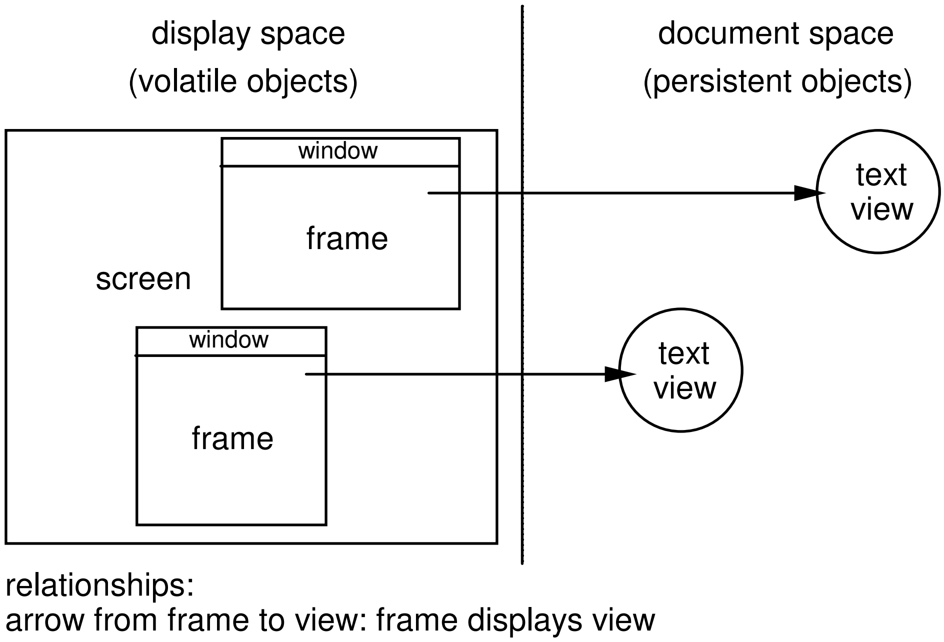


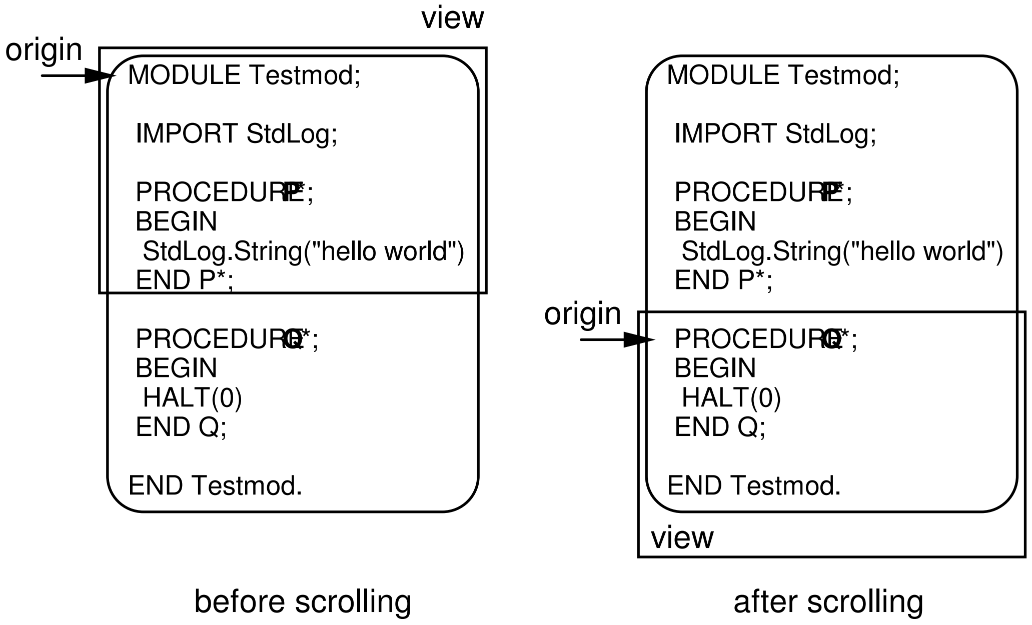
Figure 2-3. Separation of display and document space

In the BlackBox Component Framework, document space units are measured in 1/36000 millimeters. This value is chosen such that it minimizes rounding errors for typical display resolutions. A BlackBox frame provides a local coordinate system for its view. For the time being, you can regard a frame as a window and a view as a document. A view which isn't displayed has no frame. Frames are light-weight objects that come and go as needed. They are created and destroyed by the framework, and need not be implemented or extended by the view programmer.

The only case where it is necessary to extend frames is when platform-specific controls or OLE objects need to be wrapped as BlackBox views. Since writing such wrappers is a messy business, the framework provides standard OLE wrappers (Windows version) and wrappers for the important (pre-OLE) standard controls.

2.3 Multi-view editing

Simple views work fine if they are small enough to fit on a screen. However, texts, tables, or graphics can easily become larger than a screen or printed page. In this case, a frame (which lies strictly within the boundaries of its display device) can only display part of the view. To see the other parts, the user must *scroll* the view, i.e., change the way in which it translates its data to its local coordinate system. For example, a text view's first line (its origin) may be changed from the topmost line to another one further below, thus displaying a text part further below (see Figure 2-4).



```
MODULE Testmod;

 IMPORT StdLog;

 PROCEDURE P*;
 BEGIN
  StdLog.String("hello world")
 END P*;

 PROCEDURE Q*;
 BEGIN
  HALT(0)
 END Q*;

END Testmod.
```

Figure 2-4. Scrolling a text view

The need for scrolling can become very inconvenient when working with two parts of a view that are far apart, because it means that the user often must scroll back and forth. This problem can be solved by *multi-view editing*: the same data is presented in more than one view. The views may differ in their origins only, or they may differ more thoroughly. For example, one view may present a list of number values as a table of numbers, while another view may present the same data as a pie chart (see Figure 2-5).

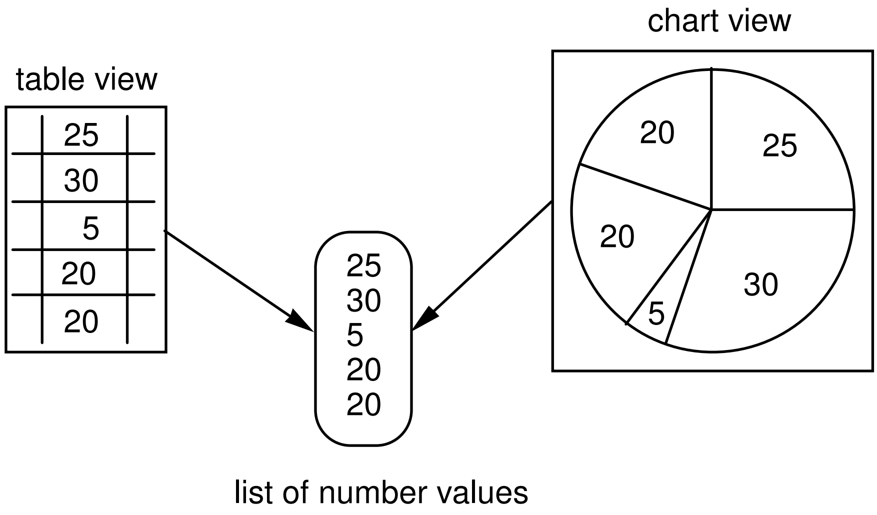


Figure 2-5. Different kinds of views presenting the same data

An object which represents data that may be presented by several views is called a *model*. The separation into view and model go back to the Smalltalk project at Xerox PARC. In the patterns terminology, the view is an *observer* of the model.

In the BlackBox Component Framework, the model/view separation is optional. Small fixed-size views that easily fit in a window usually don't need separate models, while larger or more refined views do. This can result in a situation like the following one, where a view has a separate model, and both together form the document which is displayed in a window:

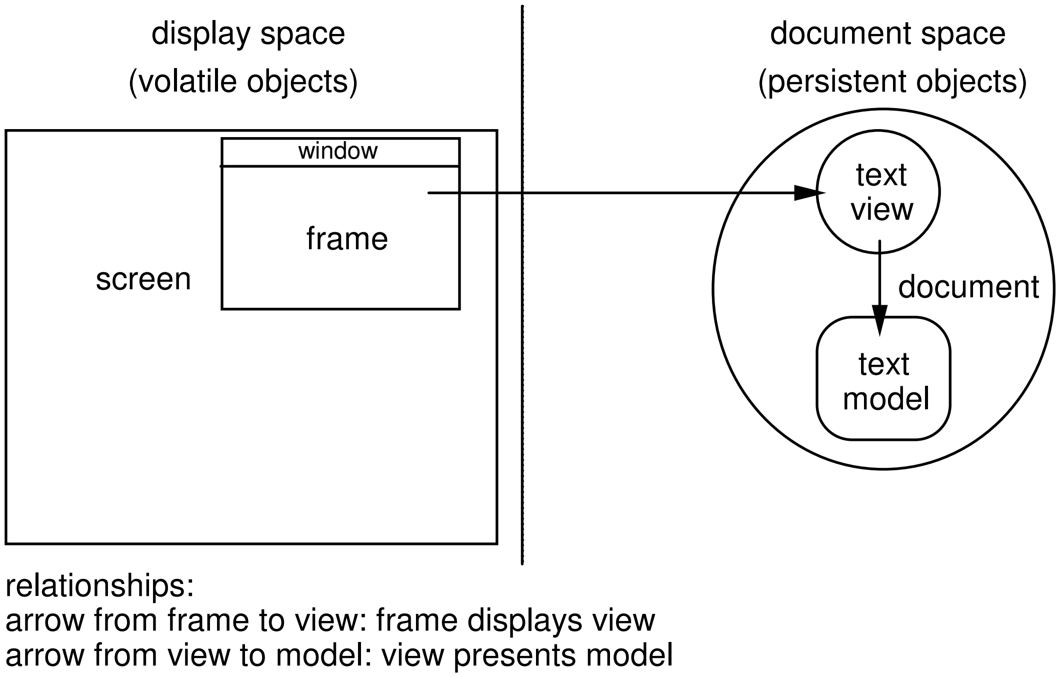


Figure 2-6. Separation of models and views in BlackBox

The user can open a second window, which displays its own view, but for the same model. This results in the following situation:

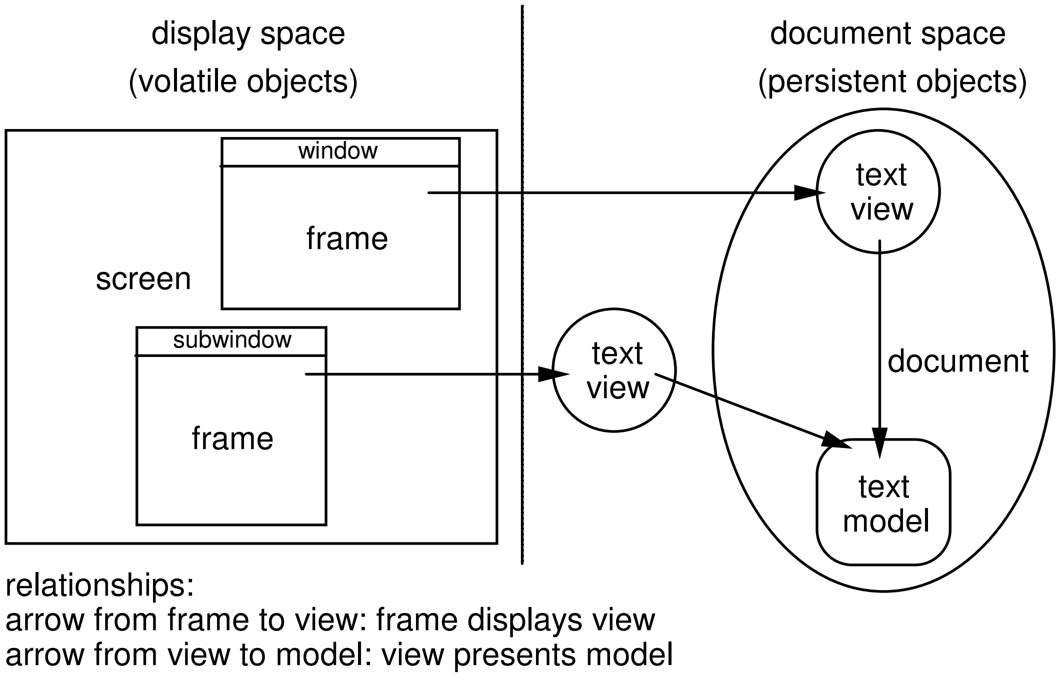


Figure 2-7: Multiple views for the same model

Only one of the two views is externalized when the document is saved in a file. To avoid schizophreny, one window is clearly distinguished as the main window, while all others are so-called *subwindows*. If you close a subwindow, only this window is closed. But if you close the main window, it and all its subwindows are closed simultaneously.

2.4 Change propagation

Multi-view editing creates a consistency problem. Editing a view means that the view causes its model to change. As a result, *all* views presenting this model must be updated if necessary, not only the view in which the user currently works. For example, if the user changes a value in the table view of Figure 2-5, then the pie chart in the other view must be updated accordingly, because their common model has changed.
Such a change propagation is easy to implement. After the model has performed an operation on the data that it represents, it notifies all views which present it, by sending them an *update message* describing the operation that has been performed.
Notification could be easily done by iterating over a linear list of views anchored in the model. Such a functionality can be provided once and for all, by properly encapsulating it in a class. This change propagation class provides an interface consisting of two parts. The first part allows to register new views that want to listen to update messages ("subscribe"). The second part allows to send update messages ("publish"):

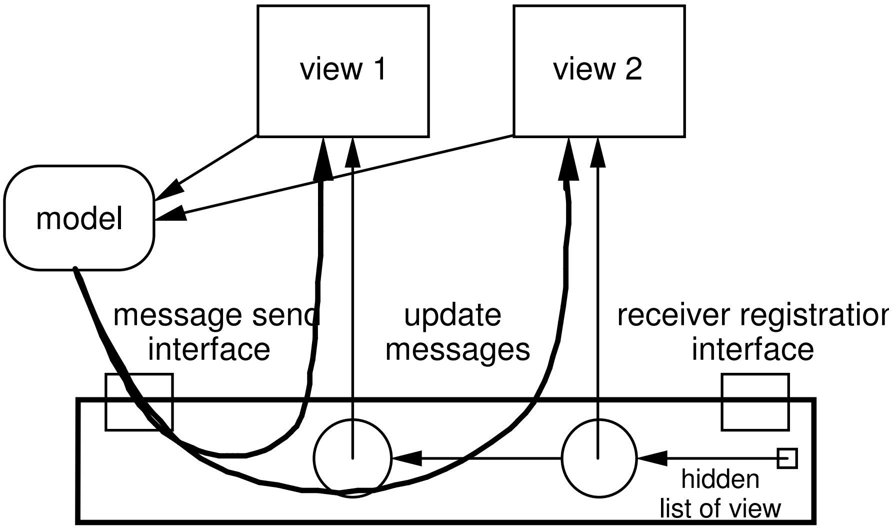


Figure 2-8. Change propagation mechanism

This mechanism has an important advantage: it decouples model and views such that the model itself need not be aware of its views, and the list of views can be encapsulated in the change propagation mechanism. Such a reduced coupling between model and views is desirable because it reduces complexity. It makes the model unaware, and thus independent, of its views. This is important in an extensible system, because it allows to add new view types without modifying the model. This would not be possible if the model had to know too much about its views.

Sometimes the mechanism described above does too much. It sends update messages to all views, independent of whether they need updating or not. For example, when a word was deleted in a text model, then some text view may display a part of the text which is not affected by the deletion at all. This view need not do anything. Of course, all the views that *do* display the affected text stretch must be updated accordingly.

For this reason, the BlackBox Component Framework sends update messages only to those views which have frames. Since every frame knows its view, the change propagation mechanism actually registers the frames, not the views. Even better, the framework already has a list of frames for window management, so it can reuse this list. Thus, instead of having one change propagation object per model, BlackBox uses one global change propagation service integrated into the window manager. Messages are represented as static message records. This is an unconventional but robust, efficient, and light-weight implementation of the observer pattern.

2.5 Nested views

So far, we have assumed that a view (possibly with a model) is a document, and a frame is a window. Actually, things are a bit more difficult. The reason is that views may be nested. Some views are able to contain ("embed") other views; for this reason they are called container views. Some containers are very general, and allow to embed arbitrary numbers and types of other views. For example, a text view may embed table views, bitmap views, and so on. It may even embed another text view, meaning that arbitrarily deep nesting hierarchies can be created.

How does this affect what we have said so far? Obviously, we need to generalize our notion of views and frames. A view can contain other views, which can contain still other views, etc.; i.e., views become hierarchical. In principle, those hierarchies need not even be trees, they can be arbitrary directed acyclic graphs (DAGs). This means that several views in a document may refer to the same model, but no references may go "upwards" in the document hierarchy, because this would lead to an endless recursion when drawing the document, like a TV which displays itself.

The frames represent a subset of the view hierarchy. Since a frame is always completely contained within its parent frame, the frame hierarchy is strictly a tree.

A document can now be regarded as the root of a view DAG, and a window can be regarded as the root of a frame tree.

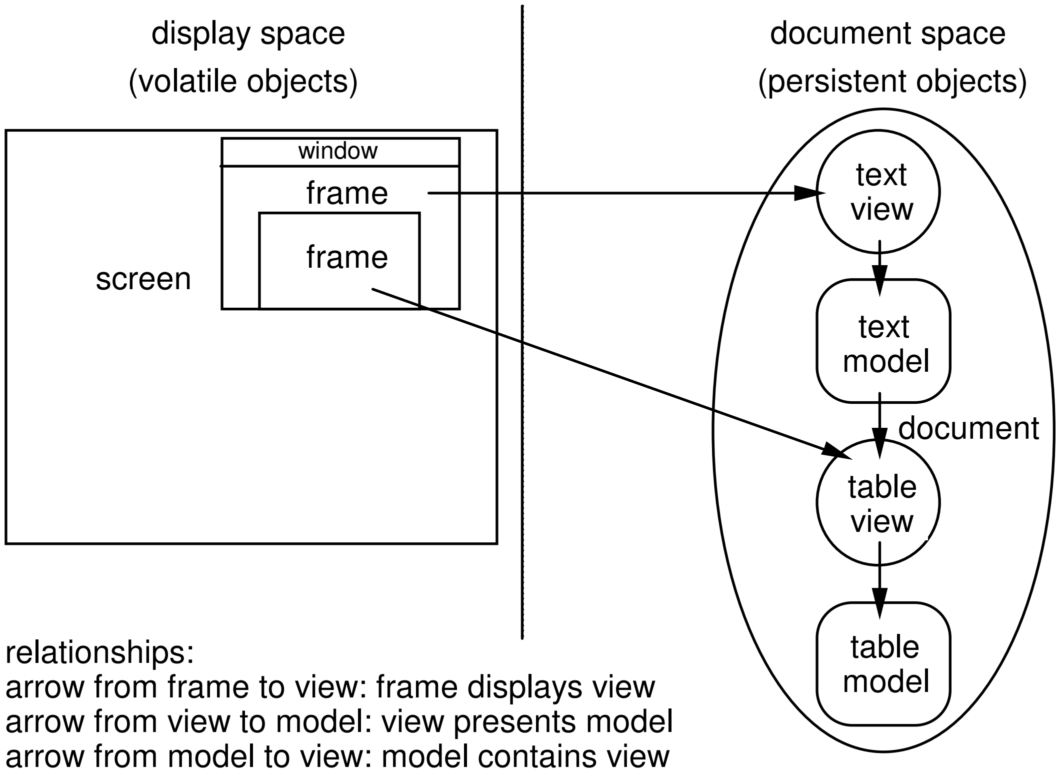


Figure 2-9. Nested views and frames

This is a straight-forward generalization which doesn't add unexpected complexity. It is a simplified implementation of the *composite* design pattern. It was simplified because the type safety of Component Pascal makes it unnecessary to burden the abstract view type with container-specific operations that are meaningless for non-containers.

Unfortunately, more complexity comes in when we consider the combination of nested views and multi-view editing. First of all, multi-view editing means the separation of views and models. An embedded view belongs to the model, since it is part of the data that the container view presents. For example, a user who edits a text view may delete embedded views just like she may delete embedded characters. Figure 2-10 illustrates the situation of two container views with a container model in which some other view is embedded:

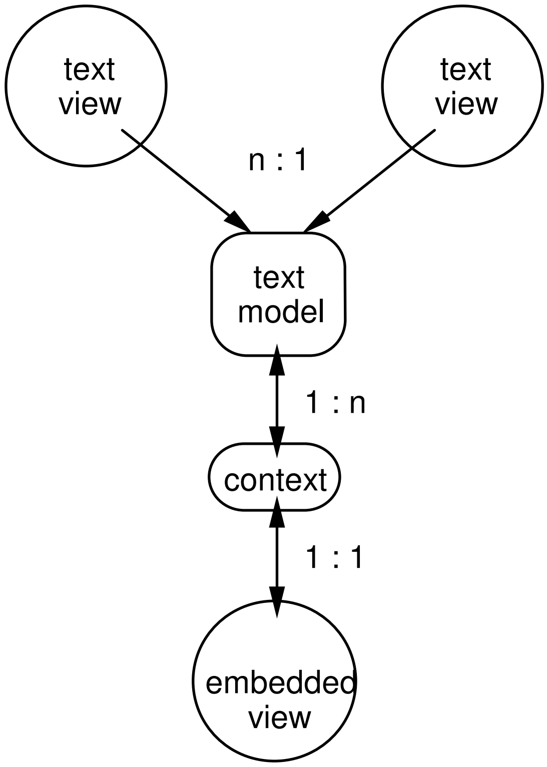


Figure 2-10. Container views with model, context, and embedded view

Note the interesting link between a model and an embedded view: the *context* object. It describes the location and possibly other attributes of one embedded view. The context's type is defined by the container. A context is created by the container when the view is being embedded. The embedded view can ask its context about its own location in the container, and may even obtain further information. For example, text container contexts may provide information about the embedded view's position in the text (character index), about its current font and color, and so on.

An embedded view may use its context object not only for getting information about its container, it may also cause the container to perform some action. For example, the embedded view may ask the container, via the context object, to make it smaller or larger. Containers are free to fulfill such requests, or to ignore them. In negotiations between a container and one of its embedded views, the container always wins. For example, a container doesn't allow an embedded view to become wider than the container itself.

A context object can be regarded as a call-back interface which the container installs in the embedded view as a plug-in, so the view can call back the container and obtain services from it. A context is part of a notification mechanism that allows the container to observe messages from the contained views.

In subsequent figures, context objects are not drawn expliclity in order to make the figures easier to understand. Just remember that every view carries a context that it can use when necessary.

Now that nested views and the separation of model and view are supported, it becomes possible to allow subwindows on embedded views, not only on root views. This can be quite helpful, if a small view must be edited. Just open a subwindow for it, enlarge the window, and editing becomes much easier.

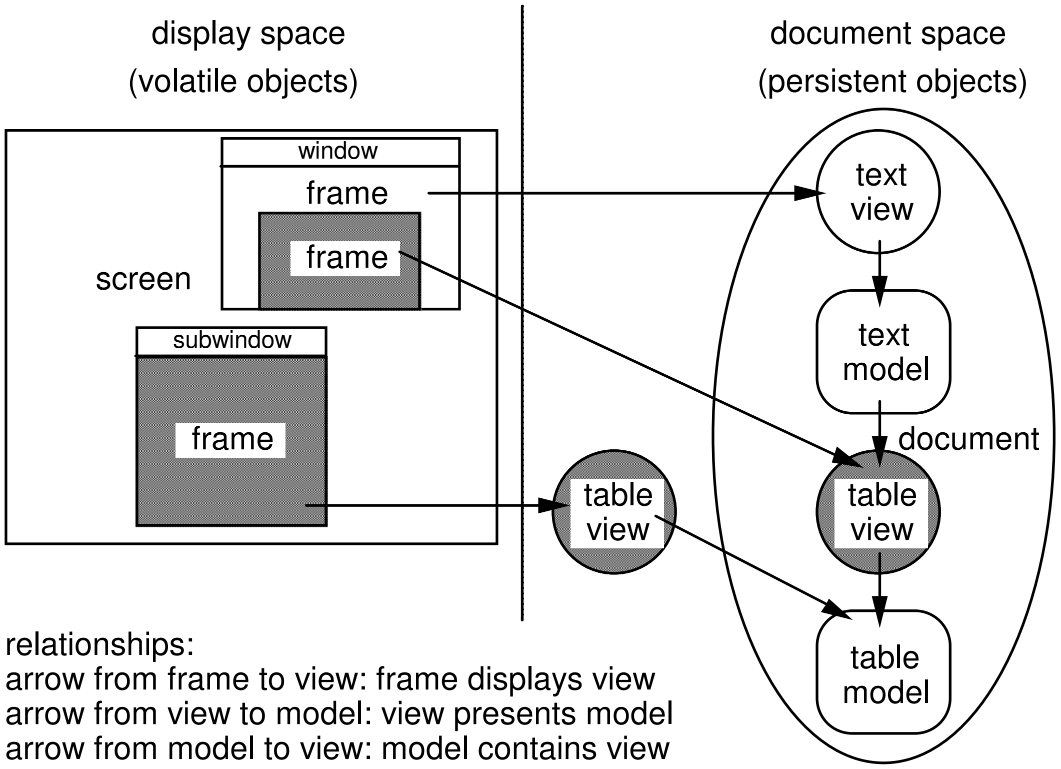


Figure 2-11. Subwindow on embedded view

Compound documents introduce a further complication. For root views, scrolling state is not retained when the document is saved to disk. At least most users seem to prefer that the scrolling state of a view not be persistent. This is different for embedded views, though. Setting the origin of an embedded view is not so much for convenience (subwindows are better for that) as for publishing: the user chooses which part of the view should be visible in the container.

The same holds for an undo/redo mechanism: scrolling in a root view is not a true document editing manipulation, and thus undo/redo would add inconvenience rather than usefulness. However, the scrolling of an embedded view must be treated as a genuine editing operation, and thus should be undoable/redoable.

In summary, this means that the implementor of a view must treat non-persistent view state differently, depending whether the view is a root view or an embedded view.

Combining nested views with multi-view editing also destroys the one-to-one relationship between frames and views. As we will see shortly, this is the most far-reaching additional burden that a view implementor must handle.

To see why a view may be displayed in several frames, consider the situation in Figure 2-12:

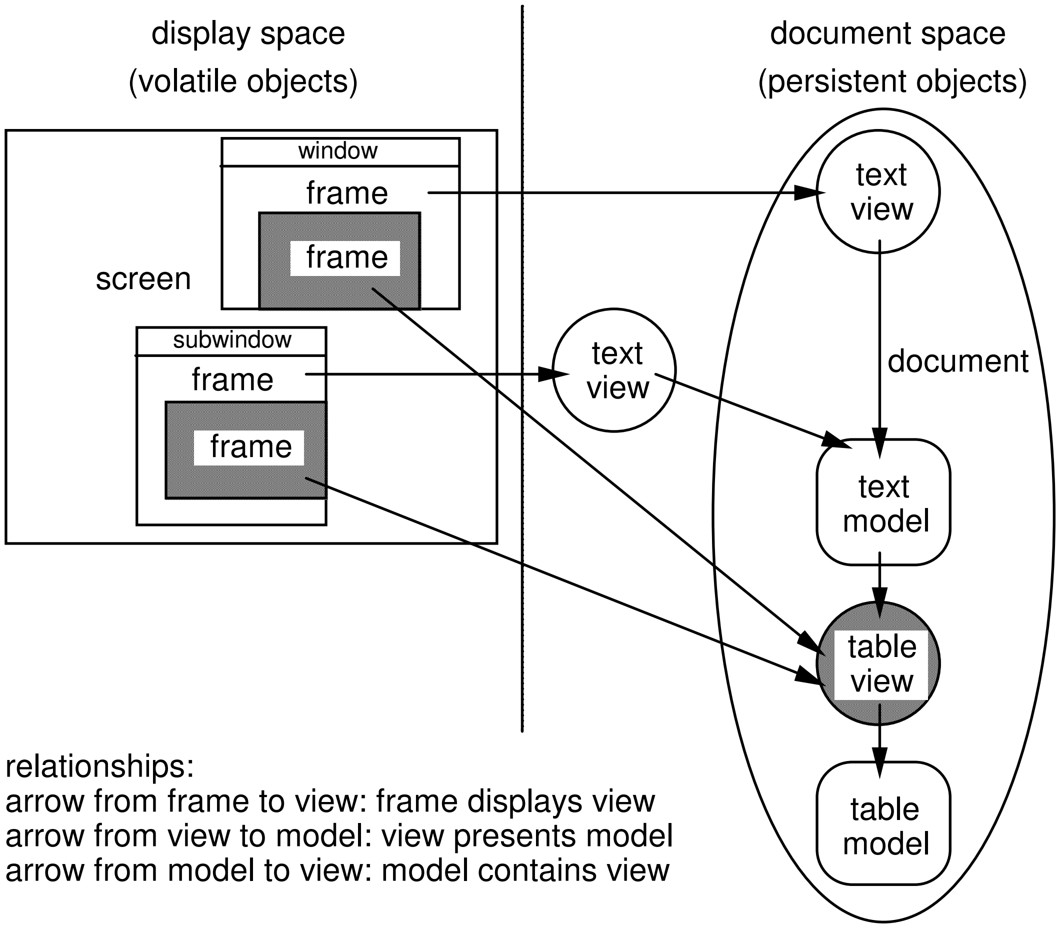


Figure 2-12. Several frames displaying the same view

Such a situation cannot occur without nested views, because the outermost view is always copied (to allow independent scrolling, or other view-specific operations). But whenever a view is embedded in a container displayed in two different windows, then this view is displayed in two different frames. This makes the management of links between frames and views more complicated; but more importantly, it makes change propagation and screen updates more complicated.

Why? Let us assume that there are n frames which display the same view. Now the view's model is changed, and this change is propagated to all views which display the model. Since there is only *one* view for the model, the change propagation mechanism as we have discussed earlier will send the view an update message exactly once. However, there are n places on the screen which need updating. As a result, the view must perform n updates, by iterating over all frames that display it.

For a view, correctly maintaining the list of frames in which it is displayed is a complicated endeavour. Therefore, the BlackBox Component Framework relieves the view implementor of this error-prone task. A BlackBox view doesn't contain a frame list. Instead, the framework maintains the frame trees automatically. This was the reason that in the above figures, the arrows from frames to views only pointed in one direction, because in BlackBox the view doesn't know its frame(s). The correct frames are passed to a view whenever the framework asks the view to do something that may involve drawing. (This approach is related to the *flyweight* pattern.)

When the view receives an update message and decides that it needs to update its frame(s), it simply asks the framework to send another update message to all its frames. Each of these frames will then ask the view to update the frame where this is necessary. This mechanism will be discussed in more detail in the chapters on view construction.

Since there is no more one-to-one relationship between views and frames anymore, it is helpful to summarize the differences between frames and view:

aspect	**frame**	**view**
lives in	display space	document space
persistent	no	yes
extended by view implementor	usually not	yes
may contain device-specific data	yes	no
hierarchy	tree	DAG
associated with	one view	0..n frames
update caused by	its view	its model (if any)
completely enclosed by container	yes	not necessarily

Table 2-13. Differences between frames and views

2.6 Delayed updates

In the previous section we have seen how updates are performed in two steps: a model notifies its views of a change, and the views notify their frames of the region which needs updating.

Now consider a complex model change. For example, a selection of graphical objects, which are part of a graphics model, are moved by some distance. It would be possible to perform an update in three phases: in the first phase, the selected objects would be removed (requiring an update); then the selected objects would be moved; and finally, the selected objects would be inserted again at their destination (requiring another update).

This approach is not very efficient, since it requires two notifications and partial updates. Furthermore, it needs careful sequencing of the actions, since some of them need the model's state prior to movement, and others need the model's state after movement.

There is a much simpler and more efficient approach. The idea is to delay the second phase of the updates, i.e., the phase where the frames for a view are redrawn. Instead of updating the frames immediately, the view only remembers the geometrical region which needs updating. In the above example, it could add the areas of the graphical objects prior to movement and their areas after movement (see Figure 2-14).

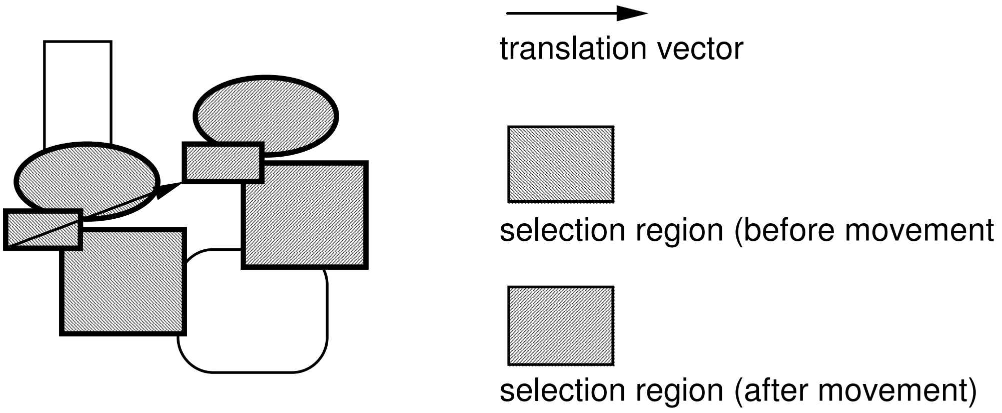

translation vector
selection region (before movement
selection region (after movement)

Figure 2-14. Update region of a selection which was moved

The accumulated update region is then updated in one single step after the translation of the graphical objects has been performed. In the figure above, the update region is the total hatched area, consisting of the area occupied by the selection before the movement and the area occupied by it after the movement.

The framework must support this approach by providing some mechanism to add up geometrical regions, and to redraw a frame in exactly this region ("clipping" away all unnecessary drawing, to increase performance and to minimize flicker).

This delayed or *lazy updating* approach is used by the BlackBox Component Framework and all major windowing systems that exist today. Its only drawback is that for long-running commands, it may be necessary to force intermediate updates in order to inform the user of the command's progress. For this reason, forced updates are also supported by the BlackBox framework.

2.7 Container modes

To create graphical user interfaces, it is convenient to create form layouts interactively, rather than burning in the coordinates of controls in the source code of a program. A visual designer, i.e., a special-purpose graphics editor can be used for interactive manipulation of form layouts. This suggests two different modes in which a form, and its controls, can be used: design-time and run-time. When a form layout is edited in a visual designer, it is *design-time*: the form is designed, but not yet used. The completed form can be saved and later opened for use by the end user. This is called *run-time*.

There are two different approaches to handling layouts at run-time: either the framework allows to open the data that has been saved by the visual designer, or it runs code that has been generated out of this saved data. The first approach requires an interpreter for the visual designer's data, the second approach requires a generator between the visual designer and actual use of a form. Typically, the generator generates source code, which is possibly completed by the programmer, and then compiled into machine code using a standard programming language compiler.

Today, the latter approach should be considered obsolete, although it is still used by many tools: it is merely an inconvenient detour. Moreover, when it forces programmers to edit the generated source code, it makes iterative changes to a layout problematic: the source code has to be regenerated without losing the changes that have been made by the programmer. This is a needless source of inconsistencies, complexity, and it slows down development.

The former approach is much simpler. The visual designer's output data is treated as a resource that can be utilized *as is* at run-time. This requires an interpreter for the visual designer's file format.

The most elegant approach is to use the same editor (i.e., view implementation) and thus the same file format both for design-time and run-time. As file format, the standard format of a compound document can be reused. This means that a form layout at design-time is a completely ordinary compound document. At run-time, it is still the same document, albeit its interaction with the user is different.

At design-time, a control is a passive box which can be moved around, copied or deleted, but which has no interactive behavior of its own. At run-time, a control may be focused and the control's *contents* may be edited. For example, the string contained in a text field control may be edited, or a button may be pressed and released again.

The direct use of compound documents as user interfaces is called *compound user interfaces*. Note that this generalization of compound documents renders the term "document" rather misleading, because an application may contain many forms (and thus compound user interface documents) even if it does not manipulate any documents in the traditional sense (i.e., as seen by an end-user).

The BlackBox Component Framework utilizes its support for multi-view editing to go one step further. Since a document can be opened in several views simultaneosly, it is possible to open a form layout in two windows. BlackBox allows to use form views in different modes: layout and mask mode. These modes correspond to design-time and run-time, with one major improvement: they can be used simultaneously. For example, one window may display a form in layout mode, and another window may display it in mask mode (see Figure 2-15). When the developer edits the layout in the layout mode window, any layout change is immediately reflected in the mask mode window. Vice versa, user input in the mask mode window immediately becomes visible in the layout window. This is a result of the standard change propagation mechanism of BlackBox.

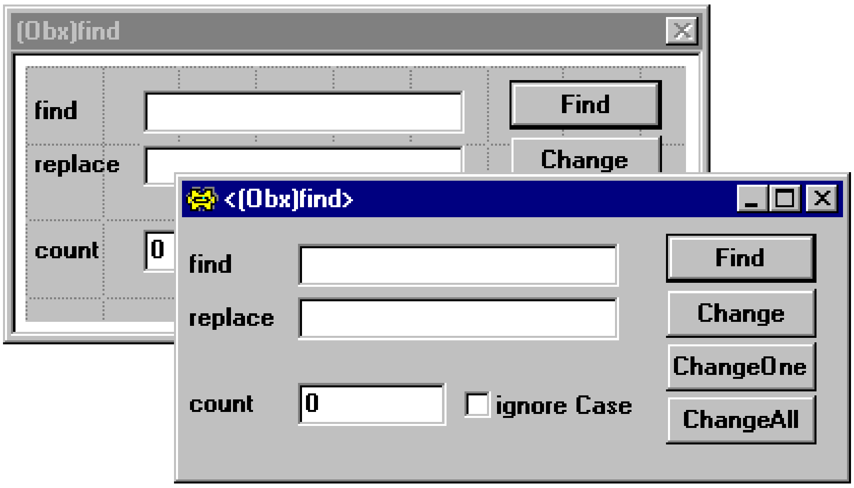


Figure 2-15. Data entry form in a layout-mode view (behind) and in a mask-mode view (front)

In fact, BlackBox is even more general. In addition to layout and mask mode, it supports a few further modes. The *edit* mode allows to modify the layout and the contents of controls simultaneously, and a *browser* mode makes the form read-only except for allowing to select and copy out parts of the document.

These general modes are not only available for graphical forms. They are a capability shared by *all* general container views, including text views. This means that it is possible (and sometimes more convenient for the developer) to use a text view instead of a form view when assembling a dialog box. The user won't notice the difference. To simplify the construction of such powerful containers, the framework provides a comprehensive container abstraction, with an abstract container model class, an abstract container view class, and an abstract container controller class. A controller can be regarded as a split-off part of a view. It performs all user interaction, including handling of keyboard input and mouse manipulations, and it also manages selections. There is a 1:1 relation between view and controller at run-time. Separating part of a view into a controller object improves extensibility (different controllers could be implemented for the same view, and the same controller works with all concrete view implementations) and it also allows to reduce complexity: the controller of a complex view, such as a container view, can become large, which makes the separation of concerns into different types (and typically into different modules) a good idea. In terms of design patterns, a controller is a *strategy* object.

The container abstractions of BlackBox have been designed to abstract from specific container user interfaces. This means that details of the container look-and-feel (such as the hatched focus marks of an OLE object) are hidden from the developer. In turn, this makes it possible to implement containers in different ways for different platforms, without affecting the developer of containers. In particular, the OLE and Apple OpenDoc container look-and-feel have been implemented. While OpenDoc isn't relevant anymore, some of the human interface guidelines developed for it are still useful today.

2.8 Event handling

When a user interacts with a compound document, this interaction always occurs in one view at a time. This view is called the current *focus*. By clicking around, the user can change the current focus. Among other events, the focus handles keyboard events, i.e., a keypress is interpreted by the focused view. The focus view and its containing views are called the *focus path*, with the innermost view being the focus.

It is possible to let a user interface framework manage the current focus. This requires a central manager for the focus. A more light-weight approach is to make focus management a decentral activity: leave focus management to the individual container views, which have to deal with focus changes and focus rendering anyway. Every container simply remembers which of its embedded views is the current focus, if any. The container is oblivious whether this view is on the current focus path or not.

In BlackBox, user events are sent along the focus path as message records, starting at the outermost view. Messages records are static (stack-allocated) variables of some subtype of *Controllers.Message*. There are controller messages for keyboard input, mouse tracking, drag & drop, and so on. A controller message is forwarded along the focus path until one of the views either interprets it or discards it. This view by definition is the focus.

In terms of design patterns, the focus path is a *chain of responsibility*.

What happens when the focus view receives and interprets a controller message? The view (or its model, if it has one) performs some operation on its own state. If this operation affects persistent state of the view or model, it should be reversible ("undoable"). How is an undo/redo mechanism implemented?

The main idea is that upon receipt of a controller message, the view / model doesn't perform a state modification directly. Instead, it creates a special *operation object* and registers it in the framework. The framework then can call the operation's appropriate procedure for performing the actual do/undo/redo functionality.

Operations are managed per document. Every document contains two operation stacks; one is the undo stack, the other is the redo stack. Executing an operation for the first time pushes the object on the undo stack and clears the redo stack. When the user performs an undo, the operation on top of the undo stack is undone, removed from the undo stack, and pushed onto the redo stack. When the user performs a redo, the operation on top of the redo stack is redone, removed from the redo stack, and pushed onto the undo stack.

A document's undo and redo stacks are cleared when the document is saved (check point). Furthermore, they may be cleared or made shallower when the framework runs out of memory. For this purpose, the garbage collector informs the framework about low-memory conditions, so old operation objects can be thrown away.

Some operations, such as a moving drag & drop (in contrast to the normal copying drag & drop), modify two documents simultaneously. In these rare cases, two operations are created: one for the source document (e.g., a delete operation) and one for the destination document (e.g., an insert operation).

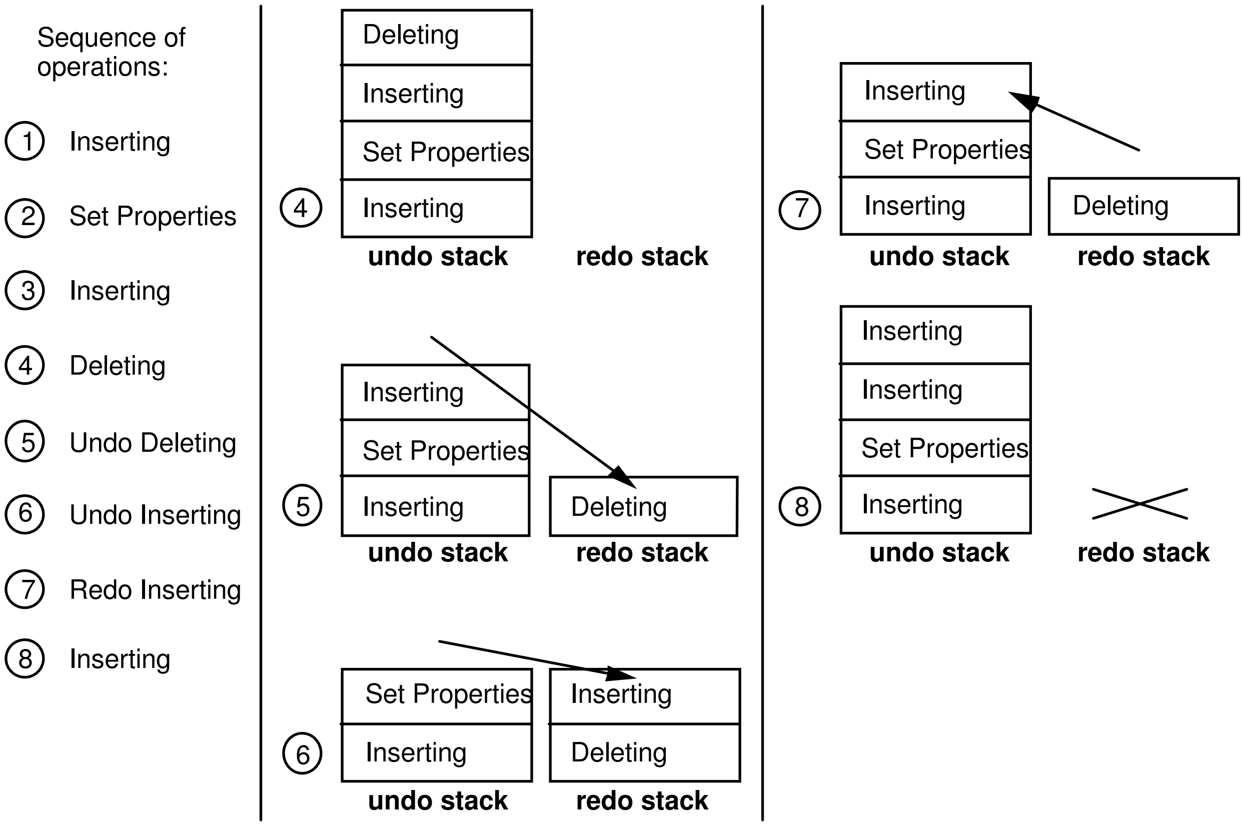


Figure 2-16. Sequence of operations and resulting modifications of undo and redo stacks.

In Figure 2-16, a sequence of operations is shown, from 1) to 8). For the last five situations, the resulting undo and redo stacks are shown. For example, after operation 5), the undo stack contains the operations Inserting, Set Properties, and Inserting (from top to bottom of stack), while the redo stack contains Deleting.

The undo/redo mechanism is only concerned with the persistent state of a document. This is the state which can be saved in a file. Modifications of temporary state, such as a view's scroll position, are not recorded as operations, and thus cannot be undone (at least for root views).

Undoable operations either modify the persistent state of a view, or of its model (controller state is mostly temporary). The framework knows to which document a persistent object belongs, because all persistent objects of a document share the same domain. Domains will be discussed in Part III where the store mechanism is discussed in more detail.

The BlackBox Component Framework is unique in that its undo/redo mechanism is component-oriented: it allows to compose undoable operations into *compound operations*, which are undoable as a whole. Without this capability, nested operations would be recorded as a flat sequence of atomic operations.

Consider what this would mean for the end user. It would mean that the user could execute a menu command, which causes the execution of a hierarchy of operations. So far so good. But when the user wanted to undo this command, he or she would have to execute Edit->Undo individually for every single operation of the command, instead of only once.

For this reason, BlackBox provides support for arbitrarily nested operations: modules *Models* and *Views* both export a pair of *BeginScript* / *EndScript* procedures. In this context, "script" means a sequence or hierarchy of atomic operations which is undoable as a whole. Model and view operations can be freely mixed in a script.

Abstract operations are very light-weight. They only provide one single parameterless procedure, called *Do*. This procedure must be implemented in a reversible way, so that if it is executed an even number of times, it has no effect. If it is executed an odd number of times, it has the same effect as when it is executed once.

In the design patterns terminology, an operation is called a *command*.

2.9 Controls

Controls are light-weight views that only provide very specific functionality, which only makes sense in concert with other controls and a container. Typical controls are command buttons and text entry fields. They are used in dialog boxes or date entry forms. The parameters or other data being represented by a control must somehow be manipulated by a corresponding program. For example, a *Find* command takes a search string as input, or the data entered into a form must be sent to an SQL database. There may even be complicated interactions between the controls of a form. For example, when a search string is empty, the *Find* button may be disabled. If something is typed in, the button is enabled. Such interactions can become very involved.

In principle, a control is simply an observer view on its data. This means that it would be most straight-forward to implement the data (e.g., the search string) as a model.

For BlackBox, it was felt that this approach is too heavy-weight and unconvenient. Instead, a special implementation of the observer pattern was realized, where typical applications never need to access control objects directly. They are completely "abstracted away". The programmer only deals with the observed data. In order to make the definition and manipulation of this data as convenient as possible, it is simply a global Component Pascal variable, called an *interactor*. A control has a symbolic name (e.g. "TextCmds.find.ignoreCase") which enables it to find its variable, using built-in metaprogramming facilities of BlackBox. In the interactor's module itself, there is no object reference to a control.

When a control is being edited, it transparently uses the standard change propagation mechanism of BlackBox, i.e., it broadcasts view messages that notify other controls which may display the same variable.

Although the interactor's module doesn't have access to its controls, it can still influence the way controls are displayed (e.g., enabled or disabled) and the way they interact with each other. This is done by exporting suitable *guard* and *notifier* procedures. They are attached to a control in the same way as its interactor link, using a suitable control property editor. The important thing is that the program need not access the controls directly, and no direct control-to-control interactions need to be programmed. This greatly simplifies the implementation and maintenance of complex user interface behaviors. The mechanism is described in more detail in Chapter 4.

In principle, the framework accesses the interactor's module (via metaprogramming) as a *singleton mediator*.

3 BlackBox Design Practices

In the previous chapters, various patterns and design approaches have been described. They help to solve problems that occur in interactive compound document applications. In this chapter, more general aspects are discussed which help to make a framework component-oriented, i.e., extensible through dynamically loaded black-box components.

We don't attempt to create a full design method that gives step-by-step recipes for how to design a new framework. This would not be realistic. But we want to demonstrate and motivate the practices, approaches, and patterns that have been followed in the design of the BlackBox Component Framework. Some of the design practices have even led to the incorporation of specific framework-related features into the language Component Pascal. Language support for framework design is discussed first, followed by the way Component Pascal components are managed, and the rules which govern many component collaborations in BlackBox.

3.1 Language support

A framework embodies a collection of design patterns, cast into the notation of a particular programming language. Thus the expressiveness of the language, in particular of its interface definition subset, has a major impact on how much of the framework's design can be captured directly in code. Capturing architectural decisions and design patterns explicitly in the language is important for making refactoring of a framework less risky. Refactoring of a framework and its extension components allows to prevent that old architectures grow into brittle structures that threaten to collapse under their own weight. Preventing architecture degradation is the key to keeping software systems productive over a longer period of time, especially if the software consists of components that are replaced, added, or removed incrementally over time.

A framework represents an architecture for solving a certain class of problems. A good architecture uses a minimal number of design patterns wherever they are applicable. Consistent use of design patterns make the framework easier to document and easier to comprehend.

Design patterns are abstract solutions to a problem. For a framework, they need to be formulated in terms of a concrete programming language. The programming language has a large influence on how well the design pattern can be represented and how well consistency with the pattern can be maintained.

There are two major "philosophies" of programming language design. One of them is exemplified by the language C. C is a terse systems programming language which allows to easily and efficiently manipulate memory data structures. The other approach is exemplified by Pascal. Pascal is a readable application programming language which provides a high degree of safety.

The C approach is particularly adequate for writing low-level code such as device drivers. When C was increasingly being used for applications as well, a *zero errors ideology* formed, which says that because programming errors must not occur, they will not occur. A "real programmer" doesn't make mistakes, and therefore doesn't need any kind of protection facilities forced upon him by the language. Safety features such as index overflow checks are for beginners only; for professionals they are merely a handicap.

However, modern psychological research has clearly shown that humans make mistakes all the time, whether they write programs, develop mathematical proofs, fly airplanes, or perform operations on a patient. Mistakes are made whether or not they are admitted. The important insight is that most mistakes can be corrected easily if they are detected early on. Detection works best in an openly communicating team, where a team member is not afraid of others double-checking his or her work (and thereby exposing the mistakes). Good airlines let their pilots train such cooperative behavior under stress. Some very forward-thinking clinics use a similar training for surgeons and their aides. They have overcome the zero errors ideology in their fields.

In the world of programming, a reverse trend has shaped the industry in the last ten years, by making C the language of choice for all kinds of programs. In terms of software engineering and safety consciousness, this was a huge step backwards behind the state-of-the-art. Large companies tried to limit the damage by imposing the use of tools that reintroduce at least a modest level of safety - instead of solving the problems where it costs least, namely at the language level.
But while it is difficult for a programmer to admit that he makes mistakes, it is much easier for him to acknowledge that *other* programmers make mistakes. This became relevant with the Internet. The Internet makes it easy and sometimes even automatic to download and execute small programs ("applets") from unknown sites. This code clearly cannot be trusted in general. Foor good reason, this has scared large corporations enough to look at safer languages again. In fact, safety concerns were the reason why the language Java was created in the first place. Superficially, Java looks similar to C++; but unlike C and C++, it is completely typesafe, like Component Pascal.

In Component Pascal, objects and their classes (record types) may be hidden completely in a module, or they may be wholly exported, i.e., they and their parts (record fields / instance variables) may be made completely visible outside of the defining module. In practice, it is useful to have even more control. For this reason, Component Pascal allows to determine for each record field whether it is fully exported, read-only exported, or hidden. The following example (Figure 3-1) shows how the asterisk ("*") is used for export, and the dash ("-") is used for the more restricted read-only export:

```
MODULE ObxSample;

    TYPE
        File* = POINTER TO RECORD
            len: INTEGER    (* hidden instance variable *)
        END;

        Rider* = POINTER TO RECORD
            (* there may be several riders on one file *)
            file-: File;    (* read-only instance variable *)
            eof*: BOOLEAN;    (* fully exported instance variable *)
            pos: INTEGER    (* hidden instance variable *)
                (* Invariant: (pos >= 0) & (pos < file.len) *)
        END;

    PROCEDURE (f: File) GetLength* (OUT length: INTEGER), NEW;
    BEGIN
        length := f.len
    END GetLength;

    PROCEDURE (rd: Rider) SetPos* (pos: INTEGER), NEW;
    BEGIN
        (* assert invariants, so that errors may not be propagated across components *)
        ASSERT(pos >= 0); ASSERT(pos < rd.file.len);
        rd.pos := pos
    END SetPos;

    ...

END ObxSample.
```

Listing 3-1. Sample module in Component Pascal

Java does not support read-only export, but it does support protected fields, which are fields that are only visible to extensions of a class, but not to normal clients. This feature is only relevant if implementation inheritance is used across module boundaries. For reasons that go beyond the scope of this text, implementation inheritance is not a good idea across black-box abstractions, such as components, and thus should not normally be used across component boundaries or in component framework interfaces. Protected export is thus not supported by Component Pascal.

Safety means different things to different people. On the one hand, C++ can be considered safer than plain C. On the other hand, the Internet also raises concerns that go beyond mere safety; security has to be considered as well (i.e., unauthorized access, criminal attacks, and so on). Security issues are a matter of authorization and authorization checking and are generally dealt with at the operating system or hardware level. Higher-level security features are a matter of library definition and implementation. However, overheads incurred by the excessive crossing of hardware protection boundaries can be avoided if it is possible to build on strong safety properties of the used language(s), which leads us back to the question of safety. But what exactly is "safety"?

In short, "safety" of a programming language means that its definition allows to specify invariants, and that its implementation guarantees that these invariants are kept. (The availability of a trusted implementation must be assumed; this trust is usually earned by successfully surviving attacks.) In short: *safety is "invariants taken seriously"*.

What kinds of invariants are we talking about? The most fundamental invariants are the memory invariants: memory occupied by a variable is only used in the way that the language (e.g., a type declaration) allows. In practice, this means that arbitrary type casting must be ruled out and that manual deallocation of dynamic data structures is not permitted anymore, because deallocation could happen too early, and some still used memory could be reallocated and thus being used simultaneously by two unrelated variables. These dangling pointers are usually desastrous, but can be avoided completely if garbage collection is used instead of manual deallocation. Consequently, Java and Component Pascal are garbage-collected.

Memory invariants are fundamental. More application-specific invariants are typically established over several cooperating objects. For example, an object which represents a file access path must always have a current position which lies within the length of the file, where the file is represented by a second object. This invariant spans two objects (and thus classes). It can only be guaranteed if the programming language allows to define interactions between the two objects that are private to them, so that no outside code may interfere.

Note that this is different from the "protected" relation discussed above. Rather than covering the relation between a base class and its yet unknown subclasses, the issue here is classes that have been co-designed and will always be used in conjunction. A module or package construct is a suitable structured means to allow the definition of such higher-order safety properties. Java packages are not ideal in this respect. Since Java packages are *open modules*, new classes can be added to a package at run-time, and may well violate the invariants that have been established earlier. Such addition of "foreign" classes to a package thus needs to be prevented by run-time management facilities that are beyond the control of the language definition. In Java, this is related to the concept of unique ownership: every package is conceptually owned by its source and only that single source should have authority to add new classes to a package. In any case, the Java package construct is an improvement over most other object-oriented languages, for example the entirely unstructured approach of C++'s friend classes, or the complete absence of a suitable construct in standard Smalltalk.

In contrast to Java packages, Component Pascal supports *closed modules*, or modules for short. In Component Pascal, a module is the appropriate unit of compilation, loading, and information hiding.

Applying strong typing and information hiding makes it easier to catch mistakes as early as possible, when they are still easy and inexpensive to correct. This is valuable because it helps increase the program's robustness.

But the benefits of type and module safety go even further. They also provide more *flexibility*. This is surprising at first sight. Why should restrictions such as types (which restrict the operations on variables) and modules (which restrict visibility) create anything else than *reduced* flexibility? The reason for this so-called *refactoring paradoxon* is two-fold. On the one hand, everything that is completely hidden in a module may be changed only by considering this one module. Local changes in the hidden part of the module don't reverberate beyond the module itself. Not even recompilation of other client modules is necessary. On the other hand, when a well-typed interface for some reason *is* changed, then mere recompilation of the clients will detect the interface usages that have become inconsistent, e.g., after a parameter's type or a method name was changed. A compiler thus can actually increase the confidence in a software system which has been "refactored" due to some interface changes.

By checking typed interfaces at a carefully chosen level of granularity, a balance can be struck between release-to-release binary compatibility and detection of definite inconsistencies. Component Pascal supports an expressive type system to allow the detection of such inconsistencies. For example, newly introduced methods must be marked as *NEW*, and *VAR* parameters can be specialized to *IN* or *OUT* parameters. In Java, for example, it is not possible to distinguish between a new method, an overloading attempt, and an overriding attempt. By misspelling a method name, by changing a base or a subclass, or by combining incompatible versions, this ambiguity can lead to errors that are hard to track.

A framework typically predefines interfaces for extensions of the framework. The framework may even contain code that uses these interfaces, although no implementation exists yet. The calling of code that resides "higher up" in the module hierarchy is typical for object-oriented frameworks. A language can support this typical framework control flow pattern by providing some form of interfaces, which are abstract classes to be implemented elsewhere. Component Pascal supports abstract record types with single inheritance, in a way that a complete spectrum between fully abstract and fully concrete types are possible. Java is similar to Component Pascal in this respect, except that it additionally supports a separate interface construct. A Java interface is the same thing as a fully abstract class, except that it allows multiple (interface) inheritance.

```
MODULE TestViews;

  TYPE
    View* = POINTER TO ABSTRACT RECORD
      (* partially abstract type *)
      context-: Context;
      ... some hidden fields ...
    END;

    Context* = POINTER TO ABSTRACT RECORD END;
    (* fully abstract type *)

    PROCEDURE (v: View) Restore* (l, t, r, b: INTEGER), NEW, ABSTRACT;

    ...

END TestViews.
```

Listing 3-2. Semi-abstract and abstract record types in Component Pascal

Invariants are properties that supposedly stay invariant. Thus it is questionable whether even a subtype (subclass / extended type) should be allowed to arbitrarily modify the behavior of its basetype. In object-oriented languages, such modifications can be achieved by overriding inherited methods. Java allows to make classes or methods *final*, to give the framework designer the possibility to prevent any kind of invariant violation through the back door of overriding. Component Pascal record types and methods are final by default, they can be marked as extensible explicitly.

Component Pascal goes beyond Java with several other language constructs. One are *limited* records. Those are records that may be extended and allocated only within their defining module. From the perspective of importing modules, limited types are final and not even allocatable. This makes it possible to guarantee that all allocation occurs centrally in the defining (framework) module, which gives this module full control over initialization. For example, it may provide factory functions that allocate an object and initialize it in different ways, establishing invariants before the objects are passed to client modules. This is more flexible and simpler than constructors as used in Java.

Implement-only export is a prime example of a feature motivated by typical framework design patterns. A record type's methods may be exported as implement-only, by using a dash instead of an asterisk. An implement-only method can only be called inside the defining module. But it can be *implemented* outside the defining module, in an implementation component of the framework. For example, the BlackBox Component Framework's store mechanism uses this feature to protect a store's *Internalize* and *Externalize* methods from being called out of their correct context. Basically, the framework (in this case the *Stores* module) uses the implement-only methods in all possible legal ways (i.e., implements all legal kinds of use-cases), and only exports them for implementation purposes.

For methods that represent optional interfaces, the method attribute *EMPTY* is supported. An empty method is a fully abstract hook that can be implemented in an extension, but unlike abstract methods it need not be implemented. For example, a view has an empty *HandleCtrlMsg* method which need only be implemented by views that react on user input, e.g., via mouse or keyboard.

In the *Design Patterns* book of Gamma et. al. ["Design Patterns, Elements of Reusable Object-Oriented Software"; Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; Addison-Wesley, 1994; ISBN 0-201-63361-2], a list of common design problems is identified. These problems often lead to unnecessary redesigns, because the software doesn't allow for a sufficient degree of change. Several of these problems are addressed by Component Pascal:

• Creating an object by specifying a class explicity

Modules allow to hide classes. A hidden class cannot be instantiated directly by a client module, since it is not visible there. This makes it possible to enfore a variety of indirect allocation mechanisms.

Abstract record types allow to separate interfaces from implementations, so that abstract record types can be exported, without risking direct allocation by clients.

Like abstract types, limited record types cannot be directly allocated by clients.

Implement-only export allows to restrict the execution of allocation and initialization sequences to the defining module, which also prevents clients from binding themselves to concrete classes by instantiating them directly.

These features can be combined to develop safe implementations of all creational patterns described in *Design Patterns*: abstract factories, builders, factory methods, prototypes, and singletons.

• Dependence on specific operations

Static record variables can be used as light-weight (stack-allocated) message objects, instead of using hard-coded method signatures or heavy-weight (heap-allocated) message objects. Message records can be forwarded, filtered, broadcast, and so on. Yet they don't violate type safety, since addresses of static records cannot be manipulated in unsafe ways.

Message records can be used to implement the chain of responsibility and observer patterns, which decouple a message sender and its receiver(s).

• Dependence on hardware and software platforms

• Dependence on object representations or implementations

Portability is one of the main advantages of using a true high-level language. Component Pascal abstracts from the underlying hardware, yet its semantic gap is small enough that very efficient machine code can be generated. As in Java, type sizes are defined so that data transfer between machines doesn't create problems.

• Algorithmic dependencies

Parts of an algorithm can be made replaceable by using abstract methods, empty (hook) methods, or auxiliary objects. Implement-only export makes it possible to safely assign responsibilities: correct method calling sequences must be implemented by the defining module, implementing the methods must be done by the extension programmer. This feature allows to develop safe implementations of all behavioral patterns (builder, internal iterator, strategy, template method, and so on).

• Tight coupling
Modules act as visibility boundaries. Thus tight coupling is possible where it is necessary (within a module's implementation), while loose coupling is possible across module boundaries. This simplifies safe implementation of virtually all design patterns.

• Extending functionality by subclassing
Classes can be hidden in modules. Final or limited classes, and final methods can be exported without risk that they may be subclassed or overridden.

• Inability to alter classes conveniently
Strong typing, record and method attributes such as *NEW*, and run-time assertions increase the confidence that an interface can be changed in a way that all clients can be made consistent again easily and reliably. This is basic software engineering, and thus important for all kinds of designs.

All these problems have one common theme: a local change may cause ripple effects that affect the entire software system. Good languages and design practices help to design for change:

Ensure that possible design changes have local effects only.
Where this is not possible, ensure that the effects are detected by the compiler.
Where this is not possible, ensure that the effects are detected at run-time as early as possible.

It is clear that there still remains a considerable gap between current state-or-the-art languages like Component Pascal, and complete specification languages that also allow to specify the semantics of a program. In the future, it will be the challenge to close those parts of the gap which have a good enough cost/benefit ratio, i.e., which help to change a component's behavior in a controlled manner, without making programs unreadable and unwriteable for average programmers.

3.2 Modules and subsystems

In this section, we give a more concrete idea of how Component Pascal code looks like and how it is managed by the BlackBox Component Builder environment. For this purpose, we have to go into more BlackBox-specific details than in the other sections of this part of the book.

All Component Pascal code lives in modules. A module is the compilation unit of Component Pascal. A module has an interface and a hidden implementation. Syntactically this is achieved by marking some items in a module, e.g., some types and procedures, as *exported*. Everything not exported is invisible and thus not directly accessible from outside of the module. If a module needs services of one or several other modules, it *imports* them. Thereby the module declares that it requires descriptions of the interfaces of the imported modules at compile-time, and suitable implementations of these modules at run-time. Listing 3-3 shows a possible implementation of a module called *ObxPhoneDB*. It exports three procedures for looking up phone book entries: by index, by name, and by number. Please note the asterisks which denote items as exported, in this case the type *String* and the three lookup procedures:

```
MODULE ObxPhoneDB;
```

```
   CONST
      maxLen = 32;   (* maximum length of name/number strings *)
      maxEntries = 5;   (* maximum number of entries in the database *)

   TYPE
      String* = ARRAY maxLen OF CHAR;

      Entry = RECORD
         name, number: String
      END;

   VAR db: ARRAY maxEntries OF Entry;

   PROCEDURE LookupByIndex* (index: INTEGER; OUT name, number: String);
   BEGIN   (* given an index, return the corresponding <name, number> pair *)
      ASSERT(index >= 0);
      IF index < maxEntries THEN
         name := db[index].name; number := db[index].number
      ELSE
         name := ""; number := ""
      END
   END LookupByIndex;

   PROCEDURE LookupByName* (name: String; OUT number: String);
      VAR i: INTEGER;
   BEGIN   (* given a name, find the corresponding phone number *)
      i := 0; WHILE (i # maxEntries) & (db[i].name # name) DO INC(i) END;
      IF i # maxEntries THEN   (* name found in db[i] *)
         number := db[i].number
      ELSE   (* name not found in db[0..maxEntries-1] *)
         number := ""
      END
   END LookupByName;

   PROCEDURE LookupByNumber* (number: String; OUT name: String);
      VAR i: INTEGER;
   BEGIN   (* given a phone number, find the corresponding name *)
      i := 0; WHILE (i # maxEntries) & (db[i].number # number) DO INC(i) END;
      IF i # maxEntries THEN   (* number found in db[i] *)
         name := db[i].name
      ELSE   (* number not found in db[0..maxEntries-1] *)
         name := ""
      END
   END LookupByNumber;

BEGIN   (* initialization of database contents *)
   db[0].name := "Daffy Duck"; db[0].number := "310-555-1212";
   db[1].name := "Wile E. Coyote"; db[1].number := "408-555-1212";
   db[2].name := "Scrooge McDuck"; db[2].number := "206-555-1212";
   db[3].name := "Huey Lewis"; db[3].number := "415-555-1212";
   db[4].name := "Thomas Dewey"; db[4].number := "617-555-1212"
END ObxPhoneDB.
```

Listing 3-3. Implementation of ObxPhoneDB

Before a module can be used, it must be loaded from disk into memory. But before it can be loaded, it must be compiled (command *Dev->Compile*). When compiling a module, the compiler produces a code file and a symbol file. The code file contains the executable code, which can be loaded into memory. The code file is a kind of super-lightweight DLL. The compiler also produces a symbol file, which contains a binary representation of the module's interface. If a module imports other modules, the compiler reads the symbol files of all these modules, in order to check that their interfaces are used correctly. The compilation process can be visualized in the following way:

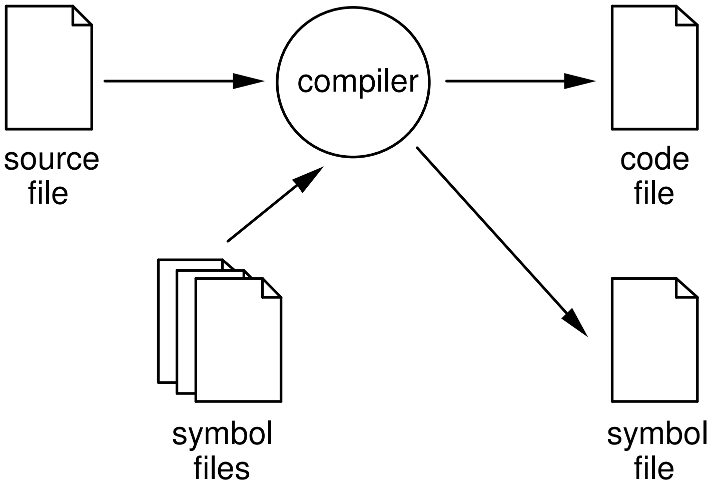


Figure 3-4. Compilation process

When you compile a module for the first time, a new symbol file is generated. In the log window, the compiler writes a message similar to the following one:

```
compiling "ObxPhoneDB"
  new symbol file   964   640
```

The first of the two numbers indicates that the machine code in the new code file is 964 bytes long. The second number indicates that the module contains 320 bytes global variables (five entries in the db variable; each entry consisting of two strings with 32 elements each; each element is a 2-byte Unicode character). If a symbol file for exactly the same interface already exists, the compiler writes a shorter message:

```
compiling "ObxPhoneDB"   964   640
```

If the interface has changed, the compiler writes a new symbol file and indicates the changes compared to the old version in the log. For example, if you just have introduced procedure *LookupByNumber* in the most recent version, the compiler writes:

```
compiling "ObxPhoneDB"
  LookupByNumber is new in symbol file   964   640
```

Symbol files are only used at compile-time, they have no meaning at run-time. In order to load a module, only its code file is needed. Modules are loaded dynamically, i.e., there is no separate linking step as required by more static languages. To see a list of currently loaded modules, call command *Info->Loaded Modules*. As result, a window will be opened with a contents similar to the following one:

module name	*bytes used*	*clients*	*compiled*		*loaded*	Update
StdLinks	20639	1	2.7.1996	18:42:15	29.8.1996	14:31:14
StdFolds	20425	1	2.7.1996	18:41:33	29.8.1996	14:31:12
StdCmds	25066	7	2.7.1996	18:39:12	29.8.1996	14:31:00
Config	125	0	2.7.1996	18:38:21	29.8.1996	14:31:20
Init	682	0	2.7.1996	18:40:21	29.8.1996	14:31:05
Controls	78876	5	7.7.1996	14:14:58	29.8.1996	14:31:00
Services	1472	5	2.7.1996	18:37:14	29.8.1996	14:30:54
Containers	37348	40	2.7.1996	18:37:51	29.8.1996	14:30:52
Properties	8337	42	2.7.1996	18:37:40	29.8.1996	14:30:49
Controllers	6037	42	2.7.1996	18:37:36	29.8.1996	14:30:49
Views	31589	49	2.7.1996	18:37:33	29.8.1996	14:30:49
Models	4267	50	2.7.1996	18:37:27	29.8.1996	14:30:48
Converters	2189	51	14.7.1996	22:45:12	29.8.1996	14:30:48
Dialog	8979	54	2.7.1996	18:37:13	29.8.1996	14:30:48
Dates	3848	45	2.7.1996	18:37:07	29.8.1996	14:30:48
Meta	19275	11	2.7.1996	18:37:10	29.8.1996	14:30:48
Stores	22302	53	2.7.1996	18:37:22	29.8.1996	14:30:47
Strings	17547	15	2.7.1996	18:37:05	29.8.1996	14:30:47
Math	15408	2	3.7.1996	1:45:05	29.8.1996	14:30:47
Ports	10631	56	2.7.1996	18:37:17	29.8.1996	14:30:46
Fonts	1589	58	2.7.1996	18:37:15	29.8.1996	14:30:46
Files	3814	62	2.7.1996	18:36:28	linked	
...						

Table 3-5. List of loaded modules

The list shows all loaded modules. For each module, it shows its code size in bytes, how many other modules import it, when it has been compiled, and when it has been loaded.

It is easy to get an overview over the already loaded modules, but what about modules not yet loaded? The idea of having access to a wealth of prebuilt components raises some organizational issues. How do you find out exactly which components are available? How do you find out which of the available components provide the services that you need?
This is a matter of conventions, documentation, and supporting tools. For the BlackBox Component Builder, it is a convention that collections of related components, called *subsystems*, are placed into separate directories; all of which are located directly in the BlackBox directory. There are subsystems like System, Std, Host, Mac, Win, Text, Form, Dev, or Obx. The whole collection of subsystems is called the BlackBox *repository*. The basic idea behind the repository's structure is that everything that belongs to a component (code files, symbol files, documentation, resources) are stored together in a systematic and simple directory structure, according to the rule

Keep all constituents of a component in one place.

It is only appropriate for component-oriented software that addition and removal of a component can be performed incrementally, by adding or removing a directory. All kinds of central installation or registration mechanisms which distribute the constituents of a component should be avoided, since they inevitably lead to (unnecessary) management problems.

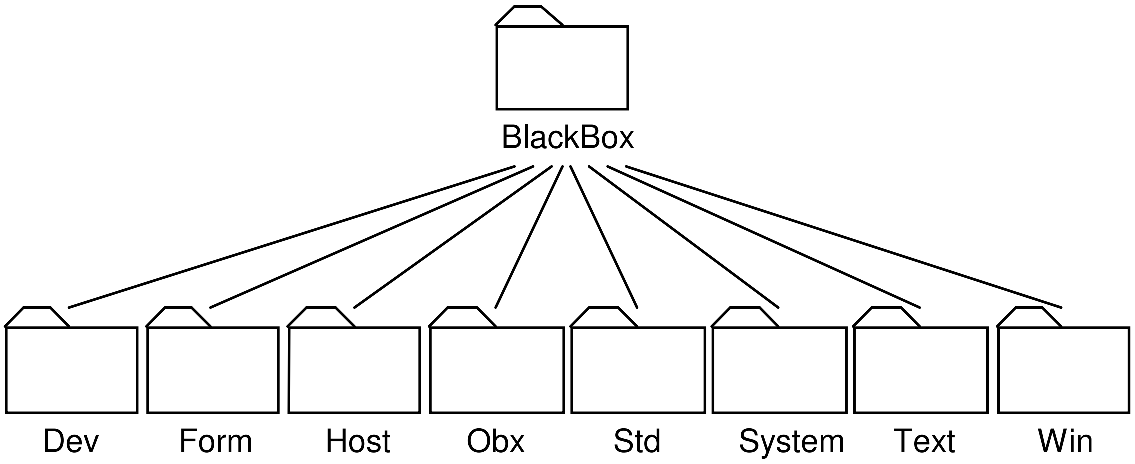


Figure 3-6. Standard subsystems of the BlackBox Component Builder

Each subsystem directory, e.g. Obx, may contain the following subdirectories:

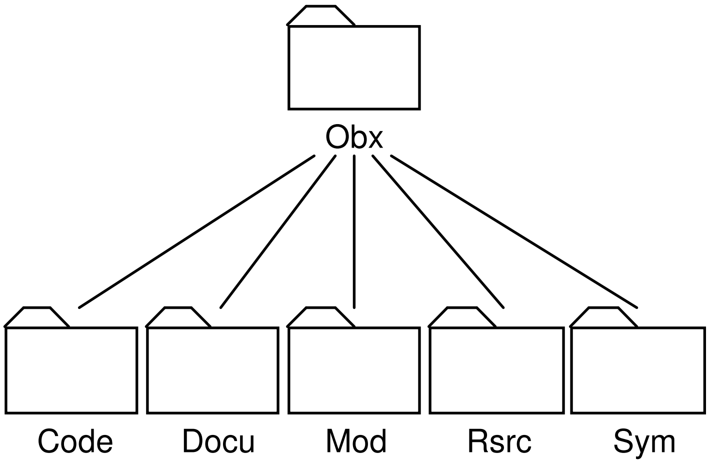


Figure 3-7. Structure of a typical subsystem directory

The module source is saved in a subsystem's Mod directory. The file name corresponds to the module name without its subsystem prefix; e.g., the modules *ObxPhoneDB* and *ObxPhoneUI* are stored as Obx/Mod/PhoneDB and Obx/Mod/PhoneUI, respectively. For each source file, there may be a corresponding symbol file, e.g., Obx/Sym/PhoneDB; a corresponding code file, e.g., Obx/Code/PhoneDB; and a corresponding documentation file, e.g., Obx/Docu/PhoneDB. There may be zero or more resource documents in a subsystem, e.g., Obx/Rsrc/PhoneUI. There is not necessarily a 1:1 relationship between modules and resources, although it is generally recommended to use a module name as part of a resource name, in order to simplify maintenance.

Modules whose names have the form SubMod, e.g., *TextModels*, *FormViews*, or *StdCmds*, are stored in their respective subsystems given by their name prefixes, e.g., *Text*, *Form*, or *Std*. The subsystem prefix starts with an uppercase letter and may be followed by several other uppercase letters and then by several lowercase letters or digits. The first uppercase letter afterwards denotes the particular module in the subsystem.

Modules which belong to no subsystem, i.e., modules whose names are not in the form of SubMod, are stored in a special subsystem called *System*. The whole BlackBox library and framework core belongs to this category, e.g., the modules *Math*, *Files*, *Stores*, *Models*, etc.

Each subsystem directory may contain the following subdirectories:

Code	Directory with the executable code files, i.e., lightweight DLLs.

	For example, for module "FormCmds" there is file "Form/Code/Cmds". A module for interfacing native DLLs (Windows DLLs or Mac OS code fragments) has no code file.
Docu	Directory with the fully documented interfaces and other docu. For example, for module "FormCmds" there is file "Form/Docu/Cmds". For a module that is only used internally, its docu file is not distributed to the customer. Often, there are further documentation files which are not specific to a particular module of the subsystem. Such files contain one or more dashes as parts of their names, e.g., "Dev/Docu/P-S-I". Typical files are "Sys-Map" (overview with hyperlinks to other documents of this subsystem) "User-Man" (user manual) "Dev-Man" (developer manual)
Mod	Directory with module sources. For example, for module "FormCmds" there is file "Form/Mod/Cmds". For a module that is not published in source code ("white box"), its source file is not distributed to the customer.
Rsrc	Directory with the subsystem's resource documents. For example, for module "FormCmds" there is file "Form/Rsrc/Cmds". There may be zero, one, or more resource files for one module. If there are several files, the second gets a suffix "1", the third a suffix "2", and so on. For example, "Form/Rsrc/Cmds", "Form/Rsrc/Cmds1", "Form/Rsrc/Cmds2", etc. Often, there are further resource files which are not specific to a particular module of the subsystem. Typical files are "Strings" (string resources of this subsystem) "Menus" (menus of this subsystem)
Sym	Directory with the symbol files. For example, for module "FormCmds" there is file "Form/Sym/Cmds". For a module that is only used internally, its symbol file is not distributed to the customer.

Table 3-8. Contents of the standard subsystem subdirectories

If you want to find out about the repository, its subsystems and their subdirectories, you can invoke the command *Info->Repository*. If you want to find out more about a module, you can select the module name in a text and then execute *Info->Source*, *Info->Interface* or *Info->Documentation*. These commands open the module's Mod, Sym or Docu files. For this purpose, *Info->Interface* converts the binary representation of the symbol file into a readable textual description. For example, type the string "ObxPhoneDB" into the log window, select the string, and then execute the *Info->Interface* browser command. As a result, the following text will be opened in a new window:

```
DEFINITION ObxPhoneDB;

   TYPE
      String = ARRAY 32 OF CHAR;

   PROCEDURE LookupByIndex (index: INTEGER; OUT name, number: String);
   PROCEDURE LookupByName (name: String; OUT number: String);
   PROCEDURE LookupByNumber (number: String; OUT name: String);

END ObxPhoneDB.
```

Listing 3-9. Definition of ObxPhoneDB

A module definition as generated by the browser syntactically is a subset of the module implementation, except for the keyword *MODULE* which is replaced by *DEFINITION*. This syntax can be regarded as the interface description language (IDL) of Component Pascal. Texts in this language are usually created out of complete module sources by the browser or similar tools, and thus need not be written manually and cannot be compiled.

Since the browser command operates on the symbol file of a module, it can be used even if there is neither a true documentation nor a code file available. During prototyping, where documentation is rarely available, the symbol file browser is very convenient for quickly looking up details like the signature of a procedure, or to get an overview over the interface of an entire module. When a full documentation is available, the command *Info->Documentation* can be used. It opens the module's documentation file, which starts with the definition of the module's interface just like above, but then continues with an explanation of the module's purpose and a detailed description of the various items exported by the module, for example:

Module ObxPhoneDB provides access to a phone database. Access may happen by index, by name, or by number. An entry consists of a name and a phone number string. Neither may be empty. The smallest index is 0, and all entries are contiguous.

PROCEDURE **LookupByIndex** (index: INTEGER; OUT name, number: ARRAY OF CHAR)
Return the <name, number> pair of entry *index*. If the index is too large, <"", ""> is returned.
The procedure operates in constant time.

Pre
index >= 0   20

Post
index is legal
  name # "" & number # ""
index is not legal
  name = "" & number = ""

PROCEDURE **LookupByName** (name: ARRAY OF CHAR; OUT number: ARRAY OF CHAR)
Returns a phone number associated with *name*, or "" if no entry for *name* is found.
The procedure operates in linear time, depending on the size of the database.

Post
name found
  number # ""
name not found
  number = ""

PROCEDURE **LookupByNumber** (number: ARRAY OF CHAR; OUT name: ARRAY OF CHAR)
Returns the name associated with *number*, or "" if no entry for *number* is found.
The procedure operates in linear time, depending on the size of the database.

Post
number found
  name # ""
number not found
  name = ""

Listing 3-10. Documentation of ObxPhoneDB

Note that preconditions and postconditions are documented in a semi-formal notation. Their goal is not to give a complete formal specification, but rather to help making the plain text description less ambiguous, where this is possible without using overly complex (and thus unreadable) formal conditions. The following assertion numbers are used for run-time checking in BlackBox:

Free	0 .. 19
Preconditions	20 .. 59
Postconditions	60 .. 99
Invariants	100 .. 120
Reserved	121 .. 125
Not Yet Implemented	126
Reserved	127

Listing 3-11. Assertion numbers in BlackBox

It is well known that the detection of an error is more difficult and more expensive the later it occurs, i.e., the farther apart the cause and its effects are. This motivates the following design rule:

Let errors become manifest as early as possible.

In a component-oriented system, defects should always be contained within their components, and not be allowed to propagate into other components. The other components may even be black-boxes for which no source code is available, which makes source-level debugging impossible. Furthermore, the control flow of a large object-oriented software system is so convoluted that it is unrealistic, and thus a waste of time, to trace it beyond component boundaries for debugging purposes.

The only viable debugging approach is to design everything, from programming language to libraries to components and applications using a defensive programming style. In particular, entry points into components (procedure/method calls) should refuse to execute if their preconditions are not met:

Never let errors propagate beyond component boundaries.

Fortunately, most precondition checks are inexpensive and thus their run-time overhead is negligible. This is important because in a component-oriented system, run-time checks cannot be switched off in a production system, because there *are* no separate development and production systems. In practice, most components during development are already debugged ("production") black-boxes, and the others are currently being debugged white-boxes. The production components must cooperate in order to make adherence to the above rule possible, which means never switching off run-time checks.

3.3 Bottleneck interfaces

One particular pattern that we would like to discuss here is the Carrier-Rider-Mapper pattern. This pattern is used in several ways in the BlackBox Component Framework: in the text subsystem, in the file abstraction, in the frame abstraction, in the container/context abstraction, and others. Let us consider texts as an example.

We regard a text as a so-called *carrier*. A carrier is an object that carries (contains) data, in this case textual data. Basically, a text can be regarded as a linear stream of elements, where elements have attributes such as font, color, style and vertical offset (for subscript and superscript characters). Elements are characters or views. For all practical purposes, a view in a text can simply be regarded as a special character.

When reading a text, it is convenient not to require the specification of a text position for each and every

character read. This convenience can be achieved by supporting a *current position*, i.e., the text itself knows where to read the next character. Each read operation automatically increments the current position.
Several client objects may use a text carrier independently. For example, a text view needs to read its text when it redraws the text on screen, and a menu command may need to read the same text as its input. These clients are independent of each other; they don't need to know about each other. For this reason, the current read position cannot be stored in the text carrier itself, since this would mean that clients could interfere with each other and lose their independence.
To avoid such interference, a carrier provides allocation functions that return so-called *rider* objects. A rider is an independent access path to a carrier's data. Every text rider has its own current position, and possibly further state such as the current attributes.

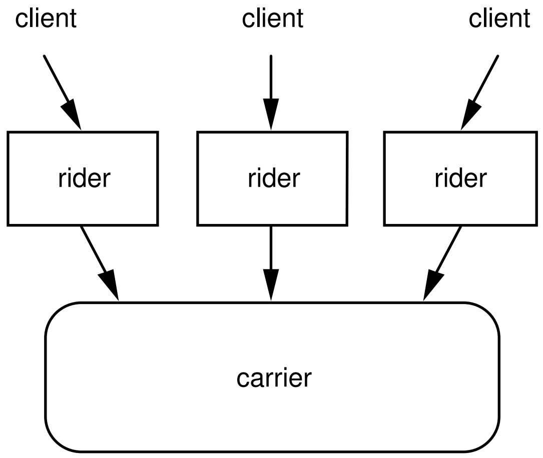


Figure 3-12. Carrier-Rider separation

The reason why carrier and rider are separated into different data types is the one-to-many relationship between the two: for one carrier there may be an arbitrary number (including zero) riders.

Typically, carrier and rider need to be implemented by the same component, since a rider must have intimate knowledge about the internals of its carrier. For example, text riders and text carriers are both implemented in module *TextModels*. A text rider contains hidden pointers to the internal data structure of a text carrier. Since this information is completely hidden by the module boundary of *TextModels*, no outside client can make a rider inconsistent with its carrier. This kind of invariant, guaranteed by Component Pascal's module system, is an example why information hiding beyond single classes is important for safety reasons.

In the design patterns terminology, a text rider is an *external iterator*. However, not all riders are necessarily iterators. For example, a context object managed by a container is not an iterator.

Text riders come in two flavors: readers and writers. Readers support the reading of characters and views, while writers support the writing of characters and views. For example, the following calls may be made:

```
reader.ReadView(view)
```

or

```
writer.WriteChar("X")
```

Often, a programmer needs to read or write complex character sequences, so that working at the level of individual characters is too cumbersome. A higher-level abstraction for reading and writing is clearly desirable. This is the purpose of the *mappers*. A mapper contains a rider that it uses to provide more problem-oriented

operations for reading or for writing. Since there exist very different programming problems, different applications may use different mappers, possibly even while working with the same carrier.
For text manipulation, BlackBox provides module *TextMappers*, which defines and implements two text mappers: *formatters* for writing, and *scanners* for reading.

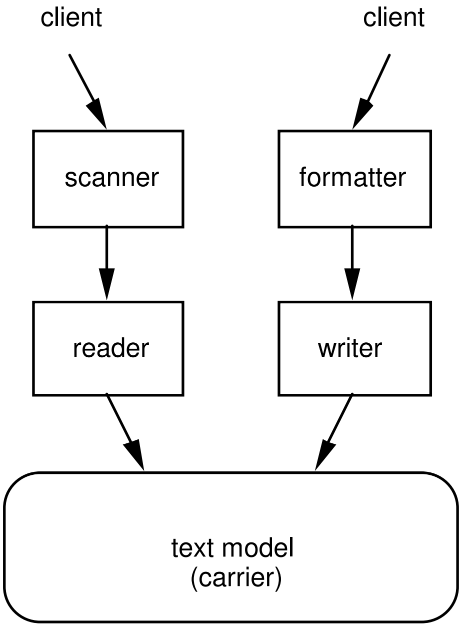


Figure 3-13. Carrier-Rider-Mapper separation

Both scanner and formatter work on the level of Component Pascal symbols: with integer numbers, real numbers, strings, and so on. For example, the following calls may be made:

```
scanner.ReadInt(int)
```

or

```
formatter.WriteReal(3.14)
```

If there are different mappers for the same kind of carrier, they all have to be implemented in terms of the rider interface. For this reason, the rider/carrier interface is sometimes called a *bottleneck interface*. The reason why riders and mappers are separated is not a one-to-many relation as with carrier and rider - there is a one-to-one relation between mapper and rider - but independent extensibility: special circumstances may require special mappers. If carriers and mappers adhere to a well-defined rider (bottleneck) interface, it is possible to add new mappers anytime, and to use them with the same carrier.
Even better: the carrier/rider interface can be an abstract interface, such that different *implementations* of it may exist. For example, a file carrier may be implemented differently for floppy disk files, hard disk files, CD-ROM files, and network files. But if they all implement the same bottleneck interface, then *all* file mappers can operate on *all* file implementations! Compare this with a situation where you would have to re-implement all mappers for all carriers: instead of implementing n carriers plus m mappers, you would have to implement n carriers plus n * m mappers! Wherever such an extensibility in two dimensions occurs (carrier/rider and mapper), a bottleneck interface is needed to avoid the so-called *cartesian product problem*, i.e., an explosion of implementations.

Note that a bottleneck interface is not extensible itself, because every extension that cannot be implemented in terms of the bottleneck interface invalidates all its existing implementations. For example, if you extend the interface of a device driver for black-and-white bitmap displays by a color extension, then all existing device driver implementations will have to be updated. Since they probably have been implemented by different companies, this can become a major problem.

The problem can be defused if the new interface extension is specified as optional: then clients must test whether the option is supported. If not, a client either has to signal that it cannot operate in this environment, or it must gracefully degrade by providing a more limited functionality. If the optional interface is supported and used, it acts as a kind of "conspiracy" between the interface implementation and its client. This is admissible, but it clearly reduces the combinations of clients and implementations that are fully functional. This underlines how crucial good bottleneck designs are, in order to avoid the need for later extensions.

mapper rider	Binary Mapper	ASCII Mapper
RS232Rider	Binary on RS232	ASCII on RS232
NetRider	Binary on net	ASCII on net

Figure 3-14. Extensibility in two dimensions

In terms of design patterns, the Carrier-Rider-Mapper pattern solves the problem of flexible and convenient access to a data carrier, however it is implemented. It can be regarded to consist of two simpler and very basic design patterns: the separation of one object into several objects to allow many-to-one relations; and the separation of one object into two objects to allow for independent extensibility:

Split an abstraction into two interfaces if several clients may access an instance simultaneously, and it independent state may have to be managed for every client.

Split an abstraction into two interfaces, if it needs to be extended independently in two different dimensions.

The Carrier-Rider-Mapper separation goes back to a research project ["Insight ETHOS: On Object-Orientation in Operating Systems"; Clemens Szyperski; vdf, Zürich, 1992, ISBN 3 7281 1948 2] predating BlackBox. This project used several design patterns and design rules (e.g., avoidance of implementation inheritance) that are also described in *Design Patterns*. In the *Design Patterns* terminology, a Rider-Mapper combination (or Carrier-Mapper combination if there is no rider) forms a *bridge* pattern.

In BlackBox, riders are often created and managed in a particular way. A rider is typically under exclusive control of one client object, because after all, the reason why there can be multiple riders on one carrier (and thus why the two are distinguished) is precisely to allow several clients access to the carrier's data via their private access paths. Since riders are used in such controlled environments, BlackBox usually creates riders in the following way:

```
rider := carrier.NewRider(rider);
```

The idea is that if there already exists an old rider that isn't used anymore, it can be recycled. Recycling is done by the *NewRider* factory method (see next section) if possible. Of course, recycling is only legal if the old rider isn't used anymore for something else (possibly by someone else); that's why it is important that the rider is maintained in a controlled environment.

Typically, a *NewRider* procedure is implemented in the following way:

```
PROCEDURE (o: Obj) NewRider (old: Rider): Rider;
    VAR r: RiderImplementation;
```

```
BEGIN
    IF (old # NIL) & (old IS RiderImplementation) THEN    (* recycle old rider *)
        r := old(RiderImplementation)
    ELSE    (* allocate new rider *)
        NEW(r)
    END;
    ... initialize r ...
    RETURN r
END NewRider;
```

Listing 3-15. Rider recycling

This approach is used in BlackBox for files, texts, and forms. It can make a considerable difference in efficiency, depending on the kind of application. However, where efficiency is not a concern, it is probably a good idea to omit this mechanism, thereby avoiding any possibility for inadvertant reuse of riders in different contexts simultaneously.

3.4 Object creation

The BlackBox Component Framework is mostly a black-box design. It strictly separates interface from implementation. A client can only import, and thus directly use or manipulate, interfaces of objects. The implementations are hidden within the modules. This enforces that module clients adhere to the following design rule (Gamma et.al.):

Program to an interface, not to an implementation.

In Component Pascal, object interfaces are represented as abstract record types. An implementation of an abstract record type is a concrete extension of it. Typically, implementations are not exported and thus cannot be extended ("subclassed") in other modules, meaning that implementation inheritance cannot be used. Thus it is also enforced that extension programmers adhere to the following design rule (Gamma et.al.):

Favor object composition over class inheritance.

Concrete record types cannot even be instantianted directly: because a client cannot import an object implementation whose type is not exported, it cannot allocate an instance by calling *NEW*. This is desirable, because *NEW* would couple the client code forever to one particular object implementation; reuse of the code for other implementations wouldn't be possible anymore. But how can client code solve the *black-box allocation problem*; i.e., obtain an object implementation without specific knowledge of it?

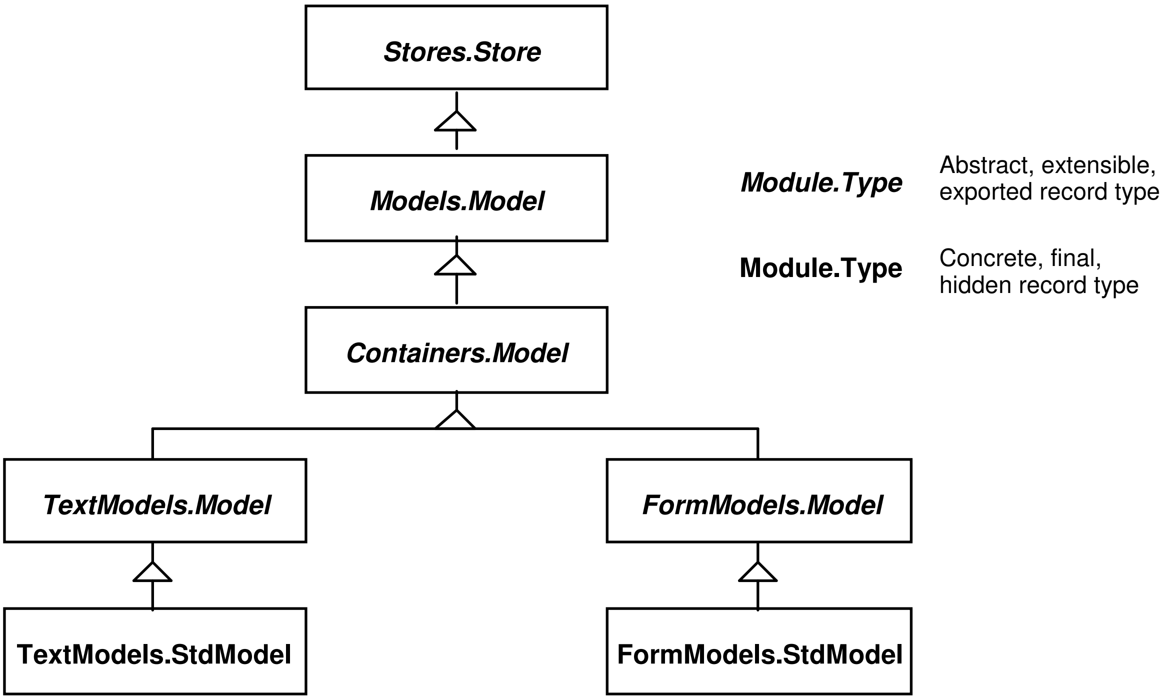


Figure 3-16. Exported interface records and hidden implementation records

Instead of calling *NEW* it might be possible to call an allocation function instead, a so-called *factory function*. For example, module *TextModels* might export a function *New* that allocates and returns a text model. Internally, it would execute a *NEW* on the non-exported implementation type and possibly perform some initialization. This would be better than clients directly calling *NEW*, because the allocating module can guarantee correct initialization, and a new release of the module could perform allocation or initialization in some different way without affecting clients. However, static factory functions are still too inflexible. A proper solution of the black-box allocation problem requires a level of indirection. There are several approaches to achieve this. They are called creational patterns. BlackBox uses four major creational patterns: prototypes, factory methods, directory objects, and factory managers. They are discussed one by one.

Sometimes it is necessary to create an object of the same type as some other already existing object. In this case, the existing object is called a *prototype*, which provides a factory function or initialization method. BlackBox models can act as prototypes. For example, it is possible to clone a prototype and to let the prototype initialize its clone. This makes sure that the newly created object has exactly the same concrete type as the prototype, and it allows the prototype and its clone(s) to share caches for performance reasons. For example, a BlackBox text model and its clones share the same spill file that they use for buffering temporary data. Sharing avoids the proliferation of open files.

Sometimes there exists an object that can be used to create an object of *another* type. For this purpose, the existing object provides a factory function, a *factory method*. Possibly, the object may provide different factory methods which support different initialization strategies or which create different types of concrete objects.
In BlackBox, a text model provides the procedures *NewReader* and *NewWriter*, which generate new readers and writers. They are rider objects that represent access paths to a text.
Factory methods are appropriate for objects that are implemented simultaneously ("*implementation covariance*"). This is always the case for riders and their carriers.

But this approach is not sufficient for allocating the text models themselves. Where does the first text model come from? BlackBox uses special objects with factory methods, so-called *factory objects*. In BlackBox, factory objects are used in a particular way: they are installed in global variables and may be replaced at run-time, without affecting client code. For historical reasons, we call factory objects which are used for configuration purposes *directory objects*.

For example, module *TextModels* exports the type *Directory* which contains a *New* function, furthermore it exports the two variables *dir* and *stdDir* and the procedure *SetDir*. By default, *TextModels.dir* is used by client code to allocate new empty text models. *TextModels.stdDir* contains the default implementation of a text model, which is mostly useful during the debugging of a new text model implementation. For example, you could use the old directory while developing a new text model implementation. When the new one is ready you install its directory object by calling *SetDir*, thereby upgrading the text subsystem to the new implementation on the fly. From this time, newly created texts will have the new implementation. Texts with the old implementation won't be affected. If the new implementation has errors, the default implementation can be reinstalled by calling *TextModels.SetDir(TextModels.stdDir)*. The text subsystem uses directory objects in this typical way; the same design pattern can be found in other container model abstractions; e.g., for forms models. In BlackBox, it is even used for much simpler models and views; e.g., for *DevMarkers.*

Note that objects allocated via directories are often persistent and their implementation thus long-lived. For this reason, several different implementations appear over time which have to be used simultaneously.

Typically, the normal default directory objects of a subsystem are installed automatically when the subsystem is loaded. If other directory objects should be installed, a *Config* module is used to set up the desired directory object configuration upon startup.

The name "directory object" comes from another use of the directory design pattern: in module *Files*, a file can be created by using the directory object *Files.dir*. But a *Files.Directory* also provides means to find out more about the current configuration of the file system (e.g., procedure *Files.Directory.FileList*).

File directories are interesting also because they show that a replacement directory may sometimes need to forward to the replaced directory object. For example, if the new directory implements special memory files, which are distinguished by names that begin with "M:\", then the new directory must check at each file lookup whether the name begins with the above pattern. If so, it is a memory file. If not, it is a normal file and must be handled by the old directory object.

There may exist several directory objects simultaneously. For example, if a service uses files in a particular way, it may provide its own file directory object which by default forwards to the file system's standard directory object.

Multiple directory objects and chaining of directory objects are part of the directory design pattern. This chaining part of the pattern is cascadable, if a new directory object accesses the most recent old one, instead of *stdDir*.

Note that unlike typical factory classes, directories are rarely extended.

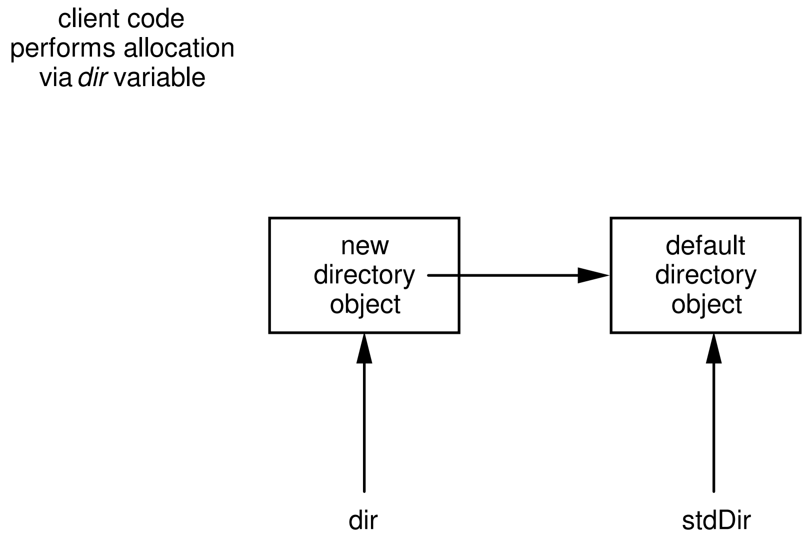


Figure 3-17. Forwarding between two directory objects

Directory objects are a particularly appropriate solution when the existence of one dominating implementation

of an extensible black-box type can be expected, without excluding the existence of other implementations.
On the other hand, there could also be several special directory objects for the same implementation but for different purposes. For example, there may be a special text directory object for texts that have a special default ruler useful for program texts. A program editor would then create new program text via this directory object rather than via the standard directory of the text subsystem.

A *registry* would be a valuable addition to directory objects. A registry is a persistent central database which stores configuration information, such as the particular directory objects to install upon startup. The problem with registries is that they contradict the decentral nature of software components, which leads to management problems such as finding registry entries that are no longer valid.

We have seen that allocation via one level of indirection is key to solving the black-box allocation problem. Directory objects are a solution involving global state. This is only desirable if this state changes rarely, and if no parallel activities occur (threads). For example, in server environments a state-free solution is often preferable. This can be achieved by parameterizing the client: the client must be passed a factory object as parameter. This makes it possible to determine the exact nature of an implementation at the "top" of a hierarchy of procedure calls, always passing the factory object "downward". This is useful since the top-level is typically much less reusable than lower-level code.
As an example, a file transfer dialog box may allow to specify the file transfer protocol along with information such as remote address, communication speed, and so on. The file transfer protocol, which indicates the implementation of a communication object, can be passed from the dialog box down to the communication software, where it is used to allocate a suitable communication object (e.g., a TCP stream object).
Instead of passing a factory object, its symbolic name may be passed. At the "bottom", a service interprets this name; if necessary loads the module which implements the corresponding factory function; and then creates an object using the factory function. In this way, the service acts as a *factory manager*.
This approach is used for some BlackBox services, in particular for the *Sql* and *Comm* subsystems. In both cases, the name of a module can be passed when allocating a new object. This module is expected to provide suitable factory functions. For *Sql*, the modules *SqlOdbc* and *SqlOdbc3* provide appropriate object implementations and factory functions. For *Comm*, the module *CommTCP* provides a suitable object implementation.
Factory managers are often more suitable for creating non-persistent objects, while directory objects are often more suitable for creating persistent objects.

Part II: Library

Part I of the BlackBox tutorial gives an introduction to the design patterns that are used throughout BlackBox. To know these patterns makes it easier to understand and remember the more detailed design decisions in the various BlackBox modules.

Part II of the BlackBox tutorial demonstrates how the most important library components can be used: control, form, and text components.

Part III of the BlackBox tutorial demonstrates how new views can be developed, by giving a series of examples that gradually become more sophisticated.

4 Forms

In the first chapter of this part, we concentrate on the forms-based composition of controls. "Forms-based" means that there exists a graphical layout editor, sometimes called a visual designer or screen painter, that allows to insert, move, resize, and delete controls. Controls can be made active by linking them to code pieces. For example, it can be defined what action should happen when a command button is clicked.

Controls can have different visible states, e.g., they can be enabled or disabled. This is a way to inform a user about which actions currently make sense. Setting up control states in a sensible way can make a large difference in the user-friendliness of an application. Unfortunately, these user interface features often require more programming effort than the actual application logic itself does. However, there are only a few concepts necessary to understand in order to build such user interfaces. These concepts will be explained by means of simple examples.

4.1 Preliminaries

We want to start as quickly as possible with a concrete example, but a few preliminary remarks are still in order. The reader is expected to have some basic knowledge of programming, preferably in some dialect of Pascal. Furthermore, he of she is expected to know how to use the platform (Windows or Macintosh), and the general user interface guidelines for the platform.
The BlackBox Component Builder is used as the tool to expose the various characteristics of component software in an exemplary way. The contents of the book applies both to the Windows and the Macintosh version of the BlackBox Component Builder; except for screendumps, which mostly use Windows. In order to minimize platform-specific remarks in the text, a few notational conventions are followed:

- Mac OS folders are called directories
- Path names contain "/" as directory separators, as in Unix and the World-Wide Web
- File and directory names may contain both capital and small letters
- Document file names are given without the ".odc" suffix used in Windows. Thus the file name Text/Rsrc/Find under Windows corresponds to Text\Rsrc\Find.odc on Windows, and to Text:Rsrc:Find under Mac OS.
- Modifier key: on Windows this is the Ctrl key, on Mac OS it is the Option key
- Menu commands: *M->I* is a shorthand notation for menu item I in menu M, e.g., *File->New*

When working in the BlackBox environment for the first time, the following may be helpful to remember: almost all modifications to BlackBox documents are undoable, making it quite safe to try out a feature. In general, multi-level undo is available; i.e., not only one, but several commands can be undone; as many as memory permits.

This book is not a replacement for the BlackBox user manual. It minimizes the use of specific BlackBox Component Builder tool features, and therefore the need for tool-specific descriptions. The text is intended to concentrate on programming, rather than on the tool. Where unavoidable, tool-specific explanations are given where they are first needed.

Most examples are also available on-line in the Obx/Mod directory of BlackBox. They can be opened and compiled immediately. At the end of every section, references to further Obx on-line examples for the same topic are given.

For readers who are not yet fluent in Component Pascal, Appendix B describes the differences beween Pascal and Component Pascal.

The Help screen of the BlackBox Component Builder gives direct or indirect access to the complete and extensive on-line documentation, e.g., to the user manual, to all the Obx ("**O**verview **b**y e**x**ample") examples, and so on. On the Web, additional resources may be found at http://www.oberon.ch.

4.2 Phone book example

Throughout this part of the book, we will meet variations of and additions to a specific example. The example is an exceedingly simple address database. The idea is not to present a full-fledged application with all possible bells and whistles, but rather a minimal example which doesn't hide the concepts to be explained behind large amounts of code. Nevertheless, the idea of how bells and whistles can be added, component by component, should become obvious over time.

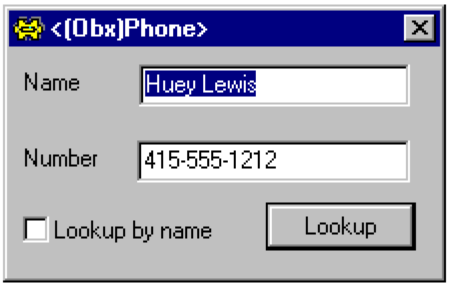
Our phone book database contains the following fixed entries:

Daffy Duck	310-555-1212
Wile E. Coyote	408-555-1212
Scrooge McDuck	206-555-1212
Huey Lewis	415-555-1212
Thomas Dewey	617-555-1212

Table 4-1. Entries in phone book database

The database can be searched for a phone number, given a name. Alternatively, the database can be searched for a name, given a phone number; or its contents can be accessed by index. We have met a possible implementation of this database in section 3.2.

Our goal is to create a user interface for the database. The user interface is a dialog box that contains two edit fields; one for the name, and the other for the phone number. For each text field, there is a caption that indicates its purpose. Furthermore, the dialog box contains a check box which allows to specify whether the name or the phone number should be looked up, and finally there is a command button which invokes the lookup command. The following screendump shows how the dialog box will look like eventually:



<(Obx)Phone>

Name: Huey Lewis

Number: 415-555-1212

☐ Lookup by name

Figure 4-2. Phone book database mask

To construct this mask, we start by creating a new empty form for the dialog box, using command *Controls->New Form*. This results in the following dialog box:

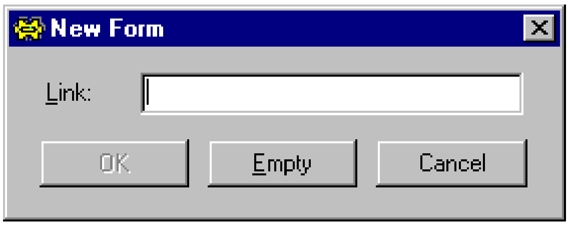


Figure 4-3. New Form dialog box

Clicking on the *Empty* command button opens an empty form:

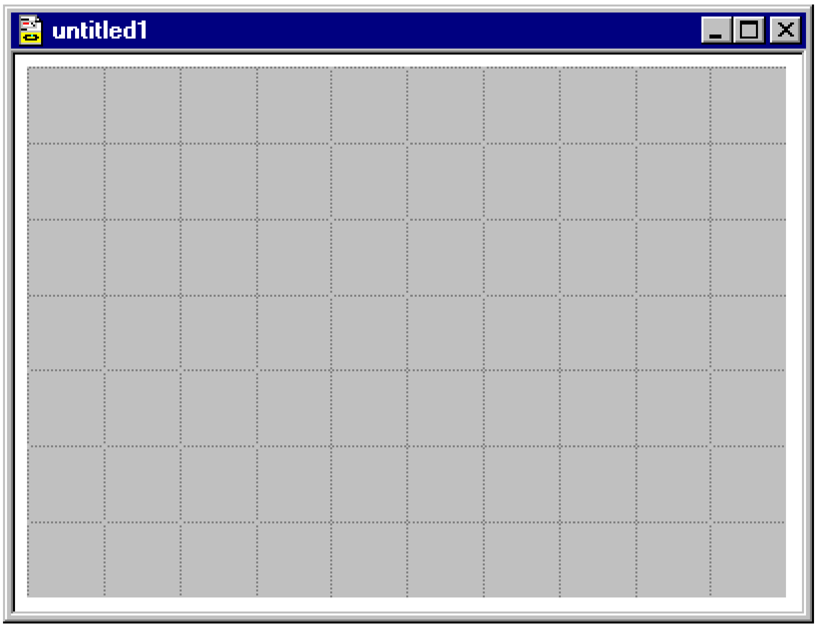
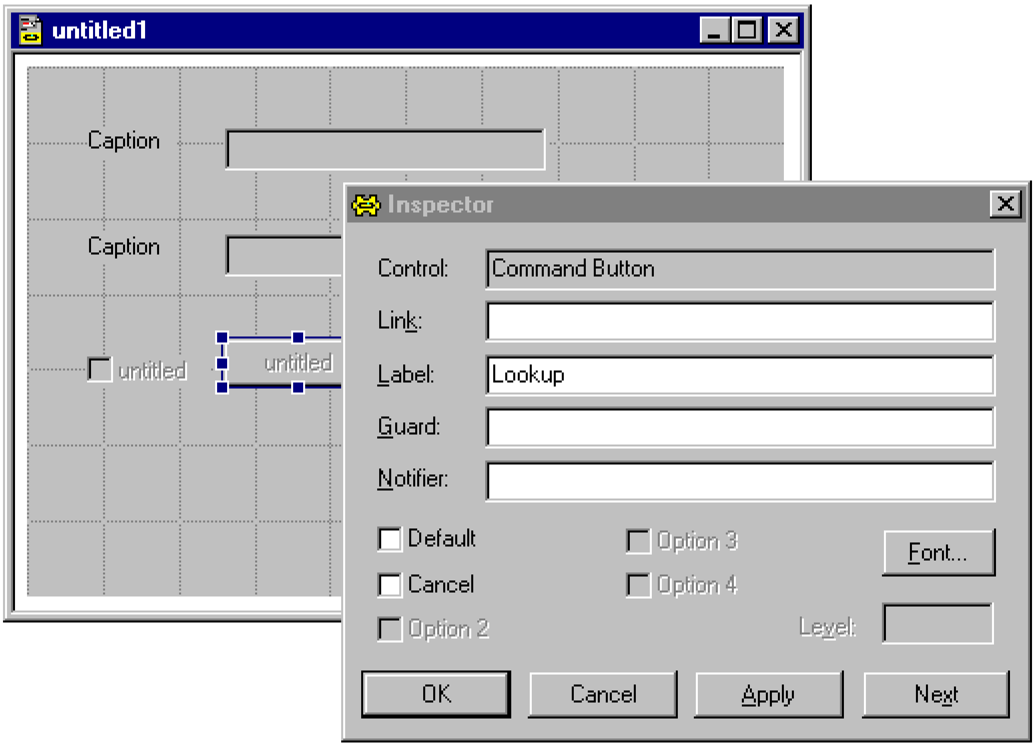


Figure 4-4. Empty form

Using the commands of menu *Controls* , we insert the various controls we need, i.e., two captions, two edit fields, a check box, and a command button. The controls we've inserted still have generic labels such as "untitled" or "Caption". To change these visual properties, a control property inspector is used. It is opened by selecting a control and then issuing *Edit->Object Properties...* (Windows) or *Edit->Part Info* (Mac OS), respectively. Edit the "label" field in order to change the selected control's label, and click on the default button to make the change permanent.



Control: Command Button

Link:

Label: Lookup

Guard:

Notifier:

□ Default
□ Cancel
□ Option 2
□ Option 3
□ Option 4

Level:

Figure 4-5. Control property editor

Change the "label" field of each control so that you end up with a layout similar to the one of Figure 4-2. Listed in a tabular way, the labels are the following ones (from left to right, top to bottom):

Control type	*Label*
Caption	Name
Text Field	
Caption	Number
Text Field	
Check Box	Lookup by Name
Command Button	Lookup

Table 4-6. List of controls in phone book dialog box

The controls can be rearranged by using the mouse or by using the layout commands in menu *Layout*. After having edited the layout, make sure to call *Layout->Sort Views*; this command sorts the controls in such a way that when being pressed, the tabulator key moves the cursor between them in the order you would expect, i.e., from left to right and from top to bottom.

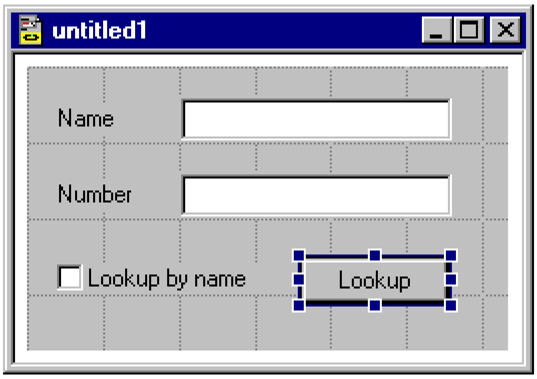


Figure 4-7. Completed layout of phone book dialog box

When you are happy with the layout, you can save the dialog box just like any other document, by using *File->Save*. As a convention, the dialog box layouts of all examples are saved in directory Obx/Rsrc. In this case, we save the new dialog box as Obx/Rsrc/PhoneUI.

The directory name *Rsrc* stands for "Resources". Resources are documents that are necessary for a program to work. In particular, they are dialog box layouts and string resources. String resources allow to move strings out of a program's source code into a separately editable document.
Resources can be edited without recompiling anything. For example, you could change all labels in the above dialog box from English to German, without having access to any source code or development tool.

4.3 Interactors

Creating a dialog box layout is fine, but there also must be a way to add behavior to the dialog box. In our example, user interactions with the dialog box should lead to lookup operations in the phone book database. To achieve this, we need an actual implementation of the database. We mentioned earlier that a suitable implementation already exists. This is not surprising, since in this part of the book we talk about component object assembly, which means taking existing components and suitably integrating persistent objects that they implement.
In our example, the phone book database is a Component Pascal module called *ObxPhoneDB.* We have already met this module in Chapter 3. Now we want to write our first new module, whose purpose is to build a bridge between the database module *ObxPhoneDB* and the dialog box we've built earlier. The new module is called *ObxPhoneUI*, where "UI" stands for "user interface". It is a typical script module whose only purpose is to add behavior to a compound document, such that it can be used as a front-end for the application logic, which in our case is simply the phone book database.
The new module uses, i.e., imports, two existing modules. On the one hand, it is the *ObxPhoneDB* module. On the other hand, module *Dialog*:

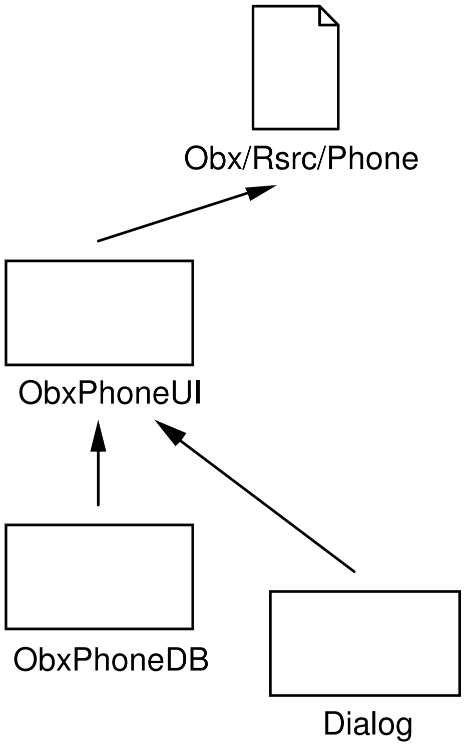


Figure 4-8. Import relation between ObxPhoneUI and ObxPhoneDB

Dialog is part of a fundamental framework coming with the BlackBox Component Builder. The module provides various services that support user interaction. We'll meet the most important ones in this chapter.

```
MODULE ObxPhoneUI;

   IMPORT Dialog, ObxPhoneDB;

   VAR
      phone*: RECORD
         name*, number*: ObxPhoneDB.String;
         lookupByName*: BOOLEAN
      END;

   PROCEDURE Lookup*;
   BEGIN
      IF phone.lookupByName THEN
         ObxPhoneDB.LookupByName(phone.name, phone.number);
         IF phone.number = "" THEN phone.number := "not found" END
      ELSE
         ObxPhoneDB.LookupByNumber(phone.number, phone.name);
         IF phone.name = "" THEN phone.name := "not found" END
      END;
      Dialog.Update(phone)
   END Lookup;

END ObxPhoneUI.
```

Listing 4-9. First version of ObxPhoneUI

ObxPhoneUI exports a global record variable *phone*, which contains two string fields and a Boolean field. Depending on the current value of the Boolean field, the *Lookup* procedure either takes *phone.name* to look up the corresponding number, or *phone.number* to look up the corresponding name. If the lookup fails, i.e., returns the empty string, the result is turned into "not found". Either result is put into *phone*, i.e., *phone* carries both input and output parameters for the database lookup.

Since the user could change the contents of *phone* by interactively manipulating a control, e.g., by typing into a text entry field, a global variable like *phone* is called an *interactor*. Controls display the contents of their interactor fields and possibly let them be modified interactively. To do this, every control must first be linked to its corresponding interactor field. This is done with the control property editor that we have seen earlier. Its "Link" field should contain the field name, e.g., *ObxPhoneDB.phone.name* or a procedure name such as *ObxPhoneDB.Lookup*. Use the control property inspector to set up the link fields according to Table 4-10:

Control type	*Label*	*Link*
Caption	Name	
Text Field		ObxPhoneUI.phone.name
Caption	Number	
Text Field		ObxPhoneUI.phone.number
Check Box	Lookup by Name	ObxPhoneUI.phone.lookupByName
Command Button	Lookup	ObxPhoneUI.Lookup

Table 4-10. Links of phone book dialog box

Note that the previously disabled controls now have become enabled. What does this mean? When a control is linked, which normally happens when it is being read from a file, or in our case when the link is changed by the inspector, then the module to which the control should be linked must be loaded. If it is already loaded, nothing needs to be done. If it isn't loaded yet (remember that you can check with *Info->Loaded Modules*), loading is done now. If loading fails, e.g., because the module's code file doesn't yet exist, then control linking fails and the control remains disabled. Linking also fails if the control and field types don't match, e.g., if a check box is linked to a string field.

To achieve this level of functionality (and safety against incorrect use), BlackBox provides several advanced "metaprogramming" services, in particular dynamic module loading on demand and typesafe lookup of variables. The latter requires extensive run-time type information (RTTI) that is relatively uncommon in fully compiled languages.
The links of all controls are reevaluated whenever a module has been unloaded. This ensures that controls are never linked to unloaded modules.

It was one of the design goals for BlackBox to separate user interface details from program logic. For this reason, module *ObxPhoneUI* doesn't know about controls and forms and the like. Instead, the controls of our form have links, which tell them the interactor fields with which they should interact. For example, the command button's "ObxPhoneUI.Lookup" link tells it to activate the *ObxPhoneUI.Lookup* procedure when the button is pressed. This procedure in turn doesn't know about the command button (there even may be several of them), the only thing it does to acknowledge the possible existence of controls is to call

```
Dialog.Update(phone)
```

at the end of the command procedure. *Dialog.Update* causes an update of all controls that need updating. For example, if *Lookup* has assigned "not found" to *phone.number*, the corresponding text field(s) or similar controls need to be redrawn accordingly.
As parameter of *Dialog.Update*, an interactor must be passed. Calling *Dialog.Update* is necessary after one or

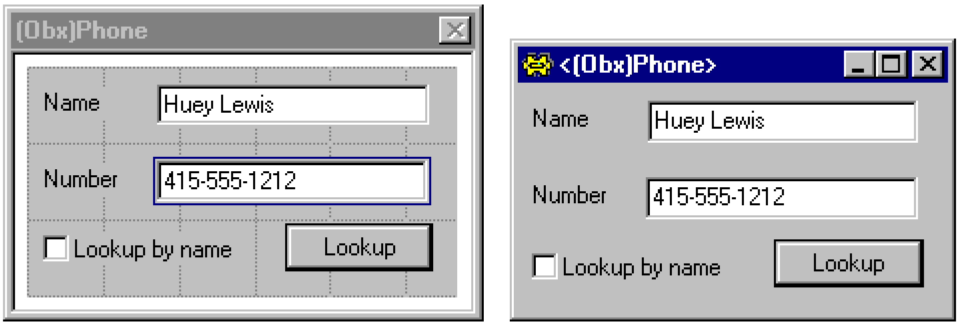
several fields of this interactor have been modified by a program. If several fields have been modified, *Dialog.Update* should only be called once, for efficiency reasons. Note that a control calls *Dialog.Update* itself when the user has modified an interactor field; you only need to call it after your own code has modified the interactor.

This strong separation of user interface from program logic is uncommon. Its advantage is simplicity: as soon as you know how to define procedures and how to declare record types and global variables, you can already construct graphical user interfaces for modules. This is possible even for someone who is just beginning to learn programming. Another advantage is that you have to write no code for simple user interfaces. User interface construction happens in the forms editor (e.g., setting the position and size of a control) and with the inspector (e.g., setting the alignment of text in a text field). This makes it easier to adapt an application to different user interface requirements, without touching the application logic itself. Only if you want to exercise more control over the user interface, e.g., disabling controls or reacting on special events such as the user's typing, then you need to write small amounts of code, which can be very cleanly separated from the application logic itself. The necessary concepts, so-called guards and notifiers, will be discussed in the next two sections. If you need still more control, then you can access controls individually, as described in section 4.9.

Currently, a disavantage of the BlackBox Component Builder's approach is that all controls have to be linked to global interactor variables. If there are several controls for the same interactor field, all of them display the same value. The controls cannot have independent state of their own.

Note an interesting feature of BlackBox: if you have written a module like *ObxPhoneUI* in Listing 4-9, you can automatically generate a form with suitable controls in a default layout. This is done by clicking "Create" in the "New Form" dialog box instead of "Empty". This feature is useful to create temporary testing and debugging user interfaces during development, where it isn't useful to spend time with manual form construction.

We now have a dialog box layout with controls linked to module *ObxPhoneUI*, and this module imports the database engine *ObxPhoneDB*. What is still missing is a way to *use* the dialog box, rather than to merely edit its layout. During editing, it can be useful to immediately try out the dialog box, even before its layout is perfect. To try this out, make sure that the layout window is on top and then execute *Controls->Open As Aux Dialog*. A new window is opened which contains the same dialog box, but in a way that its *controls* can be edited, rather than its *layout*. The window acts as a data entry mask. Now you can type, for example, "Huey Lewis" into the name string, click on the "Lookup by Name" check box, and then click on the "Lookup" button. You'll see that the appropriate phone number appears in the "Number" field.



Name Huey Lewis
Number 415-555-1212
□ Lookup by name

Name Huey Lewis
Number 415-555-1212
□ Lookup by name

Figure 4-11. Layout view (left) and mask view (right) displaying the same form model

Note that the same name and number also appeared in the layout window. Even better, if you change the layout in the layout window, e.g., by moving the check box somewhat, you'll note that the layout change is immediately reflected in the other window. This is a result of the so-called Model-View-Controller implementation of BlackBox. In Part II of the book, we have discussed this design pattern in more detail. Here it is sufficient to note that several views can share the same data, e.g., a layout view and a mask view can display the same form; and that both layout and mask views are basically the same kind of view albeit in different modes. You can switch between these modes by applying the *Dev->Layout Mode* or *Dev->Mask Mode* commands.

The two modes differ in the ways they treat selection and focus. In layout mode, you can select the embedded views and edit the selection, but you cannot focus the embedded views. In mask mode, you can focus the embedded views, but you cannot select them (only their contents) and thus cannot edit their layout, i.e., their sizes, positions, etc. In other words: layout mode prevents focusing, while mask mode prevents selection.

Opening a second view in mask mode for our form layout is convenient during layout editing, but you wouldn't want any layout view open when your program should actually be *used*. In this case, you want to open the dialog box in mask mode by invoking a suitable menu command.

A new menu command can be introduced by editing a menu configuration text. You can open this text (which resides in System/Rsrc/Menus) by calling *Info->Menus*. Append the following text to its end:

```
MENU "Priv"
   "Open..."    ""    "StdCmds.OpenAuxDialog('Obx/Rsrc/PhoneUI', 'Phonebook')"    ""
END
```

Having done this, execute *Info->Update Menus*. You'll notice that the new menu "Priv" has appeared. Execute its menu item "Open...". As a result, the command

```
StdCmds.OpenAuxDialog('Obx/Rsrc/PhoneUI', 'Phonebook')
```

will be executed. It opens the Obx/Rsrc/PhoneUI layout document, turns it into mask mode, and opens it in a window with title "Phonebook". If you want the modification of your menu text to become permanent, save the "Menus" text before closing it.

Note that the form has not been saved in mask mode (this would be inconvenient for later editing), it is only temporarily turned into mask mode by the *StdCmds.OpenAuxDialog* command.

4.4 Guards

In terms of genuine functionality, we have seen everything that is important about the standard set of controls. We will look at the palette of standard controls in more detail later. However, we first need to discuss important aspects of standard controls which, strictly speaking, do not increase the functionality of an application, but rather its useability, i.e., its user-friendliness.

In section 1.2 we have already discussed what user-friendliness means. For example, it means avoiding modes wherever possible. For this reason, BlackBox doesn't support modal dialog boxes.
Modes are unavoidable if a user action sometimes makes sense, and sometimes doesn't. For example, if the clipboard is empty, its contents cannot be pasted into a text. If it is not empty and contains text, pasting is possible. This cannot be helped, and is harmless if the current state is clearly visible or can easily be inquired by the user. For example, if the clipboard is empty, the *Paste* menu command can be visibly marked as disabled. The visual distinction gives the user early feedback that this command is currently not meaningful. This is usually much better than to let the user try out a command and then give an error message afterwards.
The following example shows a dialog box with two buttons. The *Empty* button is enabled, while the *Create* button is disabled. The *Create* button only becomes enabled if something has been typed into the dialog box's text field:

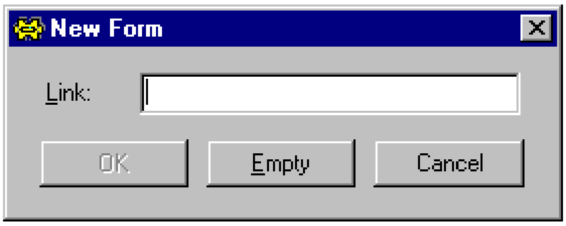


Figure 4-12. Enabled and disabled buttons

In summary, a good user interface always lets the user perform every meaningful action, and gives visual cues about actions that are not meaningful.

For the BlackBox, we thus need a way to provide feedback about the current state of the system, especially about which commands are currently possible and which aren't.

For this purpose, it must be possible to enable and disable controls and menu items. For example, looking up a phone number is only possible if some name has been entered, i.e., if the name field is not empty. To determine whether a command procedure, in our case the *Lookup* procedure, may be called, i.e., whether a corresponding control or menu item may be enabled, a suitable *guard* must be provided. A guard is a procedure called by the framework, whenever it might be necessary to change the state of a control or menu item. The guard inspects some global state of the system, uses this state to determine whether the guarded command currently makes sense, and then sets up an output parameter accordingly. A guard has the following form:

```
PROCEDURE XyzGuard* (VAR par: Dialog.Par);
BEGIN
    par.disabled := ...some Boolean expression...
END XyzGuard;
```

A guard has the following type:

```
GuardProc = PROCEDURE (VAR par: Dialog.Par);
```

To guard procedure *Lookup* in our example, we extend module *ObxPhoneUI* in the following way:

```
MODULE ObxPhoneUI;

    IMPORT Dialog, ObxPhoneDB;

    VAR
        phone*: RECORD
            name*, number*: ObxPhoneDB.String;
            lookupByName*: BOOLEAN
        END;

    PROCEDURE Lookup*;
    BEGIN
        IF phone.lookupByName THEN
            ObxPhoneDB.LookupByName(phone.name, phone.number);
            IF phone.number = "" THEN phone.number := "not found" END
        ELSE
            ObxPhoneDB.LookupByNumber(phone.number, phone.name);
            IF phone.name = "" THEN phone.name := "not found" END
        END;
        Dialog.Update(phone)
    END Lookup;

    PROCEDURE LookupGuard* (VAR par: Dialog.Par);
    BEGIN    (* disable if input string is empty *)
        par.disabled := phone.lookupByName & (phone.name = "") OR
                            ~phone.lookupByName & (phone.number = "")
    END LookupGuard;

END ObxPhoneUI.
```

Listing 4-13. ObxPhoneUI with LookupGuard

What happens if we compile the above module? Its symbol file on disk is replaced by a new version, because the module interface has been changed from the previous version. Because the change is merely an addition of a global procedure (i.e., the new version is compatible with the old version) possible client modules importing *ObxPhoneUI* are not invalidated and need not be recompiled.

Compilation also produced a new code file on disk. However, the old version of *ObxPhoneUI* is still loaded in memory! In other words: once loaded, a module remains loaded ("terminate-and-stay-resident"). This is not a problem, since modules are extremely light-weight and consume little memory. However, a programmer of course must be able to unload modules without leaving the BlackBox Component Builder entirely, in order to try out a new version of a module. For this purpose, the command *Dev->Unload* is provided which unloads the module whose source code is currently focused.

Note that compilation does not automatically unload a module, since this is often undesirable. In particular, as soon as you work on several related modules concurrently, unloading one of them before the others are correctly updated would render this whole set of modules inconsistent.

For those cases where immediate unloading after compilation *does* make sense, like in the simple examples that we are currently discussing, the command *Dev->Compile And Unload* is provided.

You may use this command to try out our new version of *ObxPhoneUI*. Notice how the *Lookup* button is disabled when the *Name* field is empty (if *Lookup by Name* is chosen) or when the *Number* field is empty (if *Lookup by Number* is chosen). Typing something into the field makes *Lookup* enabled again; deleting all characters in the field disables it again. At the end of this section we will explain more precisely when guards are evaluated; for now it is sufficient to know that they are evaluated after every character typed into a text field control.

Guards are mostly used to enable and disable user interface elements such as controls or menu items. However, they sometimes play a more general role as well. For example, controls may not only be disabled, but they also may be made read-only or undefined.

Read-only means that a control currently cannot be modified interactively. For example, the following guards set up the output parameter's *readOnly* field. The first guard sets read-only if lookup is not by name, i.e., by number. Obviously, in this case a number is input and a name is output. Pure outputs should be read-only. Thus the first guard can be used as guard for the *Name* field. The second guard can be used for the *Number* field, since it sets read-only if lookup returns a number.

```
    PROCEDURE NameGuard* (VAR par: Dialog.Par);
    BEGIN    (* make read-only if lookup is by number *)
        par.readOnly := ~phone.lookupByName
    END NameGuard;

    PROCEDURE NumberGuard* (VAR par: Dialog.Par);
```

```
BEGIN    (* make read-only if lookup is by name *)
    par.readOnly := phone.lookupByName
END NumberGuard;
```

Listing 4-14. Read-only guards for ObxPhoneUI

Interactor fields which are exported read-only, i.e., with the "-" export mark instead of the "*" export mark, are always in read-only state regardless of what a guard specifies (otherwise this would violate module safety, i.e., the invariant that a read-only exported item can only be modified within its defining module).

The *undefined* state of a control means that the control currently has no meaning at all. This happens if a control displays the state of a heterogeneous selection. For example, a check box may indicate whether the text selection is all caps or all small letters. If part of the selection is capital letters and the rest small letters, then the control has no defined value. However, by clicking on the check box, the selection is made all caps and then has a defined state again. The undefined state can be set by a guard with the statement

```
par.undef := TRUE
```

The undefined state can be regarded as "write-only", i.e., the control's state cannot be read by the user because it currently has no defined value, but it can be modified and thus set to a defined value.

Of the fields *disabled*, *readOnly* and *undef*, at most one may be set to *TRUE* by a guard. This leads to four possible temporary states of the control: it is in none or in exactly one of the three special states. When a guard is called by the framework, all three Boolean fields are preset to *FALSE*. This is a general rule in BlackBox: Boolean values default to *FALSE*.

In Table 4-15, suitable guards for the layout of Figure 4-2 are listed:

Control type	*Link*	*Guard*
Caption		
Text Field	ObxPhoneUI.phone.name	ObxPhoneUI.NameGuard
Caption		
Text Field	ObxPhoneUI.phone.number	ObxPhoneUI.NumberGuard
Check Box	ObxPhoneUI.phone.lookupByName	
Command Button	ObxPhoneUI.Lookup	ObxPhoneUI.LookupGuard

Table 4-15. Guards in phone book dialog box

A guard applies to a procedure, to an interactor field, or to several procedures or interactor fields. If it applies to mainly one procedure or field, the guard's name is constructed by appending "Guard" to the (capitalized) name of the procedure/field. For example, the guard for procedure *Lookup* is called *LookupGuard*, the guard for field *phone.name* is called *NameGuard*, etc. This is a simple naming convention which makes it easier to recognize the relation between guard and guarded item. The naming convention is "soft", i.e., it is not enforced by the framework; in fact the framework doesn't know about it at all.

It is not a convention, but a necessity, to export guards. Guards are accessed by the so-called metaprogramming mechanism of BlackBox which, for safety reasons, only operates on exported items. This is consistent with the treatment of a module as a black-box, of which only the set of exported items, i.e., its interface, is accessible from the outside. If a guard isn't exported, a control cannot call it. This is similar to the fields of an interactor, which also must be exported if an interactor should be able to link to it.

If we look at the declaration of type *Dialog.Par* (use *Info->Interface*!), we see that two fields have not yet been discussed: *label* and *checked*:

```
Par = RECORD
   disabled, checked, undef, readOnly: BOOLEAN;
   label: Dialog.String
END
```

Field *label* allows to change the label of a control. For example, instead of using the label "Toggle" for a button it may be more telling to use the labels "Switch On" and "Switch Off" depending on the current state of the system. This can be done by a procedure like this:

```
PROCEDURE ToggleGuard* (VAR par: Dialog.Par);
BEGIN
   IF someInteractor.isOn THEN
      par.label := "Switch Off"
   ELSE
      par.label := "Switch On"
   END
END ToggleGuard;
```

Listing 4-16. Label guard example

Note that the guard overrides whatever label was set up in the control property inspector. This is true for all controls (and for menu items as well, see below).

It is strongly recommended not to place string literals in the source code like in the above example, because this would force a recompilation of the code if the language were changed, e.g., from English to German (see also section 1.4). In BlackBox, user interface strings such as labels or messages are generally packed into separate parameter files, so-called string resources. For each subsystem there can be one string resource file, e.g., Text/Rsrc/Strings or Form/Rsrc/Strings. A string resource file is a BlackBox text document starting with the keyword *STRINGS* and followed by an arbitrary number of *<key, string>* pairs. The key is a string, separated by a tab character from the actual string into which it will be mapped. Every *<key, string>* pair must be terminated by a carriage return. The pairs need not be arranged in any particular order, although it is helpful to sort them alphabetically by key, because this makes it easier to find a particular key when editing the string resources.
For example, System/Rsrc/Strings starts the following way:

STRINGS	
About	About BlackBox
AlienAttributes	alien attributes
AlienCause	alien cause
AlienComponent	alien component
AlienControllerWarning	alien controller (warning)
...	

Table 4-17. String resources of System/Rsrc/Strings

To use string resources in our guard example, a special syntax must be used to indicate that the string is actually a key that first must be mapped using the appropriate subsystem's string resources. Assuming that in the *Obx* subsystem's string resources there exist "On" and "Off" keys, the following code emerges:

```
PROCEDURE ToggleGuard* (VAR par: Dialog.Par);
BEGIN
    IF someInteractor.isOn THEN
        par.label := "#Obx:Off"
    ELSE
        par.label := "#Obx:On"
    END
END ToggleGuard;
```

Listing 4-18. Label guard example with string mapping

The leading "#" indicates that a string mapping is desired. It is followed by the subsystem name, in this case "Obx". Then comes a colon, followed by the key to be mapped. A command button with this guard would either display the label "Switch Off" or "Switch On" in an English version of BlackBox, "Ausschalten" or "Einschalten" in a German version, and so on.
If there is no suitable string resource, the key is mapped to itself. For example, if there is no "Off" key in Obx/Rsrc/Strings, then "#Obx:Off" will be mapped to "Off".

The remaining field of *Dialog.Par* that we have not yet discussed is called *checked*. Actually, so far it has never been used for controls in BlackBox. It is used for menu items. Menu items are similar to controls: they can invoke actions, they may be enabled or disabled, and they have labels. For this reason it makes sense to use the same guard mechanism for them also. A menu guard is specified in the *Menus* text as a string after the menu label and the keyboard equivalent of the menu item. For example, the following entry for the *Dev* menu specifies the guard *StdCmds.SetEditModeGuard*:

```
"Edit Mode"    ""    "StdCmds.SetEditMode"    "StdCmds.SetEditModeGuard"
```

A unique feature of menu items is that they may be checked. For example, in the following menu, menu item *Edit Mode* is checked:

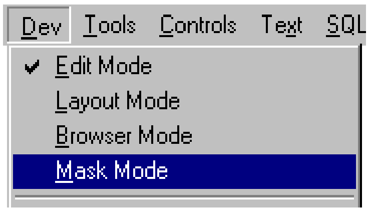


Figure 4-19. Menu item with check mark

The check mark indicates which of the items has been selected most recently, and the state that has been established by the selection. The state can be changed by invoking one of the other menu items, e.g., *Mask Mode* as in the figure above. Basically, the four menu items form a group of possibilities from which one can be selected.
Guard procedures for menu items may set up the *disabled*, *checked* and *label* fields of *Dialog.Par*. The other fields are ignored for menus.

A guard procedure may set up several fields of its *par* output parameter simultaneously, e.g., it may assign the *disabled* and *label* fields for a command button. However, a guard may set at most one of the Boolean fields of *par* and must *never* modify any state outside of *par*, e.g., a field of an interactor or something else; i.e., it must have no side effects. It may not call any procedures which may have side effects either. The reason is that a program cannot assume much about when (and especially when not) a guard is being called by the framework.
A guard may use any interactor or set of interactors as its input, or the state of the current focus or selection.

The latter is often used for menu guards. The current focus or selection is only used in control guards if the controls are in so-called tool dialog boxes, i.e., dialog boxes that operate on some document underneath. A *Find & Replace* dialog box operating on a focused text is a typical example of a tool dialog box. The other dialog boxes are self-contained and called auxiliary dialog boxes. Data entry masks are typical examples of auxiliary dialog boxes.

When is a guard evaluated? There are four reasons why a guard may be called: when the control's link is established, when the contents of an interactor is being edited, when the window hierarchy has changed, or when the user has clicked into the menu bar.

A control's link is established after it is newly inserted into a container, after it has been loaded from a file, after a module was unloaded, or after its link has been modified through the control property inspector or other tool.

When some piece of code modifies the contents of an interactor, it is required to call *Dialog.Update* for this interactor. As a result, every currently visible control is notified (see section 2.9). In turn, the control compares its actual state with the interactor field to which it is linked. If the interactor field has changed, the control redraws itself accordingly. After all controls have updated themselves, the guards of *all* visible controls are evaluated. For this reason, guards should be efficient and not perform too much processing.

Guards are also evaluated when the window hierarchy is changed, e.g., when a bottom window is brought to the top. This is necessary because many commands depend on the current focus or selection, which vanishes if another window comes to the top.

Menus are another reason why a guard may be called. When the user clicks in a menu bar, all guards of this menu, or even the guards of the whole menu bar, are evaluated.

Usually, a guard has the form

```
PROCEDURE SomeGuard* (VAR par: Dialog.Par)
```

Alternatively, the form

```
PROCEDURE SomeGuard* (n: INTEGER; VAR par: Dialog.Par)
```

may be used, which allows to parameterize a single guard procedure for several related commands. For example, the commands to set a selection to the colors red, green or blue are the following:

```
StdCmds.Color(00000FFH)
StdCmds.Color(000FF00H)
StdCmds.Color(0FF0000H)
```

For these commands, the following guards can be used:

```
StdCmds.ColorGuard(00000FFH)
StdCmds.ColorGuard(000FF00H)
StdCmds.ColorGuard(0FF0000H)
```

The actual signature of *StdCmds.ColorGuard* is

```
PROCEDURE ColorGuard (color: INTEGER; VAR par: Dialog.Par)
```

4.5 Notifiers

A control guard sets up the temporary state of a control, such as whether it is disabled or not. It has no other effect. It doesn't modify any interactor state. It doesn't add any functionality. It only lets the control give feedback about the currently available functionality, or its lack thereof. A guard is evaluated ,e.g., when the user changes the state of a control interactively. It can be regarded as a merely "cosmetic" feature whose sole purpose is to increase user-friendliness.

However, sometimes a user interaction should trigger more than the evaluation of guards only. In particular, it may sometimes be necessary to change some interactor state as a response to user interaction. For example, changing the selection in a selection box may cause the update of a corresponding counter, which counts the number of currently selected items in the selection box. Or some postprocessing of user input may be implemented, as we will see in the following example. It is an alternate implementation of our *ObxPhoneUI* example. Instead of having a *Lookup* button, this version just has two edit fields. After any character typed in, a database lookup is executed to test whether now a correct key is entered. Note that unsuccessful lookup in *ObxPhoneDB* results in returning an empty string.

```
MODULE ObxPhoneUI1;

   IMPORT Dialog, ObxPhoneDB;

   VAR
      phone*: RECORD
         name*, number*: ObxPhoneDB.String
      END;

   PROCEDURE NameNotifier* (op, from, to: INTEGER);
   BEGIN
      ObxPhoneDB.LookupByName(phone.name, phone.number);
      Dialog.Update(phone)
   END NameNotifier;

   PROCEDURE NumberNotifier* (op, from, to: INTEGER);
   BEGIN
      ObxPhoneDB.LookupByNumber(phone.number, phone.name);
      Dialog.Update(phone)
   END NumberNotifier;

END ObxPhoneUI1.
```

Listing 4-20. ObxPhoneUI1 with notifiers

These *notifiers* create dependencies between two fields of an interactor: a modification of one text field may lead to a modification of the other text field, i.e., a dependency between interactor fields is defined.

Notifiers are called right after an interaction happens, but before the guards are evaluated. A notifier has the following form:

```
   PROCEDURE XyzNotifier* (op, from, to: INTEGER);
   BEGIN
      ...
   END XyzNotifier;
```

A notifier's type is

```
   NotifierProc = PROCEDURE (op, from, to: INTEGER);
```

For simple notifiers, the parameters can be ignored. They give a more precise indication of what kind of modification actually took place. Which of the parameters are valid and what their meaning exactly is is defined separately for every kind of control. See the next section for details.

4.6 Standard controls

In this section, the various controls that come standard with the BlackBox Component Builder are presented. For each control, a list of data types to which it may be linked is given, and the meaning of the notifier parameters is indicated.
A control may have various properties, e.g., its label, or whether the label should be displayed in the normal system font or in a lighter, less flashy font. However, such cosmetic properties may not be implemented the same way on every platform, sometimes they may even be ignored.

A label may specify a keyboard shortcut, which on some platforms is used to navigate between controls without using the mouse. The shortcut character is indicated by preceding it with a "&" sign. Two successive "&" signs indicate a normal "&", without interpreting the following character as shortcut. Only one shortcut per label is allowed. For example, "Clear Text &Field" defines "F" as a keyboard shortcut, while "Find && &Replace" defines "R" as keyboard shortcut. The same syntax for keyboard shortcuts is used in menu commands.

The label property of every control can be modified at run-time, by using a guard procedure as demonstrated in Listing 4-16. For controls without visible label, this has no visible effect.

Two of the most important control properties, the guard and notifier names, are optional. If there is no guard for the control, it will use its default states (enabled, read-write, defined). If there is no notifier for the control, none will be called.
BlackBox attempts to minimize the number of different properties, in order to make it easier to use. This is consistent with the general trend, both under Windows and the Mac OS, towards globally controlled appearances. This means that a control should not define its colors, font, and so on individually. Instead, the user should be able to define a system-wide and consistent configuration of these properties.

However, if for some special reason this is still deemed important, specific fonts can be assigned to controls, simply by selecting the control(s) and applying the commands of the *Font*, *Attributes* or *Characters* menus. If the default font is set, then the system preferences will be used. This is the normal case. Note that other font attributes, such as styles, weight, or size, may be restricted on certain platforms. For example, on Mac OS the font size is always 12 points.

In the following text, all standard controls of BlackBox are described in turn: command buttons, edit fields, check boxes, radio buttons, date fields, time fields, color fields, captions, groups, list boxes, selection boxes and combo boxes. For each control, its valid properties are given, the variable types to which it may be linked, and the meaning of the *op*, *from* and *to* parameters of the notifier.

Note that there is no special control for currency values, you can use edit fields bound to variables of type *Dialog.Currency* instead.

In the following descriptions, some abbreviations are used:
pressed stands for *Dialog.pressed*
released stands for *Dialog.released*
changed stands for *Dialog.changed*
included stands for *Dialog.included*
excluded stands for *Dialog.excluded*

undefined means that the *from* or *to* parameter of a notifier has no defined value that can be relied upon
modifier means a 0 for single-clicks and a 1 for double-clicks
link, *label*, *guard*, *notifier*, and *level* stand for their respective control properties as defined in module *Controls*. There are up to five optional Boolean properties. Depending on the control, they are called differently, e.g. *default font*, *default*, *cancel*, *sorted*, *left*, *right*, *multiLine*, *password*.

Command Button

Figure 4-21. Command button

A command button is used to execute a parameterless Component Pascal command, or a whole command sequence. A button with the default property looks different than normal buttons; input of a carriage return corresponds to a mouse click in the default button. For a button with the cancel property, input of an escape character corresponds to a mouse click in it. A button should not be default and cancel button simultaneously. Note that the default and cancel properties are set and cleared individually per control, i.e., making one button a default button doesn't automatically make an existing default button a normal button again. There should be at most one default and at most one cancel button per dialog box.
A cancel button or another button that should close the dialog box in which it is contained can use the command *StdCmds.CloseDialog*.

properties:	link, label, guard, notifier, font, default, cancel
linkable to:	parameterless procedure, command sequence
op:	pressed, released
from:	undefined, modifier
to:	undefined

Text Field

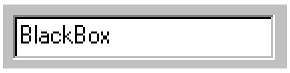


Figure 4-22. Text field

A text field displays the value of a global variable which may be a string, an integer, a floating-point number, or a variable of type *Dialog.Currency* or *Dialog.Combo*. The value of the variable can be changed by editing the field.
Whenever the contents of the linked interactor field is changed, the notifier is called. Key presses that do not change the interactor state, such as pressing arrow keys or entering leading zeroes in number fields, cause no notifier call. Changing the selection contained in the field causes no notification either.
If a modification occurs in a field that is linked to a string, real, currency, or combo variable, the notifier with *(change, undefined, undefined)* is called. If the field is linked to an integer value, the notifier with *(change, oldvalue, newvalue)* is called.
Illegal characters, e.g., characters in a field linked to an integer, are not accepted.
A text field may have a label, even though it doesn't display it. The reason for this is that keyboard shortcuts may be defined in labels, which is useful even for edit fields.
If the *level* property is 0 (the default), or the control is not linked to a number type, or it is linked to *Dialog.Currency*, then *level* has no effect. When linked to an integer variable the *level* defines the scale factor used for displaying the number, i.e., the displayed number is the linked value divided by 10^{level}. For example, if

the current value is 42 and level is 2, then 0.42 is displayed. For variable of real type, *level* indicates the format in the following way:

level > 0: exponential format (scientific) with at least *level* digits in the exponent.
level = 0: fixpoint or floatingpoint format, depending on *x*.
level < 0: fixpoint format with *-level* digits after the decimal point.

The *left* and *right* properties define the adjustment mode of the field. The following combinations are possible:

left & ~right	left adjust
left & right	fully adjusted (may have no effect on some platforms)
~left & ~right	centered
~left & right	right adjust

The default is *left & ~right* (left adjust).
Property *multiLine* defines whether a carriage return, with its resulting line break, may be accepted by the field, which then must be linked to a string variable.
Property *password* causes the text field to display only asterisks instead of the characters typed in. This makes it possible to use such a field for password entry.

properties:	link, label, guard, notifier, level, font, left, right, multiLine, password
linkable to:	ARRAY const OF CHAR, BYTE, SHORTINT, INTEGER, LONGINT, SHORTREAL, REAL, Dialog.Currency, Dialog.Combo
op:	pressed, released, changed
from:	undefined, old value, modifier
to:	undefined, new value

Check Box

Figure 4-23. Check box

A check box displays the value of a global variable which may be a Boolean or an element of a set. Clicking on the control toggles its state.
When the control's state is changed, the notifier with *(changed, undefined, undefined)* is called if the control is linked to a Boolean variable. If it is linked to an element of a set variable, *(included, level, undefined)* or *(excluded, level, undefined)* is called, depending on whether the bit is set or cleared. The value *level* corresponds to the element of the set to which the control is linked. It can be defined using the control property inspector. It may lie in the range 0..31.
Only the last and final state change of the control leads to a notifier call, possible intermediate state changes (by dragging the mouse outside of the control's bounding box or back inside) have no effect.

properties:	link, label, guard, notifier, font, level
linkable to:	BOOLEAN, SET
op:	pressed, released, changed, included, excluded
from:	undefined, level value, modifier
to:	undefined

Radio Button

Figure 4-24. Radio button

A radio button is active at a particular value of a global integer or Boolean variable. Typically, several radio buttons are linked to the same variable. Each radio button is "on" at another value, which is defined by the level property; i.e., the button is "on" if its level value is equal to the value of the variable it is linked to. For Boolean types, "on" corresponds to *TRUE* and "off" corresponds to *FALSE*.
Only the last and final state change of the control leads to a notifier call, possible intermediate state changes (by dragging the mouse outside of the control's bounding box or back inside) have no effect.

properties:	link, label, guard, notifier, font, level
linkable to:	BYTE, SHORTINT, INTEGER, LONGINT, BOOLEAN
op:	pressed, released, changed
from:	undefined, old value, modifier
to:	undefined, new value = level value

Date Field

Figure 4-25. Date field

A date field displays the date specified in a global variable of type *Dates.Date*.
Whenever the contents of the linked interactor field is changed, the notifier is called with *(change, undefined, undefined)*. Key presses that do not change the interactor state, such as pressing left/right arrow keys, cause no notifier call. The up/down arrow keys change the date. Changing the selection in the field has no effect.
Illegal date values cannot be entered.

properties:	link, label, guard, notifier, font
linkable to:	Dates.Date
op:	pressed, released, changed
from:	undefined, modifier
to:	undefined

Time Field

Figure 4-26. Time field

A time field displays the time specified in a global variable of type *Dates.Time*.
Whenever the contents of the linked interactor field is changed, the notifier is called with *(change, undefined, undefined)*. Key presses that do not change the interactor state, such as pressing left/right arrow keys, cause no notifier call. The up/down arrow keys change the time. Changing the selection in the field has no effect.
Illegal time values cannot be entered.

properties:	link, label, guard, notifier, font
linkable to:	Dates.Time
op:	pressed, released, changed
from:	undefined, modifier
to:	undefined

Color Field

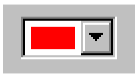


Figure 4-27. Color field

A color field displays a color. It can be linked to variables of type *INTEGER* or *Dialog.Color*. *Ports.Color* is an alias of *INTEGER* and thus can be used also.
Whenever another color is selected, the notifier is called with *(change, oldval, newval)*. The old and new values are integer values; for *Dialog.Color* type variables the *val* field's value is taken.

properties:	link, label, guard, notifier, font
linkable to:	Dialog.Color, Ports.Color = INTEGER
op:	pressed, released, changed
from:	undefined, old value, modifier
to:	undefined, new value

Up/Down Field

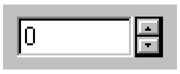


Figure 4-28. Up/down field

This is a field linked to an integer variable. The value can also be changed through arrow keys.
Whenever the contents of the linked interactor field is changed, the notifier is called with *(change, oldvalue, newvalue)*.

properties:	link, label, guard, notifier, font
linkable to:	BYTE, SHORTINT, INTEGER, LONGINT
op:	pressed, released
from:	undefined, old value, modifier
to:	undefined, new value

Caption

Figure 4-29. Caption

A caption is typically used in conjunction with a text field, to indicate the nature of its contents. A caption is passive, i.e., it cannot be edited and thus cannot have a notifier.
A caption is linkable to the same types as a text field is, and it may have a guard. This is useful since - depending on the platform - a caption may have a distinct visual appearance if it (or rather its corresponding

text field) is disabled, read-only, etc. This means that a caption may be disabled along with its corresponding text field, by linking both to the same interactor field. A caption's guard may modify the control's label, as is true for all controls with a visible label.

properties:	link, label, guard, font, right, left
linkable to:	ARRAY const OF CHAR, BYTE, SHORTINT, INTEGER, LONGINT, SHORTREAL, REAL, Dialog.Currency, Dialog.Combo

The *left* and *right* properties define the adjustment mode of the caption. The following combinations are possible:

left & ~right	left adjust
left & right	fully adjusted (may have no effect on some platforms)
~left & ~right	centered
~left & right	right adjust

The default is *left & ~right* (left adjust).

Group

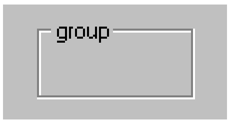


Figure 4-30. Group

A group is used to visually group related controls, e.g., radio buttons that belong together. A group is passive, i.e., it cannot be edited and thus cannot have a notifier.
A group may not be linked, but it may have a guard which allows to disable or enable it. A group's guard may modify the control's label, as is true for all controls with a visible label.

properties:	label, guard, font

List Box

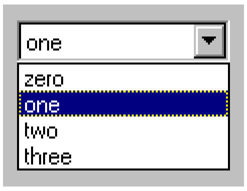


Figure 4-31. List box (expanded and collapsed shapes)

A list box allows to select one value out of a list of choices. It is linked to a variable of type *Dialog.List* (see next section). If the height of the list box is large enough, a scrollable list is displayed. If it is not large enough, the box collapses into a pop-up menu.
Interactively, the selection can be changed. It is either empty or it consists of one selected item. When the user modifies the selection, the notifier is called with *(changed, oldvalue, newvalue)*. The old/new value corresponds to the selected item's index. The top-most item corresponds to value 0, the next one below to

value 1, etc. The empty selection corresponds to value -1.
The *sorted* property determines whether the string items will be sorted lexically (no effect on Mac OS).

properties:	link, label, guard, notifier, font, sorted
linkable to:	Dialog.List
op:	pressed, released, changed
from:	undefined, old value, modifier
to:	undefined, new value

Selection Box

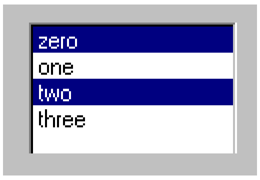


Figure 4-32. Selection box

A selection box allows to select a subset out of a set of choices. It is linked to a variable of type *Dialog.Selection* (see next section).
Interactively, the selection can be changed. Each item may be selected individually. When the user modifies the selection, the notifier is called in one of three ways:
(included, from, to): range *from..to* is now selected; it wasn't selected before
(excluded, from, to): range *from..to* is not selected now; it was selected before
(set, from, to): range *from..to* is now selected; any previous selection was cleared before
The three codes *included* and *excluded* and *set* take the place of *changed* used for most other controls. The notifier is called as often as necessary to include and/or exclude all necessary ranges of items.
The from/to value corresponds to the selected item's index. The topmost item corresponds to value 0, the next one below to value 1, and so on.
The *sorted* property determines whether the string items will be sorted lexically (no effect on Mac OS).

properties:	link, label, guard, notifier, font, sorted
linkable to:	Dialog.Selection
op:	pressed, released, included, excluded, set
from:	undefined, lowest element of range, modifier
to:	undefined, highest element of range

Combo Box

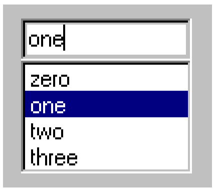


Figure 4-33. Combo box

A combo box is a text field whose contents can also be set via a pop-up menu. Unlike pure pop-up menus/selection boxes, a value may be entered which does not occur in the pop-up menu. The control is linked to a variable of type *Dialog.Combo* (see next section).

When the contents of the combo is changed, the notifier is called with *(changed, undefined, undefined)*.
The *sorted* property determines whether the string items will be sorted lexically (no effect on Mac OS).

properties:	link, label, guard, notifier, font, sorted
linkable to:	Dialog.Combo
op:	pressed, released, changed
from:	undefined, modifier
to:	undefined

Each interactive control (i.e., not a caption or a group) calls its notifier - if there is one - when the user first clicks in the control with parameter *op = Dialog.pressed*, and later with *op = Dialog.released* when the mouse button is released again. This feature is used mostly to display some string in the dialog box window's status area. This is done with the calls *Dialog.ShowStatus* or *Dialog.ShowParamStatus*. For example, the following notifier indicates a command button's function to the user:

```
PROCEDURE ButtonNotifier* (op, from, to: LONGINT);
BEGIN
  IF op = Dialog.pressed THEN
    Dialog.ShowStatus("This button causes the disk to spin down")
  ELSIF op = Dialog.released THEN
    Dialog.ShowStatus("") (* clear the status message again *)
  END
END ButtonNotifier;
```

Listing 4-34. Notifier displaying status messages

On some platforms, e.g., on Mac OS, there is no status area and the above code has no effect.
Sometimes it is useful to detect double-clicks, e.g., a double-click in a list box may select the item and invoke the default button. A double-click can be detected in a notifier with the test

```
IF (op = Dialog.pressed) & (from = 1) THEN ... (* double-click *)
```

For the above-mentioned case, where the reaction on a double-click should merely be the invocation of the default button, a suitable standard notifier is available:

```
StdCmds.DefaultOnDoubleClick
```

We have seen earlier that the BlackBox Component Builder provides a string mapping facility which maps keys to actual strings, by using resource files. This feature is also supported by *Dialog.ShowStatus*. In fact, string mapping even allows to use place holders. For example, calling

```
Dialog.ShowParamStatus("This ^0 causes the ^1 to ^2", control, object, verb)
```

allows to supply different strings for the place holders ^0, ^1 and ^2. The strings actually used are the three additional parameters *control, object* and *verb*. Note that these strings are mapped themselves before being spliced into the first string. String mapping is a feature that you can use explicitly by calling the procedure *Dialog.MapParamString*.

4.7 Complex controls and interactors

A list box has two kinds of contents: the string items which make up the list, and the current selection.

Typically, the item list is more persistent than the selection, but it too may be changed while the control is being used. For list and selection boxes, the application is not so much interested in the item list, since this list is only a hint for the user. The application is interested in the selection that the user creates. For a list box, the selection is defined by the index of the selected item. For a selection box, the selection is defined by the set of indices of selected items. For combo boxes, the relevant state is not a selection, but the string that was entered.

For all three box controls, the item list must be built up somehow. For this purpose, module *Dialog* defines suitable interactor types: *Dialog.List* for list boxes, *Dialog.Selection* for selection boxes, and *Dialog.Combo* for combo boxes. These types are defined the following way:

```
List = RECORD
   index: INTEGER;   (* index of currently selected item *)
   len-: INTEGER;   (* number of list elements *)
   PROCEDURE (VAR l: List) SetLen (len: INTEGER), NEW;
   PROCEDURE (VAR l: List) SetItem (index: INTEGER; IN item: ARRAY OF CHAR), NEW;
   PROCEDURE (VAR l: List) GetItem (index: INTEGER; OUT item: String), NEW;
   PROCEDURE (VAR l: List) SetResources (IN key: ARRAY OF CHAR), NEW
END;

Selection = RECORD
   len-: INTEGER;   (* number of selection elements *)
   PROCEDURE (VAR s: Selection) SetLen (len: INTEGER), NEW;
   PROCEDURE (VAR s: Selection) SetItem (index: INTEGER; IN item: ARRAY OF CHAR), NEW;
   PROCEDURE (VAR s: Selection) GetItem (index: INTEGER; OUT item: String), NEW;
   PROCEDURE (VAR s: Selection) SetResources (IN key: ARRAY OF CHAR), NEW;
   PROCEDURE (VAR s: Selection) Incl (from, to: INTEGER), NEW;
                                          (* select range [from..to] *)
   PROCEDURE (VAR s: Selection) Excl (from, to: INTEGER), NEW;
                                          (* deselect range [from..to] *)
   PROCEDURE (VAR s: Selection) In (index: INTEGER): BOOLEAN, NEW
                                          (* test whether index-th item is selected *)
END;

Combo = RECORD
   item: String;   (* currently entered or selected string *)
   len-: INTEGER;   (* number of combo elements *)
   PROCEDURE (VAR c: Combo) SetLen (len: INTEGER), NEW;
   PROCEDURE (VAR c: Combo) SetItem (index: INTEGER; IN item: ARRAY OF CHAR), NEW;
   PROCEDURE (VAR c: Combo) GetItem (index: INTEGER; OUT item: String), NEW;
   PROCEDURE (VAR c: Combo) SetResources (IN key: ARRAY OF CHAR), NEW
END;
```

Listing 4-35. Definitions of List, Selection and Combo

Before a variable of one of these types can be used, it must be initialized by first defining the individual items. During use, items can be changed. For example, the following code fragment builds up a list:

```
list.SetLen(5);   (* define length of list *)
list.SetItem(0, "Daffy Duck");
list.SetItem(1, "Wile E. Coyote");
list.SetItem(2, "Scrooge Mc Duck");
list.SetItem(3, "Huey Lewis");
```

```
    list.SetItem(4, "Thomas Dewey");
    Dialog.UpdateList(list);    (* must be called after any change(s) to the item list *)
```

Listing 4-36. Setting up a list explicitly

When used with list-structured controls (list, selection, or combo boxes), *Dialog.Update* only updates the selection or text entry state of these controls, but not the list structure. If the list structure, i.e., the elements of the control's list, is changed, then the procedure *Dialog.UpdateList* must be called instead of *Dialog.Update*.

For fixed item lists, the individual strings should be stored in resources. This is simplified by the *SetResources* procedures. They look up the strings in a resource file. For example, the above statements can be replaced completely by the statement

```
    list.SetResources("#Obx:list")
```

which will read the Obx/Rsrc/Strings file and look for strings with keys of the kind "list[index]", e.g., for

list[0]	Daffy Duck
list[1]	Wile E. Coyote
list[2]	Scrooge Mc Duck
list[3]	Huey Lewis
list[4]	Thomas Dewey

Table 4-37. Resources for a list

The indices must start from 0 and be consecutive (no holes allowed).
Procedure *SetLen* is optional. Its use is recommended wherever the number of list items is known in advance. If this is not the case, it may be omitted. The list will become as large as the largest index requires.
SetLen is necessary if an existing item list should be shortened, or if it should be cleared completely.

4.8 Input validation

Validity checks are an important, and for the uninitiated sometimes a surprising, aspect of non-modal user interfaces. Among other things, non-modality also means that the user must not be forced into a mode depending on where the caret currently is and whether or not the entered data is currently valid. In particular, a user must not be forced to correctly enter some data into a field before permitting him or her to do something else.
This basically leaves two strategies for checking the validity of entered data: early or late. Early checks are performed whenever the user has manipulated a control. Late checks are performed when the user has completed input and wants to perform some action, e.g., entering the new data into a database.
Late checks are most suitable for checking global invariants, e.g., whether all necessary fields in the input mask contain some input. Early checks are most suitable for local, control-specific invariants, e.g., the correct syntax of an entered string.

We give examples of both early and late checks for our module *ObxPhoneUI*. Let us assume that a phone number always has the following form: 310-555-1212. For a late check, we extend the *Lookup* procedure (boldface text) and add a few auxiliary procedures. Note the use of the "$" operator, which makes sure the *LEN* function returns the length of the string, not of the array containing the string.

```
    PROCEDURE Valid (IN s: ARRAY OF CHAR): BOOLEAN;
```

```
   PROCEDURE Digits (IN s: ARRAY OF CHAR; from, to: INTEGER): BOOLEAN;
   BEGIN   (* check whether range [from..to] in s consists of digits only *)
      WHILE (from <= to) & (s[from] >= "0") & (s[from] <= "9") DO INC(from) END;
      RETURN from > to   (* no non-digit found in checked range *)
   END Digits;

BEGIN   (* check syntax of phone number *)
   RETURN (LEN(s$) = 12) & Digits(s, 0, 2) & (s[3] = "-") & Digits(s, 4, 6) &
                  (s[7] = "-") & Digits(s, 8, 11)
END Valid;

PROCEDURE ShowErrorMessage;
BEGIN
   phone.name := "illegal syntax of number"
END ShowErrorMessage;

PROCEDURE Lookup*;
BEGIN
   IF phone.lookupByName THEN
      ObxPhoneDB.LookupByName(phone.name, phone.number);
      IF phone.number = "" THEN phone.number := "not found" END
   ELSE
      IF Valid(phone.number) THEN
         ObxPhoneDB.LookupByNumber(phone.number, phone.name);
         IF phone.name = "" THEN phone.name := "not found" END
      ELSE
         ShowErrorMessage
      END
   END;
   Dialog.Update(phone)
END Lookup;
```

Listing 4-38. Late check for input validation

A more rude reminder would be to display an error message. If the BlackBox Component Builder's log window is open, the message is written to the log and the window is brought to the top if necessary. If no log is used, a dialog box is displayed. This behavior can be achieved by replacing the statement in procedure *ShowErrorMessage* by the following statement:

```
Dialog.ShowMsg("Please correct the phone number")
```

Now we look at an early checked alternative to the above solution to input checking. It uses a notifier to check after each character typed into the phone number field whether the number is legal so far. In contrast to the late check, it must be able to deal with partially entered phone numbers. Whenever an illegal suffix is detected, the string is simply clipped to the legal prefix.

```
PROCEDURE Correct (VAR s: ObxPhoneDB.String);

   PROCEDURE CheckMinus (VAR s: ObxPhoneDB.String; at: INTEGER);
   BEGIN
      IF s[at] # "-" THEN s[at] := 0X END   (* clip string *)
   END CheckMinus;
```

```
        PROCEDURE CheckDigits (VAR s: ARRAY OF CHAR; from, to: INTEGER);
        BEGIN
            WHILE from <= to DO
                IF (s[from] < "0") OR (s[from] > "9") THEN
                    s[from] := 0X; from := to    (* clip string and terminate loop *)
                END;
                INC(from)
            END
        END CheckDigits;

    BEGIN    (* clip string to a legal prefix if necessary *)
        CheckDigits(s, 0, 2);
        CheckMinus(s, 3);
        CheckDigits(s, 4, 6);
        CheckMinus(s, 7);
        CheckDigits(s, 8, 11)
    END Correct;

    PROCEDURE NumberNotifier (op, from, to: INTEGER);
    BEGIN
        Correct(phone.number)
        (* Dialog.Update is not called, because it will be called afterwards by the notifying control *)
    END NumberNotifier;
```

Listing 4-39. Early check for input validation

Note that a focused view, e.g. a control, has no means to prevent the user from focusing another view. A view may merely change the way that it displays itself, its contents, or its marks (selection, caret). For this purpose, a view receives a *Controllers.MarkMsg* when the focus changes.

4.9 Accessing controls explicitly

In most circumstances, guards and notifiers allow sufficient control over a control's specific look & feel. However, sometimes you may want to exercise direct control over a control in a form. How to do this is described in this section. For example, assume that you want to write your own special alignment command, and add it to the commands of the *Layout* menu. To do this, you need to get access to the form view in the window that is currently being edited. The form view is a container that contains the controls that you want to manipulate. The function *FormControllers.Focus* delivers a handle on the currently focused form editor. This leads to the typical code pattern for accessing a form (Listing 4-40):

```
        VAR c: FormControllers.Controller;
    BEGIN
        c := FormControllers.Focus();
        IF c # NIL THEN
            ...
```

Listing 4-40. Accessing a form's controller

A form contains controls and possibly some other types of views. The views can be edited if the enclosing form view is in layout mode. Typically, some views are selected first, and then a command is executed that operates on the selected views. The following example shifts every selected view to the right by one centimeter

(Listing 4-41):

```
MODULE ObxControlShifter;

    IMPORT Ports, Views, FormModels, FormControllers;

    PROCEDURE Shift*;
        VAR c: FormControllers.Controller; sel: FormControllers.List;
    BEGIN
        c := FormControllers.Focus();
        IF (c # NIL) & c.HasSelection() THEN
            sel := c.GetSelection();    (* generates a list with references to the selected views *)
            WHILE sel # NIL DO
                c.form.Move(sel.view, 10 * Ports.mm, 0);    (* move to the right *)
                sel := sel.next
            END
        END
    END Shift;

END ObxControlShifter.
```

Listing 4-41. Shift all selected views to the right

This code works on selected views (whether they are controls or not). Sometimes you may want to manipulate views that are not selected. In this case, you need a form reader (*FormModels.Reader*) to iterate over the views in the currently focused form. The following code pattern shows how a command can iterate over the views of a form (Listing 4-42):

```
        VAR c: FormControllers.Controller; rd: FormModels.Reader;
    BEGIN
        c := FormControllers.Focus();
        IF c # NIL THEN
            rd := c.form.NewReader(NIL);
            rd.ReadView(v);    (* read first view *)
            WHILE v # NIL DO
                ...
                rd.ReadView(v)    (* read next view *)
            END;
            ...
```

Listing 4-42. Iterating over the selected views in a form

Controls are special views (of type *Controls.Control*). Controls can be linked to global variables. The following example shows how the labels of all controls in a form can be listed (Listing 4-43):

```
MODULE ObxLabelLister;

    IMPORT Views, Controls, FormModels, FormControllers, StdLog;

    PROCEDURE List*;
        VAR c: FormControllers.Controller; rd: FormModels.Reader; v: Views.View;
    BEGIN
```

```
        c := FormControllers.Focus();
        IF c # NIL THEN
            rd := c.form.NewReader(NIL);
            rd.ReadView(v);    (* read first view *)
            WHILE v # NIL DO
                IF v IS Controls.Control THEN
                    StdLog.String(v(Controls.Control).label); StdLog.Ln
                END;
                rd.ReadView(v)    (* read next view *)
            END
        END
    END List;

END ObxLabelLister.
```

Listing 4-43. Listing the labels of all controls in a form

With the same kind of iteration, any other type of view could be found in forms as well, not only controls. For example, consider that you have a form with a "Clear" command button that causes a plotter view in the same form to be cleared. The command button is a standard control, the plotter view is your special view type. The command associated with the command button first needs to search for its plotter view, in the same form where the button is placed. The code pattern shown below demonstrates how such a command can be implemented (Listing 4-44):

```
        VAR c: FormControllers.Controller; rd: FormModels.Reader; v: Views.View;
    BEGIN
        c := FormControllers.Focus();
        IF c # NIL THEN
            rd := c.form.NewReader(NIL);
            rd.ReadView(v);    (* read first view *)
            WHILE v # NIL DO
                IF v IS MyPlotterView THEN
                    (* clear v(MyPlotterView) *)
                END;
                rd.ReadView(v)    (* read next view *)
            END
        END
```

Listing 4-44. Finding a particular view in a form

There is one potential problem with the above code pattern. Let us assume that we have a dialog box containing a command button with *ObxLabelLister.List* as associated command. Beneath the dialog box, we have a window with a focused form layout. Now, if you click on the dialog box' command button, which labels are listed? The ones in the form of the dialog box itself, or the ones in the focused form layout underneath? In other words: does *FormControllers.Focus* yield the form that contains the button you clicked on, or the form focused for editing? Depending on what the button's command does, both versions could make sense. For a form editing command like *ObxControlShifter.Shift*, the focused layout editor should be returned. This is the top-most document window, but it is overlaid by the dialog box window. In contrast, if you want to find your plotter view, the form of the dialog box should be returned instead. In this case, you want to search the direct "neighborhood" of the button.

One solution for the plotter view search would be to start searching in the context of the button that is currently being pressed. This can be done using the following code pattern (Listing 4-45):

```
    VAR button: Controls.Control; m: Models.Model; rd: FormModels.Reader; v: Views.View;
  BEGIN
    button := Controls.par;    (* during the button click, this variable contains a reference to the button *)
    m := button.context.ThisModel();    (* get the model of the container view that contains the button *)
    IF m IS FormModels.Model THEN    (* the container is a form *)
      rd := m(FormModels.Model).NewReader(NIL);
      rd.ReadView(v);    (* read first view *)
      WHILE v # NIL DO
        IF v IS MyPlotterView THEN
          (* clear v(MyPlotterView) *)
        END;
        rd.ReadView(v)    (* read next view *)
      END
    END
```

Listing 4-45. Finding a particular view in the same form as a button

This code assumes that the command is executed from a command button. It doesn't work if placed in a menu. In order to better decouple the user interface and the application logic, this assumption should be eliminated. BlackBox solves the problem by giving the programmer explicit control over the behavior of *FormControllers.Focus*. The trick is that BlackBox supports two kind of windows for dialog boxes: *tool windows* and *auxiliary windows,* in addition to the normal document windows. If the command button is in a tool window dialog box, then *FormControllers.Focus* yields the form in the top-most *document* window, or *NIL* if there is no such document window or if the focus of the top-most document window is not a form view. If the command button is in an auxiliary window, then *FormControllers.Focus* yields the dialog box' form whose command button you have clicked. The following paragraphs further explain the differences between these two kinds of windows.

Dialog boxes in tool windows are not self-contained, they provide the parameters and the control panel for an operation on a document underneath. A typical example is the "Find / Replace" dialog box that operates on a text document underneath. The command buttons in a tool dialog box invoke a command, and this command fetches the view on which it operates. In the "Find / Replace" example, this is the focused text view. Under Windows, tool windows always lie above the topmost document window; i.e., they can never be overlapped by document windows or auxiliary windows, only by other tool windows. Unlike other windows, tool windows cannot be resized or iconized, but can me moved outside of the application window. On Mac OS, tool windows look and behave like document windows, except that they have no scroll bars and cannot be resized or zoomed.

An auxiliary window on the other hand is self-contained. It contains, and operates on, its own data. At least, it knows where to find the data on which to operate (e.g., in a database). The "Phone Database" dialog box is a typical auxiliary window, like most data-entry forms. Auxiliary windows can be manipulated like normal document windows; in particular, the may be overlapped by other document or auxiliary windows.

Both kinds of dialog boxes are stored as documents in their appropriate *RSRC* directories. But to open them, module *StdCmds* provides two separate commands: *OpenToolDialog* and *OpenAuxDialog*. Both of them take a portable path name of the resource document as first parameter, and the title of the dialog box window as a second parameter. For example, the following menu commands may be used:

```
  "Find / Replace..."    ""    "StdCmds.OpenToolDialog('Text/Rsrc/Cmds', 'Find / Replace')"    ""
```

```
"Phone Database..."    ""    "StdCmds.OpenAuxDialog('Obx/Rsrc/PhoneUI', 'Phone Database')"    ""
```

The function *FormControllers.Focus* returns different results, depending on whether the form is in a tool or in an auxiliary window.

For more examples on how forms can be manipulated, see also module *FormCmds*. It is available in source form, like the rest of the *Form* subsystem.

4.10 Summary

In this chapter, we have discussed the important aspects of how a graphical user interface and its supporting code is implemented in the BlackBox Component Builder. There are three parts of an application: application logic, user interface logic, and user interface resources, i.e., documents. A clean separation of these parts makes a program easier to understand, maintain, and extend.

For more examples of how the form system and controls can be used, see the on-line examples *ObxAddress0*, *ObxAddress1*, *ObxAddress2*, *ObxOrders*, *ObxControls*, *ObxDialog,* and *ObxUnitConv*. For advanced programmers, the sources of the complete *Form* subsystem may be interesting.

For more information on how to use the BlackBox development environment, consult the documentation of the following modules:

DevCompiler, DevDebug, DevBrowser, DevInspector, DevReferences,
DevMarkers, DevCmds, StdCmds, StdMenuTool, StdLog, FormCmds, TextCmds

To obtain more information on a module, select a module name and then invoke *Info->Documentation*.
In order to provide a straight-forward description, we only used the most important commands in this book. However, there are other useful commands in the menus *Info*, *Dev*, *Controls*, *Tools*, and *Layout*. For example, there are several "wizards" for the creation of new source code skeletons. Consult the files *System/Docu/User-Man*, *Text/Docu/User-Man*, *Form/Docu/User-Man*, and *Dev/Docu/User-Man* for a comprehensive user manual on how to use the framework, the editor (*Text* subsystem), the visual designer (*Form* subsystem), and the development tools (*Dev* subsystem).

5 Texts

Graphical user interfaces have introduced graphical elements into everyday computing. While this was a very positive step towards overcoming the old-style modal textual user interfaces, texts have by no means become obsolete. Even in reports containing many illustrations, texts serve as indispensible glue.

In typical database applications, no graphical contents is manipulated. The data entered and retrieved are mostly textual, and reports to be printed usually are texts arranged in a tabular fashion.

Because of the importance of texts, this chapter demonstrates how the BlackBox Component Builder's text abstraction can be used in various ways. In several examples, we will reuse the simple database module introduced in the previous chapter.

We will work "bottom-up" by providing examples first for writing new texts, then for reading existing text, and finally for modifying existing texts. When discussing these examples, we will meet text models/carriers and riders, mappers, and auxiliary abstractions such as fonts, attributes, rulers, and others.

5.1 Writing text

In the BlackBox Component Builder, texts use the Carrier-Rider-Mapper design pattern (see Chapter 3). They provide text models as carriers for texts, and text writers and text readers as riders on text models. Both types are defined in the Text subsystem's core module *TextModels*. A text writer maintains a current position and current attributes, which will be used when writing the next element into the text. The full definition of a writer looks as follows:

```
Writer = POINTER TO ABSTRACT RECORD
    attr-: TextModels.Attributes;
    (wr: Writer) Base (): TextModels.Model, NEW, ABSTRACT;
    (wr: Writer) Pos (): INTEGER, NEW, ABSTRACT;
    (wr: Writer) SetPos (pos: INTEGER), NEW, ABSTRACT;
    (wr: Writer) SetAttr (attr: TextModels.Attributes), NEW;
    (wr: Writer) WriteChar (ch: CHAR), NEW, ABSTRACT;
    (wr: Writer) WriteView (view: Views.View; w, h: INTEGER), NEW, ABSTRACT
END;
```

Listing 5-1. Definition of TextModels.Writer

The most important procedure of a writer is *WriteChar*, it allows to write a new character at the current position. If the character position lies within an existing text (i.e., in 0 .. text.Length() - 1), the character at this text position is inserted. (This is a difference to the BlackBox files abstraction, where old data would be overwritten by the writer in this case. With texts, writers *always* insert and never overwrite.) If the writer's position is at the end of the text (i.e., at text.Length()), the character is appended to the text and thus makes the text one element longer. After *WriteChar*, the writer's text position is increased by one. The current position can be inquired by calling the writer's *Pos* function, and it can be modified by calling the *SetPos* procedure.

Since text models are capable of containing views as well as normal characters, there exists a procedure *WriteView* which allows to write a view, given an appropriate size (*w* for the width, *h* for the height).

Like most geometrical distances in BlackBox, a view's width and height are specified in so-called *universal units* of 1/36000 millimeters. This value was chosen to eliminate rounding errors for many common screen and printer resolutions. For example, an inch, a 1/300-th of an inch, a millimeter, and a desktop publishing point can be represented in units without rounding errors. Where not specified otherwise, distances can be assumed to be represented in universal units.

If no specific size for the view is desired, passing the value *Views.undefined* for the width and/or height allows the text to choose a suitable default size for the written view.

How is a writer created? Like all riders, a writer is created by a factory function of its carrier. For this purpose, a text model provides the procedure *NewWriter*, which creates a writer and connects it to the text. A writer can return its text through the *Base* function. A newly created writer is positioned at the end of the text.

Text elements don't have character codes only, they also have attributes. A writer's *attr* field contains the attributes to be used when the next character or view is written. The current attributes can be changed by calling the writer's *SetAttr* procedure.

In terms of attributes, the text subsystem of the BlackBox Component Builder supports **colors** and vertical offsets. Colors are encoded as integers (*Ports.Color*). Module *Ports* predefines the constants *black*, *grey6*, *grey12*, *grey25*, *grey50*, *grey75*, *white*, *red*, *green*, and *blue*. "grey6" means a mixture of 6% white and 94% black, which is almost black. The other grey values are increasingly lighter. Vertical offsets are measured in universal units.

Besides colors and vertical offset, a text element also has a font attribute. A font is a collection of glyphs. A glyph is a visual rendering of a character code. Different fonts may contain different glyphs for the same characters. For example, Helvetica looks different from Times and Frutiger. A font may not provide glyphs for all character codes that can be represented in Component Pascal. This is not surprising, since the *CHAR* datatype of Component Pascal conforms to the Unicode standard. Unicode is a 16-bit standard, and thus can represent over 65000 different characters. The first 256 characters are the Latin-1 characters. The first 128 characters of Latin-1 are the venerable ASCII characers.

If a font doesn't contain a particular glyph, BlackBox's font framework either returns a glyph from some other font for the same character, or it yields a generic "missing glyph" symbol; e.g., a small empty rectangle.

Font objects are defined in module *Fonts.Font*. It won't be necessary to take a closer look at this module here; it is sufficient to interpret a pointer of type *Fonts.Font* as the identification of a font with its particular typeface, style, weight, and size:

Typeface

This is the font family to which a font belongs; it defines a common "look and feel" for the whole family. Examples of typefaces are Helvetica, Times, or Courier. Typeface names are represented as character arrays of type *Fonts.Typeface*.

Style

A font may optionally be *italicized*, underlined, ~~struck out~~ or ~~*a combination thereof*~~. Styles are represented as sets. Currently, the set elements *Fonts.italic*, *Fonts.underline* and *Fonts.strikeout* are defined.

Weight

A font might exist in several weights, e.g., ranging from light to normal to **bold** to black to **ultra black**. Usually, a font is either normal or bold. Weights are represented as integers. The values *Fonts.normal* and *Fonts.bold* are predefined.

Size

A font may be rendered in different sizes, taking into account the resolution of the used output device. Often, sizes are given in points. A point is 1/72-th of an inch. In BlackBox, font sizes are measured in universal units, which are chosen such that points (*Ports.point*) and other typographical units can be represented without round-off errors. Typical font sizes are *10 * Ports.point* or *12 * Ports.point*.

In principle, typeface, style, weight and size are independent of each other. This means that a given typeface can be combined with arbitrary styles, weights and sizes. However, from a typographical point of view, some combinations are more desirable than others, and may be optimized by the font designer and the underlying operating system's font machinery.

The text attributes font, color and vertical offset are packaged into objects of type *TextModels.Attributes*. Slightly simplified, this type looks as follows:

```
Attributes = POINTER TO RECORD (Stores.Store)
    color-: Ports.Color;
    font-: Fonts.Font;
    offset-: INTEGER
END;
```

Listing 5-2. Simplified definition of TextModels.Attributes

This is the type of the writer's *attr* field. It is an extension of *Stores.Store*. This means that such an object can be stored persistently in a file. Other examples of stores are *TextModels.Model* and *TextViews.View*, but not *TextModels.Writer*. A more detailed description of stores is given in Part III.

We have now seen the capabilities of a text writer, including the text attributes that it supports. The typical use of text writers is shown in the code pattern below. A code pattern is an excerpt of a procedure which shows one or a few important aspects of the objects currently discussed. Such a code fragment constitutes a recipe that should be known by a programmer "fluent" in this topic.

```
    VAR t: TextModels.Model; wr: TextModels.Writer; ch: CHAR;
BEGIN
    t := TextModels.dir.New();     (* allocate new empty text model *)
    wr := t.NewWriter(NIL);
    ... produce ch ...
    WHILE condition DO
        wr.WriteChar(ch);
        ... produce ch ...
    END;
```

Listing 5-3. Code pattern for TextModels.Writer

The code pattern is not very interesting since it doesn't show how the generated text can be displayed. A text object only represents a text with its attributes. It doesn't know about how to draw this text, not even how to do text setting; i.e., how to break lines and pages such that they fit a given rectangle (view, paper size, etc.). When it is desired to make a text visible, it is necessary to provide a text view for the text object. Data objects that may be displayed in several views are generally called models. The separation of model and view is an important design pattern that was discussed in Chapter 2.

Module *TextModels* only exports an abstract record type for text models, no concrete implementation. Instead, it exports a directory object (*TextModels.dir*) which provides the necessary factory function (*TextModels.dir.New*). This indirection pattern for object creation was motivated in Chapter 2.

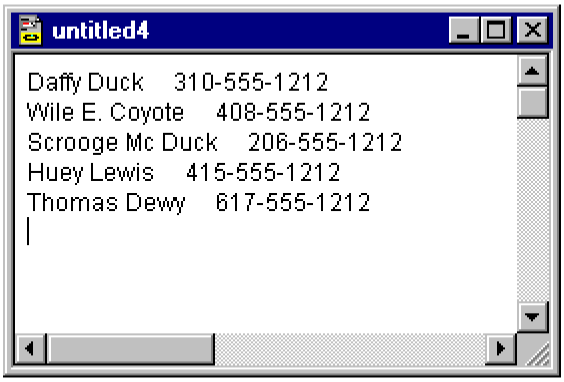
Working directly with text writers is inconvenient, they don't even support the writing of strings. For this reason, the following examples will use *formatters* when writing text. A formatter is a mapper (see Chapter 3) that contains a text writer and performs the mapping of higher-level symbols, such as the Component Pascal symbols, to a stream of characters that can be fed to the writer. In the next example, we we will use a formatter to write a text; a very simple "report". The text consists of the phone database contents, one line per entry. This first version of a phone database report is implemented in module *ObxPDBRep0*:

```
MODULE ObxPDBRep0;

   IMPORT Views, TextModels, TextMappers, TextViews, ObxPhoneDB;

   PROCEDURE GenReport*;
      VAR t: TextModels.Model; f: TextMappers.Formatter; v: TextViews.View;
         i: INTEGER; name, number: ObxPhoneDB.String;
   BEGIN
      t := TextModels.dir.New();    (* create empty text carrier *)
      f.ConnectTo(t);               (* connect a formatter to the text *)
      i := 0;
      ObxPhoneDB.LookupByIndex(i, name, number);
      WHILE name # "" DO
         f.WriteString(name);       (* first string *)
         f.WriteTab;                (* tab character *)
         f.WriteString(number);     (* second string *)
         f.WriteLn;                 (* carriage return *)
         INC(i);
         ObxPhoneDB.LookupByIndex(i, name, number)
      END;
      v := TextViews.dir.New(t);    (* create a text view for the text generated above *)
      Views.OpenView(v)             (* open the text view in its own window *)
   END GenReport;

END ObxPDBRep0.
```

Listing 5-4. Writing the phone database using a formatter

Note that name and number are separated by a tab character (09X). Since no tab stops are defined for this text, a tab will be treated as a wide fixed-width space (whose width is a multiple of the normal space character glyph). We will later see how tab stops can be defined.

Execution of the *ObxPDBRep0.GenReport* command results in the following window:



```
Daffy Duck    310-555-1212
Wile E. Coyote    408-555-1212
Scrooge Mc Duck    206-555-1212
Huey Lewis    415-555-1212
Thomas Dewy    617-555-1212
```

Figure 5-5. Text editor window containing the phone database

The text in the newly opened window is a fully editable text view. It can be edited; stored as a BlackBox document; printed; and so on. If you look at the source code, you can recognize the following code pattern:

```
    VAR t: TextModels.Model; f: TextMappers.Formatter; v: Views.View;
  BEGIN
    t := TextModels.dir.New();
    f.ConnectTo(t);
    ... use formatter procedures to construct the text ...
    v := TextViews.dir.New(t);
    Views.OpenView(v)
```

Listing 5-6. Code pattern for TextMappers.Formatter and for open a text view

This code pattern occurs in many BlackBox commands, e.g., in *DevDebug.ShowLoadedModules*. It is useful whenever you need to write tabular reports of varying length, possibly spanning several printed pages.
It also clearly shows that texts embody several design patterns simultaneously: a text object is a model which can be observed by a view (Observer pattern), it is a carrier (Carrier-Rider-Mapper pattern), and it is a container (Composite pattern).

An interesting aspect of the above code pattern is that the text carrier *t* is created first, then its contents is constructed using a formatter, and only then is a view *v* on the text carrier created. Finally, the view is opened in its own document window.
In BlackBox it is important that a model can be manipulated before there exists any view for it. In fact, it is far more efficient to create a text before a view on it is opened, because screen updates and some internal housekeeping of the undo mechanism are avoided this way, which can cause a dramatic speed difference.

BlackBox text models represent sequences of text elements. A text element is either a Latin-1 character (*SHORTCHAR*), a Unicode character (a *CHAR > 0FFX*) or a view (an extension of type *Views.View*). Text stretches may be attributed with font information, color and vertical offset. Since a text model may contain views, it is a *container*. In the following examples, we will see how these text facilities can be used when creating new texts. Our phone book database is used as source of the material that we want to put into texts, applying different textual representations in the various examples.

The following example demonstrates how a text can be generated out of our example database. In this text, names are written in green color. The differences to module *ObxPDBRep0* are marked with underlined text.

```
MODULE ObxPDBRep1;

  IMPORT Ports, Views, TextModels, TextMappers, TextViews, ObxPhoneDB;

  PROCEDURE GenReport*;
    VAR t: TextModels.Model; f: TextMappers.Formatter; v: TextViews.View;
      i: INTEGER; name, number: ObxPhoneDB.String;
      default, green: TextModels.Attributes;
  BEGIN
    t := TextModels.dir.New();    (* create empty text carrier *)
    f.ConnectTo(t);               (* connect a formatter to the text *)
    default := f.rider.attr;    (* save old text attributes for later use *)
    green := TextModels.NewColor(default, Ports.green);    (* use green color *)
    i := 0;
    ObxPhoneDB.LookupByIndex(i, name, number);
    WHILE name # "" DO
      f.rider.SetAttr(green);    (* change current attributes of formatter's rider *)
      f.WriteString(name);        (* first string *)
      f.rider.SetAttr(default);    (* change current attributes of formatter's rider *)
```

```
            f.WriteTab;                  (* tab character *)
            f.WriteString(number);   (* second string *)
            f.WriteLn;                      (* carriage return *)
            INC(i);
            ObxPhoneDB.LookupByIndex(i, name, number)
        END;
        v := TextViews.dir.New(t);    (* create a text view for the text generated above *)
        Views.OpenView(v)             (* open the text view in its own window *)
    END GenReport;

END ObxPDBRep1.
```

Listing 5-7. Writing the phone database using green color

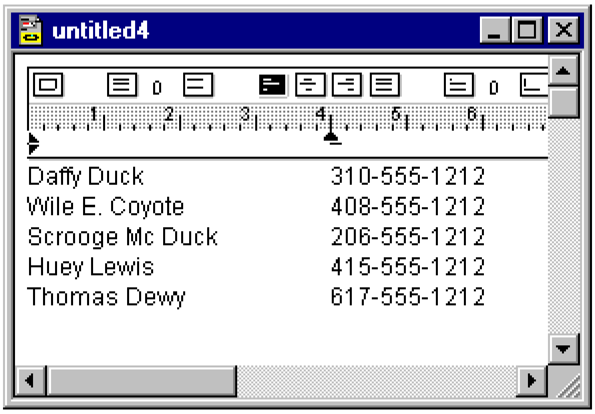
Note that a formatter's rider contains a set of current attributes (*TextModels.Writer.attr*). This value includes the current color and is read-only; it can be set with the writer's *SetAttr* procedure.

Our first text examples, *ObxPDBRep0* and *ObxPDBRep1*, don't produce nice-looking output, because the phone numbers are not aligned below each other in a tabular fashion.
ObxPDBRep1 remedies this defect by inserting a ruler at the beginning of the text. This ruler defines a tab stop, which causes all phone numbers - separated from their phone names by tabs - to line up nicely. The differences to module *ObxPDBRep0* are marked with underlined text.

```
MODULE ObxPDBRep2;

    IMPORT Ports, Views, TextModels, TextMappers, TextViews, TextRulers, ObxPhoneDB;

    PROCEDURE WriteRuler (VAR f: TextMappers.Formatter);
        CONST cm = 10 * Ports.mm;    (* universal units *)
        VAR ruler: TextRulers.Ruler;
    BEGIN
        ruler := TextRulers.dir.New(NIL);
        TextRulers.AddTab(ruler, 4 * cm);    (* define a tab stop, 4 cm from the left margin *)
        TextRulers.SetRight(ruler, 12 * cm);    (* set right margin *)
        f.WriteView(ruler)                          (* a ruler is a view, thus can be written to the text *)
    END WriteRuler;

    PROCEDURE GenReport*;
        VAR t: TextModels.Model; f: TextMappers.Formatter; v: TextViews.View;
            i: INTEGER; name, number: ObxPhoneDB.String;
    BEGIN
        t := TextModels.dir.New();    (* create empty text carrier *)
        f.ConnectTo(t);                   (* connect a formatter to the text *)
        WriteRuler(f);
        i := 0;
        ObxPhoneDB.LookupByIndex(i, name, number);
        WHILE name # "" DO
            f.WriteString(name);        (* first string *)
            f.WriteTab;                  (* tab character *)
            f.WriteString(number);   (* second string *)
            f.WriteLn;                      (* carriage return *)
            INC(i);
            ObxPhoneDB.LookupByIndex(i, name, number)
        END;
        v := TextViews.dir.New(t);     (* create a text view for the text generated above *)
        Views.OpenView(v)              (* open the text view in its own window *)
    END GenReport;

END ObxPDBRep2.
```

Listing 5-8. Writing the phone database using a formatter and a ruler

Execution of the *ObxPDBRep2.GenReport* command results in a window with the improved presentation of the phone database. Executing Text->Show Marks causes the ruler to be displayed:



Daffy Duck	310-555-1212
Wile E. Coyote	408-555-1212
Scrooge Mc Duck	206-555-1212
Huey Lewis	415-555-1212
Thomas Dewy	617-555-1212

Figure 5-9. Text editor window containing the tabular phone database

A ruler is a view. A ruler on its own doesn't make sense. Like all controls, a ruler makes only sense in a container. Unlike many other controls, a ruler is an example of a control that can only function properly in a particular container; in this case the container must be a text model. You may copy a ruler into a form or other container, but it will just be passive and have no useful effect there. If it is embedded in a text model that is displayed in a text view, a ruler can influence the way in which the subsequent text is set. For example, a ruler can define tab stops, margins, and adjustment modes.

The attributes of a ruler are valid in the ruler's *scope*. The scope starts with the ruler itself and ends right before the next ruler, or at the end of the text if there follows no ruler anymore. A ruler introduces a new paragraph, possibly with its own distinct set of paragraph attributes. Inserting a paragraph character (0EX) instead of a ruler also starts a new paragraph, which inherits all paragraph attributes from the preceding ruler. Paragraph characters are the normal way to start a new paragraph; rulers are only used if particular paragraph attributes should be enforced. Note that a carriage return (0DX) doesn't start a new paragraph.

A text view contains an invisible default ruler that defines the text setting of the text before the first ruler in the text, or of all text if no ruler has been written to the text at all. Interactively, this default ruler, and default attributes, can be set with the menu commands *Text->Make Default Ruler* and *Text->Make Default Attributes*.

A ruler has many attributes that can be set. Unlike character attributes, which can be applied to arbitrary text stretches, these attributes apply to the ruler or the whole paragraph(s) in its scope.
We will discuss these attributes so that you know about ruler functionality when you need it. We won't discuss rulers in more detail though, since you can learn about the necessary details on demand if and when you need the more advanced ruler capabilities (see on-line documentation of module *TextRulers*).

Styles

A ruler's attributes are not directly stored in the ruler view itself. Instead, they are bundled into a separate object of type *TextRulers.Attributes*, which in turn is contained in an object of type *TextRulers.Style*.

A style is the model of one or several ruler views. If several views share the same model, the user can change the attributes of all the rulers simultaneously. For example, changing the left margin will affect all rulers bound to the same style. However, models cannot be shared across documents, which means that all rulers bound to the same style must be embedded in the same document.

TextRulers.Attributes are persistent collections of ruler attributes, much in the same way as *TextModels.Attributes* are persistent collections of character attributes.

```
Attributes = POINTER TO EXTENSIBLE RECORD (Stores.Store)
    first-, left-, right-, lead-, asc-, dsc-, grid-: INTEGER;
    opts-: SET;
    tabs-: TextRulers.TabArray
END;
```

Listing 5-10. Simplified definition of TextRulers.Attributes

In the following paragraphs, we discuss the individual attributes. For these attributes, there exist auxiliary procedures in module *TextRulers* which make it simple to set up a text ruler in the desired way: just create a new ruler with *TextRulers.dir.New(NIL)*, and then apply the auxiliary procedures that you need to obtain the correct attribute settings. An example was given above in the procedure *WriteRuler* of module *ObxPDBRep2*.

These auxiliary procedures greatly simplify setting up text rulers. For example, they hide that the attributes are actually not contained in the ruler itself, but rather in attributes objects which are used by style objects which are used by the rulers. Providing such simplified access to a complex feature is called a *facade* in the design patterns terminology [GHJV94]. Here the mechanism is particularly simple because the facade merely consists of simple procedures, there is not even a facade object that would have to be created and managed.

Tab stops

The *tabs* field of type *TextRulers.TabArray* contains information about tab stops. The number of tab stops is given in *tabs.len*, and must not be larger than *TextRulers.maxTabs* (which is currently 32). If the user enters more than *maxTabs* tabs, or more tabs than given in *len*, then the superfluous tabs will be treated as fixed size space characters. The fixed size used is an integer multiple of the width of a space in the used font, approximately 4 mm in total.

```
Tab = RECORD
    stop: INTEGER;    (* stop >= 0 *)
    type: SET    (* type IN {TextRulers.centerTab, TextRulers.rightTab, TextRulers.barTab} *)
END;

TabArray = RECORD
    len: INTEGER;    (* 0 <= len <= TextRulers.maxTabs *)
    tab: ARRAY TextRulers.maxTabs OF TextRulers.Tab
        (* tab[0 .. len-1] sorted in ascending order without duplicates *)
END
```

Listing 5-11. Definition of TextRulers.Tab and TabArray

In *tabs.tab.stop*, the tab's distance from the left border of the text view is specified (in universal units of course). The tab stops in the *tabs.tab* array must be sorted in ascending order, and there must be no duplicate values.

In addition to the stop position kept in field *stop*, a tab stop can be modified using a set of possible options kept in field type. In particular, these options allow to center text under a tab stop or to right-align text under a tab stop (the default is left alignment):

tab type	**tabs.tab[i].type**
left adjusted	{}
right adjusted	{TextRulers.rightTab}
centered	{TextRulers.centerTab}

For all of the above tab types, it can optionally be specified whether the tab itself is drawn as a vertical bar. Bar tabs are specified by including the value *TextRulers.barTab* in the tab type set. Since the bars occupy the whole line height, they are useful to to format simple tables.

When created with *TextRulers.dir.New(NIL)*, a ruler has no tabs.
The following auxiliary procedures are provided to modify a ruler:

```
PROCEDURE AddTab (r: Ruler; x: INTEGER)
PROCEDURE MakeRightTab (r: Ruler)
PROCEDURE MakeCenterTab (r: Ruler)
PROCEDURE MakeLineTab (r: Ruler)
```

AddTab adds a new left-adjusted normal tab. It can be changed into a right-adjusted or centered tab by calling *MakeRightTab* or *MakeCenterTab*. Independently of the adjustment mode, the tab can be turned into a bar tab by calling *MakeLineTab*.

Margins
Text is set within a left and a right margin. These margins can be specified as distances from the left border of the text view. The left margin must be smaller than the right margin.

```
left-, right-: INTEGER    (* (left >= 0)  &  (right >= left) *)
```

It is possible to signal to the text setter that it should ignore the right margin and use the text view for line breaking instead. In fact, this is the default behavior. It means that when the view size is changed, the text's line breaking will probably change also. Even if the right margins is ignored for text setting, it is still useful: a text view uses it as a hint for how wide the view should be when it is opened.
Line breaking at a fixed right margin is only needed in rare circumstances, for example when this particular paragraph should be narrower than the others. If you want line breaking to be controlled by the right margin, then you have to include *TextRulers.rightFixed* in the *opts* set (see further below).

Note that the *Tools->Document Size...* command allows to flexibly specify the width (and height) of a document's outermost view, the so-called *root view*. The width can be set to a fixed size, which is useful for page layout types of application.
Usually however, the width is defined by the page setup, which derives the view's width from the currently selected paper size, reduced by the user-specified print margins. This is adequate for letters, program sources, and similar kinds of documents.
As a third possibility, the root view's width can be bound to the window size (this is the case mentioned above, when the view uses the first ruler's right margin as a hint, whether the right margin is fixed or not). Whenever the user resizes the window, the root view is resized accordingly, such that it is kept always as large as the window allows. This is desirable in particular for on-line documentation, where the documentation should use as much screen space as granted by the user, adapting the text flow to the available width whenever the window is resized. An example for this style is the default formatting of Web pages.
Note that these features are independent of the root view's type. They work the same way for all resizable

views, such as text views, form views, and most other kinds of editor view.

When created with *TextRulers.dir.New(NIL)*, a ruler has a left margin of 0 and an implementation-specific non-fixed right margin > 0.
The following auxiliary procedures are provided to modify a ruler:

```
PROCEDURE SetLeft (r: Ruler; x: INTEGER)
PROCEDURE SetRight (r: Ruler; x: INTEGER)
PROCEDURE SetFixedRight (r: Ruler; x: INTEGER)
```

SetLeft and *SetRight* set the left or right margin. *SetFixedRight* sets the right margin and marks it as fixed.

First line indentation
A ruler defines the *first line indentation*, i.e., how much the first line after the ruler is indented from the left border of the text view. Note that it is possible to define a first line indentation that is smaller than the left margin (first < left). This is the only case where a text view draws text to the left of the specified left margin.

```
first-: INTEGER;    (* first >= 0 *)
```

First line indentation is applied to the first line of each paragraph in the ruler's scope.

When created with *TextRulers.dir.New(NIL)*, a ruler has no first line indentation (first = 0).
The following auxiliary procedure is provided to modify a ruler:

```
PROCEDURE SetFirst (r: Ruler; x: INTEGER)
```

Ascender and descender
When a character is drawn, its rendering depends on its font attributes, such as typeface (Helvetica or Times?), style (straight or slanted?), weight (normal or bold?) and size. When a whole string is drawn, all characters are placed on a common base line. The ascender is the largest extent to which any character of a line extends above the base line. Some characters extend below the base line, such as "g" and "y". The descender is the largest extent to which any character of a line extends below the base line (see Figure 5-12).
A text setter should calculate base line distances in a way that one line's descender doesn't overlap the next line's ascender, in order to avoid overlapping characters. In order to avoid degenerate cases, such as empty lines or lines that only contain tiny font sizes, a ruler can specify minimal values for ascender (asc) and descender (dsc).

```
asc-, dsc-: INTEGER;    (* (asc >= 0)  &  (dsc >= 0) *)
```

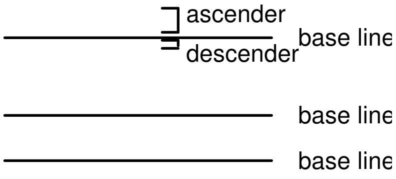


Figure 5-12. Ascender, descender, and base line

When created with *TextRulers.dir.New(NIL)*, a ruler uses the current default font's (*Fonts.dir.Default()*) ascender

and descender as initial values.
The following auxiliary procedures are provided to modify a ruler:

```
PROCEDURE SetAsc (r: Ruler; h: INTEGER)
PROCEDURE SetDsc (r: Ruler; h: INTEGER)
```

Paragraph lead

The paragraph lead defines the additional vertical space to be inserted between the first line of a paragraph and the previous paragraph's last line. Where the ruler's grid attribute (see below) is used to define a line grid, it is normally considered good style to choose a lead that is a multiple of the grid spacing, or sometimes half of the grid spacing. It allows to visually distinguish new paragraphs from mere carriage returns.

```
lead-: INTEGER;    (* lead >= 0 *)
```

When created with *TextRulers.dir.New(NIL)*, a ruler has a lead of 0.
The following auxiliary procedure is provided to modify a ruler:

```
PROCEDURE SetLead (r: Ruler; h: INTEGER)
```

Line grid

If *grid* is zero, then lines are packed as closely as possible, so that there remains no free room between one line's descender and the next line's ascender. Different lines may have different ascenders and descenders, depending on the text and fonts that they contain. For this reason, distances between base lines need not be regular. For typographical reasons, irregular line spacing is often undesirable.
To ensure a more regular spacing of grid lines, it is possible to switch on a line grid, by setting *grid* to a value larger than zero. This value defines a vertical grid, onto which all base lines are forced. If a line is too high to fit on the next grid line, it is placed one or more grid distances further below (see Figure 5-14).

```
grid-: INTEGER;    (* grid >= 0 *)
```

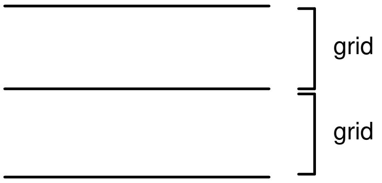


Figure 5-13. Line grid

When created with *TextRulers.dir.New(NIL)*, a ruler has a line grid of 1.
The following auxiliary procedure is provided to modify a ruler:

```
PROCEDURE SetGrid (r: Ruler; h: INTEGER)
```

Adjustment modes

For longer text stretches, it becomes necessary to break lines that don't fit between the text view's margins.

This text setting may occur in several different ways. Text may be left adjusted (left flush), right adjusted (right flush), centered, or aligned to both left and right margins (full justification). These four adjustment modes can be controlled by two option flags: *TextRulers.leftAdjust* and *TextRulers.rightAdjust*.

leftAdjust	**rightAdjust**	**adjustment mode**
FALSE	FALSE	centered
FALSE	TRUE	right adjusted
TRUE	FALSE	left adjusted (default)
TRUE	TRUE	fully adjusted

Table 5-14. Adjustment modes

These adjustment modes can be included in or excluded from a more comprehensive option set, which is represented as a *SET*:

opts-: SET *(* opts is subset of {leftAdjust, rightAdjust, pageBreak, rightFixed, noBreakInside, parJoin; the other set elements are reserved *)*

When created with *TextRulers.dir.New(NIL)*, a ruler's option set is *{leftAdjust}*.
The following auxiliary procedures are provided to modify a ruler:

```
PROCEDURE SetLeftFlush (r: Ruler)
PROCEDURE SetCentered (r: Ruler)
PROCEDURE SetRightFlush (r: Ruler)
PROCEDURE SetJustified (r: Ruler)
```

Page breaks
A ruler can force a page break, by including the *TextRulers.pageBreak* option element in the *opts* set.
The element *TextRulers.noBreakInside* is valid for the entire scope of a paragraph. As the name indicates, it forces the beginning of the paragraph to start on a new page, if this isn't the case anyway and if the page would be broken before the next paragraph.
The element *TextRulers.parJoin* is similar to *noBreakInside*, except that it additionally forces at least the first line of the next paragraph onto the same page.

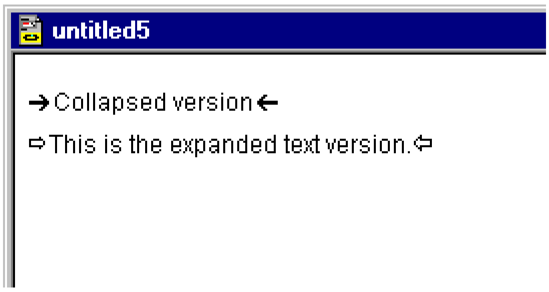
When created with *TextRulers.dir.New(NIL)*, a ruler has none of the above special attributes.
The following auxiliary proceduresare provided to modify a ruler:

```
PROCEDURE SetPageBreak (r: Ruler)
PROCEDURE SetNoBreakInside (r: Ruler)
PROCEDURE SetParJoin (r: Ruler)
```

This concludes our discussion of ruler attributes. Ruler attributes apply to the ruler, or to all the paragraphs in the ruler's scope. In contrast, *character attributes* apply to arbitrary stretches of characters. The BlackBox text model supports all font attributes plus color and vertical offsets.

Rulers are views that can be inserted into a text, since texts are containers. Rulers are special in that they are known to the text system, since a text view needs to know about a ruler's paragraph attributes when setting the text. However, arbitrary other views, which are not known to the text system, may be inserted into a text. Particularly interesting effects can be achieved when these views know about texts. Fold views and link views are examples of such text-aware views.

The following example shows how fold views can be generated. Fold views are standard objects of the BlackBox Component Builder; they are implemented in the *StdFolds* module, only using BlackBox and its Text subsystem. Fold views come in pairs: a left fold and a right fold. The left fold contains a hidden text. When the user clicks on one of the fold views, the text between the two views is swapped with the hidden text. Typically, fold views are used to hide a large text behind a short one; e.g., a book chapter may be hidden, while the chapter's name is visible. This is a way to achieve abstraction of document parts, by hiding details. For this reason, one state of a fold view is called its *collapsed* state, and the other is called its expanded *state*. Technically there is no reason why the hidden text should be longer than the visible one; the naming is more a reflection of the typical use of folds.



➔Collapsed version⇐

⇨This is the expanded text version.⇦

Figure 5-15. Fold views in collapsed and expanded states

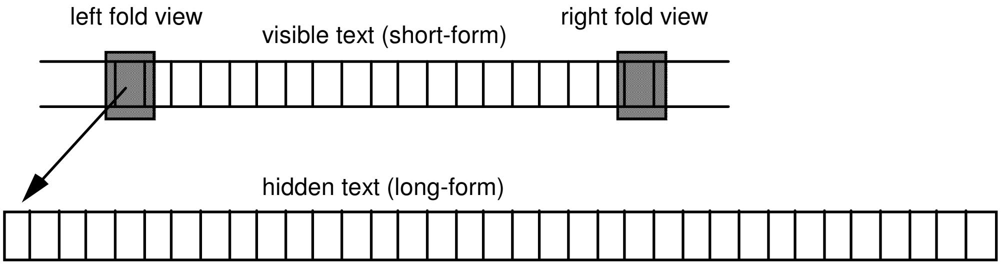


Figure 5-16. Structure of text folds

```
MODULE ObxPDBRep3;

   IMPORT Views, TextModels, TextMappers, TextViews, StdFolds, ObxPhoneDB;
   PROCEDURE WriteOpenFold (VAR f: TextMappers.Formatter;
                                          IN shortForm: ARRAY OF CHAR);
      VAR fold: StdFolds.Fold; t: TextModels.Model;
   BEGIN
      t := TextModels.dir.NewFromString(shortForm);    (* convert a string into a text model *)
      fold := StdFolds.dir.New(StdFolds.expanded, "", t);
      f.WriteView(fold)
   END WriteOpenFold;

   PROCEDURE WriteCloseFold (VAR f: TextMappers.Formatter);
      VAR fold: StdFolds.Fold; len: INTEGER;
   BEGIN
      fold := StdFolds.dir.New(StdFolds.expanded, "", NIL);
      f.WriteView(fold);
      fold.Flip;   (* swap long-form text, now between the two fold views, with hidden short-form text *)
      len := f.rider.Base().Length();   (* determine the text carrier's new length *)
      f.SetPos(len)   (* position the formatter to the end of the text *)
```

```
  END WriteCloseFold;

  PROCEDURE GenReport*;
    VAR t: TextModels.Model; f: TextMappers.Formatter; v: TextViews.View;
      i: INTEGER; name, number: ObxPhoneDB.String;
  BEGIN
    t := TextModels.dir.New();     (* create empty text carrier *)
    f.ConnectTo(t);                (* connect a formatter to the text *)
    i := 0;
    ObxPhoneDB.LookupByIndex(i, name, number);
    WHILE name # "" DO
      WriteOpenFold(f, name$);    (* write left fold view into text, with name as its short-form text *)
      (* now write the long-form text *)
      f.WriteString(name);        (* first string *)
      f.WriteTab;                 (* tab character *)
      f.WriteString(number);      (* second string *)
      WriteCloseFold(f);          (* write closing fold, and swap short- and long-form texts *)
      f.WriteLn;                  (* carriage return *)
      INC(i);
      ObxPhoneDB.LookupByIndex(i, name, number)
    END;
    v := TextViews.dir.New(t);     (* create a text view for the text generated above *)
    Views.OpenView(v)              (* open the text view in its own window *)
  END GenReport;

END ObxPDBRep3.
```

Listing 5-17. Writing the phone database using folds

Procedure *WriteCloseFold* has two interesting properties. First, the fold is written in its expanded form. This is reasonable, since the expanded form is often more complex than the collapsed form, which thus can be used conveniently in *WriteOpenFold* (note the useful text directory procedure *TextModels.dir.NewFromString*). The collapsed version is more complex, and thus it is convenient to use the already existing text formatter. After the left fold view, the expanded text, and the right fold view have been written, the text is collapsed using the *StdFolds.Flip* procedure.
This leads us to the second interesting point. Collapsing the text modifies the text model. This invalidates all riders, and consequently all formatters, that operate on the text. For example, formatter *f* has always been at the end of the created text, but now that the text is collapsed, its position is beyond the end of the text and thus illegal. For this reason, the correct position is set up again.

This is a general observation when working with mutable carriers: whenever you modify a carrier's state (or whenever someone else may do so!) you possibly have to update some state of your riders. In this example, the text's length was modified, which made it necessary to update the formatter's position.

The last example in our series of demonstrations on how to generate texts uses link views. Link views are standard objects of the BlackBox Component Builder; they are implemented in the *StdLinks* module, only using BlackBox and its Text subsystem. Like fold views, link views are used in pairs. Their implementation uses a special interface of the text subsystem (*TextControllers.FilterPollCursorMsg*, *TextControllers.FilterTrackMsg*) to cause the cursor to turn into a hand cursor when it lies above the text stretch between the pair of link views. This behavior is known from typical hypertext systems, such as Web browsers, when the mouse is moved over a text stretch that symbolizes a hyperlink. This use of link views is what gave them their name. However, they are much more general than simple hyperlink delimiters. A hyperlink

contains a passive reference to another document or to another location in the same document. For example, a document reference may be stored as a string, such as "http://www.oberon.ch" or "BlackBox/System/Rsrc/About".
Link views in contrast store a Component Pascal command, optionally with one or two string literals as parameters. The command is executed when the user clicks the mouse button when the mouse is between the two link views.
The command is stored in the left link view of a pair. Typical commands are:

```
DevDebug.ShowLoadedModules

StdCmds.OpenDoc('System/Rsrc/Menus')

StdCmds.OpenAux('System/Rsrc/About', 'About BlackBox')
```

In our example, we will create link views with commands similar to the following one:

```
ObxPDBRep4.Log('Daffy Duck    310-555-1212')
```

Note: consult the documentation of module *StdInterpreter*, since there may be some restrictions on the possible parameter lists supported for such commands.

The *Log* procedure of our example module simply writes the parameter string to the log. In a more realistic example, the parameter string could be used to dial the respective number.

```
MODULE ObxPDBRep4;

   IMPORT Views, TextModels, TextMappers, TextViews, StdLinks, StdLog, ObxPhoneDB;

   CONST cmdStart = "ObxPDBRep4.Log('"; cmdEnd = "')";

   PROCEDURE GenReport*;
      VAR t: TextModels.Model; f: TextMappers.Formatter; v: TextViews.View;
         i: INTEGER; name, number: ObxPhoneDB.String; link: StdLinks.Link;
   BEGIN
      t := TextModels.dir.New();    (* create empty text carrier *)
      f.ConnectTo(t);               (* connect a formatter to the text *)
      i := 0;
      ObxPhoneDB.LookupByIndex(i, name, number);
      WHILE name # "" DO
         link := StdLinks.dir.NewLink(cmdStart + name + "    " + number + cmdEnd);
         f.WriteView(link);
         f.WriteString(name);       (* the string shown between the pair of link views *)
         link := StdLinks.dir.NewLink("");
         f.WriteView(link);
         f.WriteLn;                 (* carriage return *)
         INC(i);
         ObxPhoneDB.LookupByIndex(i, name, number)
      END;
      v := TextViews.dir.New(t);    (* create a text view for the text generated above *)
      Views.OpenView(v)            (* open the text view in its own window *)
   END GenReport;
```

```
    PROCEDURE Log* (param: ARRAY OF CHAR);
    BEGIN
        StdLog.String(param); StdLog.Ln
    END Log;

END ObxPDBRep4.
```

Listing 5-18. Writing the phone database using links

Note that commands are normal procedure calls; like all procedure calls from outside of a module, they can only call those procedures of the module that are exported. Thus procedure *Log* of the above module must be exported, otherwise the program wouldn't work.

Another interesting property of link views is that they display themselves in different ways depending on whether the text view via which the user interacts with them displays or hides marks. If there are several text views on the same model, the user can independently switch marks on or off, such as inactive soft hyphens, ruler views or link views. The text subsystem provides an interface which allows views to distinguish between the two states (*TextSetters.Pref*). In particular, a view may decide to hide itself when no marks are shown, by setting its width to zero in this case. This is what link views do: when text marks are switched off, they are invisible, so that only the text between them is visible. Typically, this text is blue and underlined, to indicate that it represents a link. A ruler in contrast always occupies the whole width between its paragraph's left and right margin, but it sets its height to zero when the text marks are turned off. We will learn more about how containers and embedded views can cooperate to achieve such effects in Part III of this tutorial.

The above example demonstrates that link views can cause arbitrary behavior when the user clicks between them. This generality is powerful, but it should be used with caution. A command in a document is a kind of "implicit import" relation between the document and the command's module, and thus constitutes a dependency. This is not fundamentally different from the dependencies that are caused by views embedded in the document, they also function correctly only when their implementing component(s) are available. Like all dependencies, those caused by links, buttons embedded in a control, editors embedded in documents, and so on also need to be tracked and updated when necessary.

5.2 The Model-View-Controller pattern applied to texts

The next major examples demonstrate how existing text can be read. It is assumed that the text to be processed is visible, in the topmost window. On how to access texts that are stored in files, please refer to the on-line Obx examples *ObxOpen0*, *ObxOpen1*, and *ObxAscii*. Before starting with the specifics of text reading, we need to discuss how the topmost window's contents can be accessed from within a Component Pascal command.

Basically, this is done by calling *Controllers.FocusView()*. This function returns *NIL* if no document is open. Otherwise it returns the topmost window's contents, which is a view.

We have learned earlier that views, whose common base type is *Views.View*, are *the* pivotal BlackBox abstractions for interactive components, such as editors and controls. Everything revolves around views. In simple cases, such as controls, a view implementation only implements an extension to *Views.View*, and that's all. In more complex cases, the data that are visualized by a view are split off into a separate model. This was discussed in detail in Chapter 2.

Since container implementations are among the most complex programming tasks for which the BlackBox Component Builder was designed, the framework provides some help in how a container should be constructed. The abstract design of a container is given in module *Containers*. This module declares extensions of the types *Views.View*, *Models.Model* and *Controllers.Controller*; these types constitute a miniframework for container construction.

Typical containers are implemented in at least four modules: one for the model, one for the view, one for the controller, and one for standard commands. For example, the text subsystem contains the modules *TextModels*, *TextViews*, *TextControllers*, *TextCmds*; plus the auxiliary modules *TextMappers*, *TextRulers* and *TextSetters*. The form subsystem consists of *FormModels*, *FormViews*, *FormControllers*, and *FormCmds*.

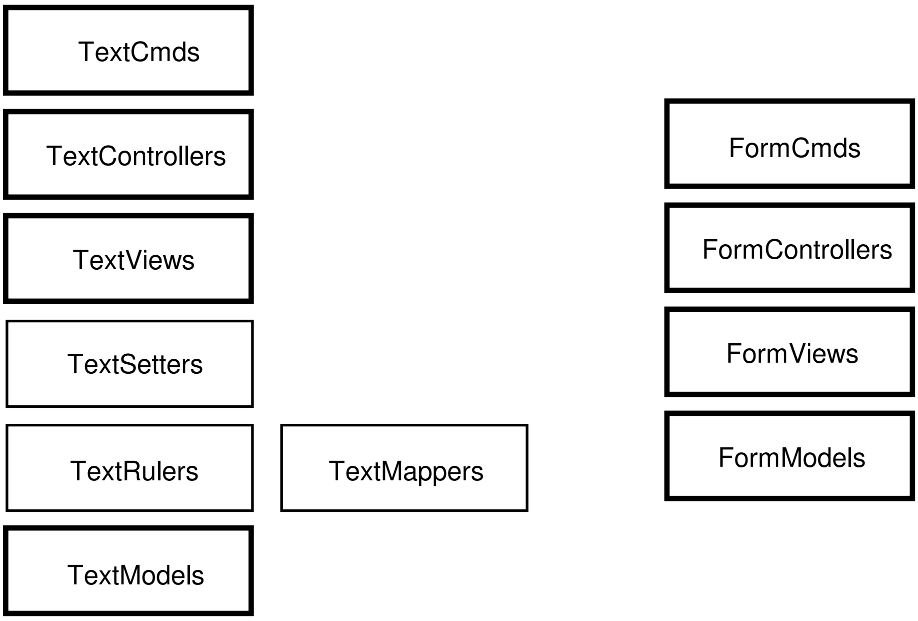


Figure 5-19. Variations of the generic container module decomposition

We won't look at all the details of these types; but it is important to know that a container object is not only split into a model and one or several views, but that a view is further split into a view and a so-called controller. For the time being, we can regard the management of selections and carets as the main functionality that a controller encapsulates. This is why we now look at controllers, because some of the following examples will operate on the current selection.

Once you have a controller, you can obtain its view by calling its *ThisView* procedure. Other procedures allow to get or set the current selection.

There may be several views displaying the same model (many-to-one relationship), but there is exactly one controller per view (one-to-one relationship). (This holds for views derived from *Containers.View*; other views may or may not have a separate controller.)You can get a controller's view by calling its *ThisView* procedure, and you can get a container view's controller by calling its *ThisController* procedure.

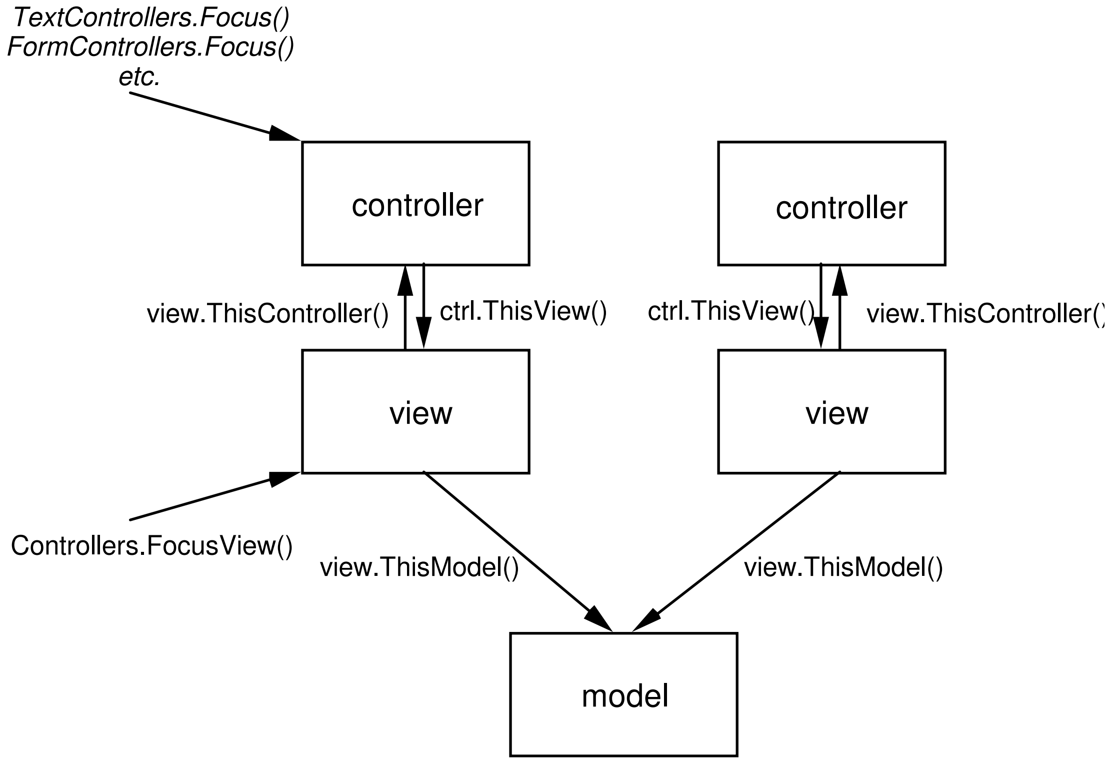


Figure 5-20. Model-View-Controller separation

More specific container abstractions typically provide a suitable *Focus* function which yields the controller of the currently focused view, if there is one and if it has the desired type.

We now know enough about the container scenario that we can tell how to access focus view, model, or controller:

```
The following declarations are assumed:
    VAR v: Views.View; m: Models.Model; c: Controllers.Controller;

Simple view:
    v := Controllers.FocusView();

View with a model:
    v := Controllers.FocusView();
    IF v # NIL THEN m := v.ThisModel() ELSE m := NIL END;

Extension of a Containers.View:
    v := Controllers.FocusView();
    IF v # NIL THEN m := v.ThisModel() ELSE m := NIL END;
    IF (v # NIL) & (v IS Containers.View) THEN
        c := v(Containers.View).ThisController()
    ELSE
        c := NIL
    END
```

Listing 5-21. Accessing focus view, model, and controller

Typical container abstractions, such as the Text and Form subsystems, use the above mechanisms to provide more convenient access functions tailored to the specific container type. For example, module *TextControllers* exports the function *Focus*, which returns the focus view's controller *if and only if* the focus view is a text view. Otherwise it returns *NIL*. This approach is appropriate, since once you have a container controller, you can easily access its view, its model, information about its selection or caret, and so on. The typical code pattern that arises is some subset of the following one, depending on the controller's parts that you need:

```
PROCEDURE SomeCommand*;
    VAR c: TextControllers.Controller;
        v: TextViews.View; t: TextModels.Model; from, to, pos: INTEGER;
        ...
BEGIN
    c := TextControllers.Focus();
    IF c # NIL THEN
        ...
        v := c.ThisView();    (* if you need the text view... *)
        ...
        t := v.ThisModel();    (* or if you need the text model... *)
        ...
        IF c.HasSelection() THEN    (* or if you need the selection range... *)
            c.GetSelection(from, to);
            ...
        END;
        ...
        IF c.HasCaret() THEN    (* or if you need the caret position... *)
            pos := c.CaretPos();
            ...
        END;
        ...
    ELSE    (* no open document window, or focus has wrong type *)
        (* no error handling is necessary if this command is guarded by appropriate guard command *)
    END
END SomeCommand;
```

Listing 5-22. Code pattern for TextControllers.Controller

A menu command can use the guard *TextCmds.FocusGuard* to make sure that the command is only executed when a text view is focus. *TextCmds.SelectionGuard* additionally checks whether there exists some (non-empty) text selection.

5.3 Reading text

Now that we have seen how to access the focus views' controller, we can write our first command that reads the focus view's text. The command counts the number of Latin-1 characters (1 byte), Unicode characters above Latin-1 (2 byte), and embedded views.

```
MODULE ObxCount0;

    IMPORT TextModels, TextControllers, StdLog;

    PROCEDURE Do*;
    (** use TextCmds.SelectionGuard as guard for this command **)
        VAR c: TextControllers.Controller; from, to, schars, chars, views: INTEGER;
```

```
            rd: TextModels.Reader;
    BEGIN
        c := TextControllers.Focus();
        IF (c # NIL) & c.HasSelection() THEN
            c.GetSelection(from, to);    (* get selection range; from < to *)
            rd := c.text.NewReader(NIL);    (* create a new reader for this text model *)
            rd.SetPos(from);    (* set the reader to beginning of selection *)
            rd.Read;                (* read the first element of the text selection *)
            schars := 0; chars := 0; views := 0;    (* counter variables *)
            WHILE rd.Pos() # to DO    (* read all elements of the text selection *)
                IF rd.view # NIL THEN    (* element is a view *)
                    INC(views)
                ELSIF rd.char < 100X THEN    (* element is a Latin-1 character *)
                    INC(schars)
                ELSE    (* element is Unicode character *)
                    INC(chars)
                END;
                rd.Read        (* read next element of the text selection *)
            END;
            StdLog.String("Latin-1 characters: "); StdLog.Int(schars); StdLog.Ln;
            StdLog.String("Unicode characters: "); StdLog.Int(chars); StdLog.Ln;
            StdLog.String("Views: "); StdLog.Int(views); StdLog.Ln;
            StdLog.Ln
        END
    END Do;

END ObxCount0.
```

Listing 5-23. Counting Latin-1 characters, Unicode characters, and views in a text

The above example uses a *TextModels.Reader*, which is the text rider type for reading. Each reader maintains a current position on its base text; there can be several independent readers on the same text simultaneously. A text reader has the following definition:

```
TYPE
    Reader = POINTER TO ABSTRACT RECORD
        eot: BOOLEAN;
        attr: TextModels.Attributes;
        char: CHAR;
        view: Views.View;
        w, h: INTEGER;
        (rd: Reader) Base (): TextModels.Model, NEW, ABSTRACT;
        (rd: Reader) Pos (): INTEGER, NEW, ABSTRACT;
        (rd: Reader) SetPos (pos: INTEGER), NEW, ABSTRACT;
        (rd: Reader) Read, NEW, ABSTRACT;
        (rd: Reader) ReadChar (OUT ch: CHAR), NEW, ABSTRACT;
        (rd: Reader) ReadView (OUT v: Views.View), NEW, ABSTRACT;
        (rd: Reader) ReadRun (OUT attr: TextModels.Attributes), NEW, ABSTRACT;
        (rd: Reader) ReadPrev, NEW, ABSTRACT;
        (rd: Reader) ReadPrevChar (OUT ch: CHAR), NEW, ABSTRACT;
        (rd: Reader) ReadPrevView (OUT v: Views.View), NEW, ABSTRACT;
        (rd: Reader) ReadPrevRun (OUT attr: TextModels.Attributes), NEW, ABSTRACT
```

```
END;
```

Listing 5-24. Definition of TextModels.Reader

Given a text model *text*, a reader can be created by calling *rd := text.NewReader(NIL)*. Such a newly allocated reader will be positioned at the beginning of the text; i.e., at *rd.Pos() = 0*. The position can be changed whenever necessary by calling *rd.SetPos(pos)*, with *pos* in the range of *0..text.Length()*. The reader's text model can be obtained by calling its *Base* procedure.

Reading of the next text element is done by calling the reader's *Read* procedure, which increases the reader's current position unless the end of text has been reached, in which case the reader's *eot* field is set. After *Reaa* is called, the character code of the read element can be found in the reader's *char* field. If the character value is less than *100X*, the character fits in the Latin-1 range (1-byte character), otherwise it's a 2-byte Unicode character. For convenience, the auxiliary reader procedure *ReadChar* is provided, which performs a *Read* and returns the contents of *char.*

What happens if a view is read? In most cases, *char* is set to the reserved value *TextModels.viewcode* (2X). However, this is not necessarily the case. A text-aware view may choose to represent an arbitrary character code by using a special preference message (*TextModels.Pref*). In order to find out whether a view has been read, the field *view* should be tested. It returns the view just read, or *NIL* if it wasn't a view.
When a view has been read, the fields *w* and *h* indicate the size of the view in universal units. Otherwise, these fields are undefined.
It is expected that the implementation of a text model optimizes access to views. For this purpose, the auxiliary reader procedure *ReadView* is provided, which reads the next view after the current position, skipping as many non-view elements as necessary.

After an element (character or view) has been read, its text attributes are returned in the reader's *attr* field. The text attributes consist of font attributes, color and vertical offset; as described earlier. For efficiency and convenience reasons, the auxiliary reader procedure *ReadRun* is provided, which advances the reader position until the attributes change from their current setting. Text runs that have the same attributes can be drawn on the screen with one port procedure (*Ports.Frame.DrawString*), which is considerably faster than drawing them character by character.

Normally, text is read forward; from lower positions to higher positions. The reader procedures *ReadPrev*, *ReadPrevView* and *ReadPrevRun* are symmetrical to their *Read*, *ReadView* and *ReadRun* counterparts, but they read backwards. In particular, *ReadPrevView* is used by a text view's so-called text setter to find the most recent ruler view, which defines how the text must be set (The auxiliary procedure *TextRulers.GetValidRuler* implements this search.)
When one of the reading procedures has been called (whether one of the forward or one of the backward reading procedures), the reader's *eot* field is set. If an element could be read, *eot* is set to *FALSE*. If no element could be read (reading forward at the end of the text, or reading backward at the beginning of the text), *eot* is set to *TRUE*.

Typically, the following code pattern arises when working with a text reader. *condition* may be something like *rd.Pos() # end* or *~rd.eot*.

```
    VAR rd: TextModels.Reader; start: INTEGER;
BEGIN
    ... define start ...
    rd := text.NewReader(NIL);
    rd.SetPos(start);    (* only necessary if start # 0 *)
    rd.Read;
```

```
WHILE condition DO
   ... consume text element denoted by rd ...
   rd.Read
END;
```

Listing 5-25. Code pattern for TextModels.Reader

The next example uses a text scanner. Scanners are reading text mappers; like formatters, they are defined in module *TextMappers*. Text scanners are similar to text readers, except that they operate on whole symbols rather than characters. Strings and numbers are examples of Component Pascal symbols that are supported by text scanners.

When you compare the definition of a *TextMappers.Scanner* with the one of *TextModels.Reader*, you'll note that a scanner can also return characters (field *char*) and views with their sizes (fields *view*, *w*, and *h*); that a scanner has a position (*Pos*, *SetPos*), but that a scanner cannot read backwards. *Scan* corresponds to a reader's *Read* procedure. It skips white space (blanks, carriage returns) until it either reaches the end of the text, which is signaled by setting *type* to *TextMappers.eot*, or until it has read a symbol. The type of symbol that was read is returned in *type*. It may be one of the *TextMappers* constants

char, string, int, real, bool, set, view, tab, line, para, eot, invalid.

By default, views and control characters are treated as white space and thus ignored. Using an option set (*opts*, *SetOpts*) it is possible to treat them as valid symbols, too. The set element *TextMappers.returnViews* indicates that views should be recognized, while *TextMappers.returnCtrlChars* indicates that the three control characters for tabulators (*tab*), carriage returns (*line*) and paragraphs (*para*) should be recognized.

When a symbol starts like a Component Pascal string or identifier, the scanner attempts to read a string. When successful, *type* is set to *string*; which may also be a short string, i.e., only containing Latin-1 characters. The option *TextMappers.returnQualIdents* lets the scanner recognize complete qualified identifiers as single symbols; e.g., the string *ThisMod.ThatObj* . The length of the string is returned in *len*.

If it is possible to interpret the symbol as an integer or real number, *type* is set accordingly, as are the fields *int* or *real*. Integer numbers are recognized in different number bases, the base that was recognized is returned in *base* (10 for decimal numbers).

Special characters such as *!@#$* are interpreted as *type = TextMappers.char*, which may also be a short character, i.e., a Latin-1 character.

By default, Boolean values and sets are not interpreted, except if the option elements *TextMappers.interpretBool* or *TextMappers.interpretSets* are set. Sets use the normal Component Pascal syntax, while Booleans are represented as *$TRUE* and *$FALSE*.

The scanner field *start* is the position of the first character of the most recently read symbol. The fields *lines* and *paras* count the number of carriage return and paragraph characters since the last *SetPos* (see below).

The scanner's *Skip* procedure is an auxiliary procedure that advances the scanner position until all white space characters - as specified in the *opts* set - are skipped and the first character of the symbol has been read. This character is returned in *Skip*'s *ch* parameter.

Since there may be an arbitrary number of specialized text mappers, in addition to the ones provided by *TextMappers*, the text model cannot create text mapper objects, like it creates text readers and writers. Instead, scanners (and formatters likewise) are connected to their texts explicitly, by calling their *ConnectTo* procedures. They may be called repeatedly to connect the same scanner to different models.

```
Scanner = RECORD
   rider-: TextModels.Reader;
   opts-: SET;
   type: INTEGER;
   start, lines, paras: INTEGER;
   char: CHAR;
```

```
        int, base: INTEGER;
        real: REAL;
        bool: BOOLEAN;
        set: SET;
        len: INTEGER;
        string: TextMappers.String;
        view: Views.View;
        w, h: INTEGER;
        (VAR s: Scanner) ConnectTo (text: TextModels.Model), NEW;
        (VAR s: Scanner) Pos (): INTEGER, NEW;
        (VAR s: Scanner) SetPos (pos: INTEGER), NEW;
        (VAR s: Scanner) SetOpts (opts: SET), NEW;
        (VAR s: Scanner) Scan, NEW;
        (VAR s: Scanner) Skip (OUT ch: CHAR), NEW
    END;
```

Listing 5-26. Definition of TextModels.Scanner

The following example uses a text scanner to count the number of integer numbers, real numbers, and strings found in a text.

```
MODULE ObxCount1;

    IMPORT TextModels, TextMappers, TextControllers, StdLog;

    PROCEDURE Do*;
    (** use TextCmds.SelectionGuard as guard for this command **)
        VAR c: TextControllers.Controller; from, to, ints, reals, strings: INTEGER;
            s: TextMappers.Scanner;
    BEGIN
        c := TextControllers.Focus();
        IF (c # NIL) & c.HasSelection() THEN
            c.GetSelection(from, to);    (* get selection range; from < to *)
            s.ConnectTo(c.text);    (* connect scanner to this text model *)
            s.SetPos(from);    (* set the reader to beginning of selection *)
            s.Scan;                (* read the first symbol of the text selection *)
            ints := 0; reals := 0; strings := 0;    (* counter variables *)
            WHILE s.Pos() < to DO    (* read all symbols starting in the text selection *)
                IF s.type = TextMappers.int THEN    (* symbol is an integer number *)
                    INC(ints)
                ELSIF s.type = TextMappers.real THEN    (* symbol is a real number *)
                    INC(reals)
                ELSIF s.type = TextMappers.string THEN    (* symbol is a string/identifier *)
                    INC(strings)
                END;
                s.Scan        (* read next symbol of the text selection *)
            END;
            StdLog.String("Integers: "); StdLog.Int(ints); StdLog.Ln;
            StdLog.String("Reals: "); StdLog.Int(reals); StdLog.Ln;
            StdLog.String("Strings: "); StdLog.Int(strings); StdLog.Ln;
            StdLog.Ln
        END
    END Do;

END ObxCount1.
```

Listing 5-27. Counting integers, reals, and strings in a text

The most typical code pattern for a scanner looks similar to the code pattern for a reader. *condition* may be something like *s.Pos() < end* or *s.type # TextMappers.eot*.

```
        VAR s: TextMappers.Scanner; start: INTEGER;
    BEGIN
        ... define start ...
        s.ConnectTo(text);
        s.SetPos(start);    (* only necessary if start # 0 *)
        s.Scan;
        WHILE condition DO
            IF s.type = TextMappers.int THEN
                ... consume s.int ...
            ELSIF s.type = TextMappers.real THEN
                ... consume s.real ...
            ELSIF ...
                ...
            END;
            s.Scan
        END;
```

Listing 5-28. Code pattern for TextMappers.Scanner

5.4 Modifying text

In the previous sections, we have seen how new texts can be created using writers or formatters, and how existing texts can be read using readers or scanners. Many text commands first read a text stretch, perform some computation on it, and then replace it by the result of the computation. Examples are commands that set the font of a selection, set the color of a selection, or turn all small letters into capital letters.
The following example command reads the text selection and checks whether it is a string. If so, it interprets the string as a name and looks the name up in our example phone database. Then it replaces the name by the phone number.

```
MODULE ObxLookup0;

    IMPORT TextModels, TextMappers, TextControllers, ObxPhoneDB;

    PROCEDURE Do*;
    (** use TextCmds.SelectionGuard as guard command **)
        VAR c: TextControllers.Controller; buf: TextModels.Model; from, to: INTEGER;
            s: TextMappers.Scanner; f: TextMappers.Formatter; number: ObxPhoneDB.String;
    BEGIN
        c := TextControllers.Focus();
        IF (c # NIL) & c.HasSelection() THEN
            c.GetSelection(from, to);
            s.ConnectTo(c.text);
            s.SetPos(from);
            s.Scan;
            IF s.type = TextMappers.string THEN
                buf := TextModels.CloneOf(c.text);
                f.ConnectTo(buf);
                ObxPhoneDB.LookupByName(s.string$, number);
                f.WriteString(number);
                from := s.start; to := s.Pos() - 1;    (* scanner has already read on character beyond string! *)
                c.text.Delete(from, to);                        (* delete name *)
                c.text.Insert(from, buf, 0, buf.Length());    (* move phone number from buffer into text *)
                c.SetSelection(from, from + LEN(number$))    (* select the phone number *)
            END
        END
    END Do;

END ObxLookup0.
```

Listing 5-29. Replacing a phone name by a phone number

The basic idea of the above command is that an auxiliary text object is used which acts as a buffer. The text selection is scanned, the scanned string is used as key to make a lookup in the phone database, the returned phone number is written into the buffer text, the original name is deleted from the text, and then the buffer contents is moved to where the name was. As a final step, the newly inserted text stretch is selected.
Note that the text model's *Delete* procedure shortens the text, while the *Insert* procedure makes the text longer. In particular, *Insert* doesn't overwrite existing text. *Insert* moves text, i.e., it removes them at the source and inserts them at the destination.

One way to make the above command more useful would be to accept a caret as input, instead of a selection. Then you could write a name and hit a keyboard shortcut for the command. The command then reads backwards from the caret position, until it finds the beginning of the name, and then substitutes the appropriate phone number for the name. This would be a simple way to implement keyboard macros.

Buffer texts, as used above, are particularly useful in helping to avoid screen flicker: since a buffer text is not displayed, it can be built up piecemeal, without causing any screen updates. When the buffer has been completed, its contents can be moved over to the visible destination text. This only causes one single screen update, which is fast and creates minimal flicker.

Moving text stretches from one text model to another one may be optimized if both text models have the same implementation. This is not necessarily the case, since different implementations of type *TextModels.Model* may coexist with each other. Two calls to *TextModels.dir.New()* may well return two text models that have different implementations.
For this reason, module *TextModels* exports the procedure *Clone*, which returns an empty text model of exactly the same type as its parameter. In fact, the standard text implementation even performs some further optimizations when a text model is cloned from another one, rather than allocated via *TextModels.dir.New()*.

We can now give a (slightly simplified) definition of *TextModels.Model*:

```
TYPE
    Model = POINTER TO ABSTRACT RECORD (Containers.Model)
        (m: Model) NewWriter (old: TextModels.Writer): TextModels.Writer, NEW, ABSTRACT;
        (m: Model) NewReader (old: TextModels.Reader): TextModels.Reader, NEW, ABSTRACT;
        (m: Model) Length (): INTEGER, NEW, ABSTRACT;
        (m: Model) Insert (pos: INTEGER; m0: TextModels.Model;
                                            beg0, end0: INTEGER), NEW, ABSTRACT;
        (m: Model) InsertCopy (pos: INTEGER; m0: TextModels.Model;
                                            beg0, end0: INTEGER), NEW, ABSTRACT;
        (m: Model) Delete (beg, end: INTEGER), NEW, ABSTRACT;
        (m: Model) Append (m0: TextModels.Model), NEW, ABSTRACT;
        (m: Model) Replace (beg, end: INTEGER; m0: TextModels.Model;
                                            beg0, end0: INTEGER), NEW, ABSTRACT;
        (m: Model) SetAttr (beg, end: INTEGER; attr: TextModels.Attributes), NEW, ABSTRACT;
    END;
```

Listing 5-30. Definition of TextModels.Model

We have already met most of its procedures. *InsertCopy* is similar to *Insert*, but instead of moving, it copies a text stretch, without modifying the source. *Delete* removes a text stretch.
Insert, *InsertCopy*, and *Delete* are the elementary text operations. *Append* and *Replace* are provided for convenience and efficiency; they can be expressed in terms of the elementary operations:

```
  dst.Append(src)  ≈  dst.Insert(dst.Length(), src, 0, src.Length())
```

i.e., it moves the whole contents of the source text to the end of the destination text, thereby leaving an empty source text.

```
  t.Replace(b, e, t1, b1, e1)  ≈  t.Delete(b, e); t.Insert(b, t1, b1, e1)
```

i.e., it overwrites some part of the destination text by some part of the source text (which is thereby removed). Note that source and destination texts must be different.

SetAttr allows to set the text attributes of a whole text stretch.

5.5 Text scripts

When you have successfully applied the command *ObxLookup0.Do* to a text stretch, you can try out something that may surprise you: you can reverse ("undo") the operation by executing the *Edit->Undo* command! With *Edit->Redo* you can reverse the operation's reversal.
This capability is surprising because we have only implemented the plain *Do* procedure, but no code for undoing or redoing it. How come that the framework is able to undo/redo the operation anyway?

It is possible because the text model procedures which modify their model's state, such as *Model.Delete* and *Model.Insert*, are implemented in a particular way. They don't execute the modification directly. Instead, they create special operation objects and register them in the framework. The framework then can call the operation's appropriate procedure for performing the actual do/undo/redo functionality. This mechanism has been described in Chapter 3.

The following example shows how a sequence of text-modifying procedures is bracketed by *BeginScript* and *EndScript* to make the whole sequence undoable. *BeginScript* returns a script object, it gets the name of the compound operation as input.

```
MODULE ObxLookup1;

  IMPORT Stores, Models, TextModels, TextMappers, TextControllers, ObxPhoneDB;
```

```
   PROCEDURE Do*;
   (** use TextCmds.SelectionGuard as guard command **)
      VAR c: TextControllers.Controller; buf: TextModels.Model; from, to: INTEGER;
         s: TextMappers.Scanner; f: TextMappers.Formatter; number: ObxPhoneDB.String;
         script: Stores.Operation;
   BEGIN
      c := TextControllers.Focus();
      IF (c # NIL) & c.HasSelection() THEN
         c.GetSelection(from, to);
         s.ConnectTo(c.text);
         s.SetPos(from);
         s.Scan;
         IF s.type = TextMappers.string THEN
            buf := TextModels.CloneOf(c.text);
            f.ConnectTo(buf);
            ObxPhoneDB.LookupByName(s.string$, number);
            f.WriteString(number);
            from := s.start; to := s.Pos() - 1;    (* scanner has already read on character beyond string! *)
            Models.BeginScript(c.text, "#Obx:Lookup", script);
            c.text.Delete(from, to);                        (* delete name *)
            c.text.Insert(from, buf, 0, buf.Length());    (* move phone number from buffer into text *)
            Models.EndScript(c.text, script);
            c.SetSelection(from, from + LEN(number$))    (* select the phone number *)
         END
      END
   END Do;

END ObxLookup1.
```

Listing 5-31. Example of a script (compound operation)

In *Models.BeginScript*, the name of the compound operation is given. It is mapped by the string mapping facility of BlackBox, because this name may be displayed to the user (in the undo/redo menu item).

In modules *ObxLookup0* and *ObxLookup1* use the following statement sequence to replace the name with the number:

```
   c.text.Delete(from, to);
   c.text.Insert(from, buf, 0, buf.Length());
```

To combine these two operations into a single compound operation (as experienced by the user), it was necessary to bracket the two calls with the *BeginScript / EndScript* pair. In this special case here, there actually would be a better solution:

```
   c.text.Replace(from, to, buf, 0, buf.Length());
```

TextModels.Model.Replace is a powerful procedure of which *Insert*, *Delete*, and *Append* are special cases (provided for convenience):

```
   t.Delete(beg, end)                =    t.Replace(beg, end, t, 0, 0)
   t.Insert(pos, t0, beg, end)       =    t.Replace(pos, pos, t0, beg, end)
   t.Append(t0)                      =    t.Replace(t.Length(), t.Length(), t0, 0, t0.Length())
```

Note that *InsertCopy* cannot be expressed in terms of *Replace*:

```
t.InsertCopy(pos, t0, beg, end)    #    t.Replace(pos, pos, t0, beg, end)
```

The reason is that *Replace*, like *Insert*, deletes the source text. *InsertCopy* on the other hand is a pure copying operation which doesn't modify the source text (unless source and destination texts are identical and the destination position lies within the source text range).

5.6 Summary

We now conclude the discussion of the BlackBox Component Builder's Text subsystem. The goal of this tutorial was to give an overview over a relatively complex extension subsystem; to show the use of the previous part's design patterns; and to show the typical Text code patterns.
The treatment of the text subsystem cannot be complete, there are still many details for which the reference documentation of the various modules must be consulted. However, the general scope and typical applications should have become clearer. While we started with as little theory as possible and used simple examples only, the basic structure of a complex container subsystem will have become clearer by now.

Before we give a list of further on-line examples that you may want to study, here is a list of all standard modules of the Text subsystem (lowest module in the hierarchy first), with the most important types that they export:

module / type	**description**
TextModels	abstraction and default implementation of text models
Attributes	attributes of a text elem (font, color, vertical offset)
Context	link between a text model and a view embedded in it
Directory	factory for text models
Model	text carrier, factory for readers and writers
Reader	rider for element-wise text reading
Writer	rider for element-wise text writing
TextMappers	abstraction and implementation of text mappers for Component Pascal symbols
Formatter	mapper for symbol-wise text writing
Scanner	mapper for symbol-wise text reading
TextRulers	abstraction and default implementation for ruler views, which embody paragraph attributes
Attributes	attributes of text ruler (tabs, margins, grid, etc.)
Directory	factory for text rulers
Ruler	view for rulers
Style	model for rulers
TextSetters	abstraction and default implementation for text setters, which perform line / page breaking
Directory	factory for text setters
Reader	text mapper which collects text into units of text setting (non-breakable sequences) and determines their size
Setter	object which implements a text setting algorithm
TextViews	abstraction and default implementation of text views
Directory	factory for text views
View	view for text models

TextControllers		abstraction and default implementation of text controllers
	Controller	controller for text views
	Directory	factory for text controllers
TextCmds		command package with most important text commands

Table 5-32. Table of Text system modules and most important types

ObxHello0	write "Hello World" to log
ObxHello1	write "Hello World" to new text
ObxOpen0	open text document from a file, using standard file open dialog
ObxOpen1	modify a text document on a file, using standard file open and save dialogs
ObxCaps	change string to uppercase, using compound operation
ObxDb	manage sorted list of records
ObxTabs	transform some tabs into blanks
ObxMMerge	mail merge of template and database
ObxParCmd	interpret text which follows a command button
ObxLinks	create a directory text with hyperlinks
ObxAscii	traditional text file I/O

Table 5-33. Additional Obx on-line examples of Text subsystem

Part III: View Construction

Part I of the BlackBox tutorial gives an introduction to the design patterns that are used throughout BlackBox. To know these patterns makes it easier to understand and remember the more detailed design decisions in the various BlackBox modules.

Part II of the BlackBox tutorial demonstrates how the most important library components can be used: control, form, and text components.

Part III of the BlackBox tutorial demonstrates how new views can be developed, by giving a series of examples that gradually become more sophisticated.

6 View Construction

Views are the central abstraction in the BlackBox Component Builder. Everything revolves around views: most commands operate on views, windows display views, views can be internalized and externalized, views perform interaction with the user, and views may be embedded into other views.

6. 1 Introduction

This view programming tutorial consists of four sections, each of which introduces a special aspect of view programming. The section "message handling" (chapters 6.2 to 6.5) explains how the behavior of a view is defined through the view's message handlers, i.e., how the answering of preferences, controller messages and properties influences the view's relation to its environment and the view's reaction on user input. The second section explains the aspect of the model-view separation which allows multi view editing (6.6). The next section shows how the undo/redo facility of the BlackBox Component Framework can be used if the content of a view is changed with the help of operation objects (6.7). In the last section the special structure of all extensible BlackBox Component Framework modules is explained, namely the separation of the interface and the implementation of a view (6.8). Directory objects are the key to this design pattern.

6.2 Message handling

In this section, we present a sequence of increasingly more versatile implementations of a view object which displays a rectangular colored area. In particular, this view will handle *Preferences*, *Controller messages* and *Properties*.

Every view is a subtype of the abstract type *Views.View*. The concrete extension must at least implement the abstract procedure *Views.Restore* that draws the view's contents. The first example of our view simply draws a red rectangle. This version is about the simplest possible view implementation. Its major components are the declaration of a *Views.View* extension, the implementation of a *Views.Restore* procedure which draws the contents of the view, and a command procedure which allocates and initializes the view:

```
MODULE ObxViews0;

    IMPORT Views, Ports;
```

```
        TYPE View = POINTER TO RECORD (Views.View) END;

        PROCEDURE (v: View)  Restore (f: Views.Frame; l, t, r, b: INTEGER);
        BEGIN
            f.DrawRect(l, t, r, b, Ports.fill, Ports.red)
        END Restore;

        PROCEDURE Deposit*;
            VAR v: View;
        BEGIN
            NEW(v); Views.Deposit(v)
        END Deposit;

    END ObxViews0.
```

To execute this program, invoke the following command:

❗ "ObxViews0.Deposit; StdCmds.Open"

The result of this command is shown in Figure 6-1.

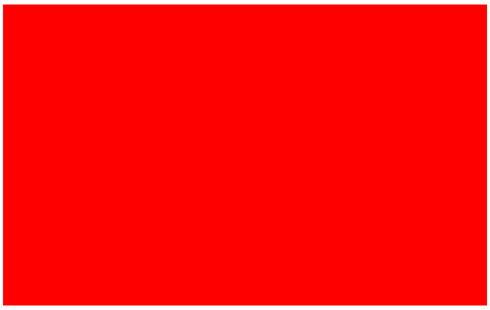


Figure 6-1: Result of "ObxViews0.Deposit; StdCmds.Open"

This simple program is completely functional already. The procedure *ObxViews0.Deposit* puts a newly allocated view into a system queue. The command *StdCmds.Open* in the command string above removes a view from this queue, and opens it in a new window. Note that whole sequences of commands must be enclosed between quotes, while for single commands the quotes may be omitted.

Instead of opening the view in a window, it could be pasted into the focus container (e.g., a text or a form):

❗ "ObxViews0.Deposit; StdCmds.PasteView"   >< *set the caret here before executing the command*

Like *StdCmds.Open,* the command *StdCmds.PasteView* removes a view from the queue, but pastes it to the focus view. The above command sequence could also be used in a menu, for example.

Every document which contains an *ObxViews0* view can be saved in a file, and this file can be opened again through the standard *Open...* menu entry. Our simple red rectangle will be displayed correctly in the newly opened document, provided the code file of the module *ObxViews0* is available.
A view which has been opened in its own window can also be saved in a file. When opened again, the document then consists of this single view only.

Every view performs its output operations and mouse/keyboard polling via a *Views.Frame* object. In the above example, *ObxViews0.View.Restore* uses its frame parameter *f* to draw a string. A frame can be regarded as a mapper object, in this case for both input and output simultaneously (here we need not be interested in the rider and carrier for this mapper, which are both defined in module *Ports*). A frame embodies coordinate transformations and clipping facilities.
If the view is displayed on the screen, the frame is a mapper on a screen port. If the view is printed, the frame is a mapper on a printer port. From the view's perspective, this difference is not relevant. Thus no special code is necessary to support printing.

The view may be copied and pasted into *containers* such as text views or form views. Additionally, the size of the view can be changed by selecting it and then manipulating the resize handles. All this functionality is offered by the BlackBox Component Framework and requires no further coding. If this default behavior is not convenient, it can be modified. In the following we will show how this is done.

6.3 Preference messages

You might have noticed that the size of a newly opened view is rather arbitrary. However, before opening the view, the framework sends the view a message with a proposed size. The view may adapt this proposal to its own needs. To do that, the message of type *Properties.SizePref* must be answered in the view's *HandlePropMsg* procedure. Before a view is displayed for the first time, the proposed size for the width and the height of the view is *Views.undefined.* The following version of our sample view draws a rectangle with a width of 2 cm and a height of 1 cm. Changes compared to the previous version are written in bold face.

```
MODULE ObxViews1;

    IMPORT Views, Ports, Properties;

    TYPE View = POINTER TO RECORD (Views.View) END;

    PROCEDURE (v: View)  Restore (f: Views.Frame; l, t, r, b: INTEGER);
    BEGIN
        f.DrawRect(l, t, r, b, Ports.fill, Ports.red)
    END Restore;

    PROCEDURE (v: View) HandlePropMsg (VAR msg: Properties.Message);
    BEGIN
        WITH msg: Properties.SizePref DO
            IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
                msg.w := 20 * Ports.mm; msg.h := 10 * Ports.mm
            END
        ELSE (* ignore other messages *)
        END
    END HandlePropMsg;

    PROCEDURE Deposit*;
        VAR v: View;
    BEGIN
        NEW(v); Views.Deposit(v)
    END Deposit;
END ObxViews1.
```

❗ "ObxViews1.Deposit; StdCmds.Open"

Every newly generated view now assumes the desired predefined size. This is only possible because the view and its container cooperate, in this case via the *Properties.SizePref* message. A container is expected to send this message whenever it wants to change a view's size, and then adhere to the returned data. However, for the understanding of the BlackBox Component Framework it is essential to know that while a well-behaved container *should* follow this protocol, it is not *required* to do so. That's why such a message is called a *preference*. It describes merely a view's preference, so that the surrounding container can make allowances for the special needs of the view. But it is always the container which has the last word in such negotiations. For example, the container will not let embedded views become larger than itself. The standard document, text, and form containers are well-behaved in that they support all preferences defined in module *Properties*. Special containers, e.g. texts, may define additional preferences specific to their type of contents. It is important to note that a view can ignore all preferences it doesn't know or doesn't care about.

The *Properties.SizePref* allows us to restrict the possible values for the width and the height of a view. For example, we can enforce a minimal and a maximal size, or fix the height or width of the view, or specify a constraint between its width and height. The procedures *Properties.ProportionalConstraint* and *Properties.GridConstraint* are useful standard implementations to specify constraints. The next version of our view implementation specifies that the rectangle is always twice as wide as it is high. In addition, minimal and maximal values for the height (and thus also for the width) are specified. If the view is resized using the resize handles, the constraints are not violated. Try it out!

```
MODULE ObxViews2;

    IMPORT Views, Ports, Properties;

    TYPE View = POINTER TO RECORD (Views.View) END;

    PROCEDURE (v: View)  Restore (f: Views.Frame; l, t, r, b: INTEGER);
    BEGIN
        f.DrawRect(l, t, r, b, Ports.fill, Ports.red)
    END Restore;

    PROCEDURE (v: View) HandlePropMsg (VAR msg: Properties.Message);
        CONST min = 5 * Ports.mm; max = 50 * Ports.mm;
    BEGIN
        WITH msg: Properties.SizePref DO
            IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
                 msg.w := 20 * Ports.mm; msg.h := 10 * Ports.mm
            ELSE
                Properties.ProportionalConstraint(2, 1, msg.fixedW, msg.fixedH,
                                                                msg.w, msg.h);
                IF msg.h < min THEN
                    msg.h := min; msg.w := 2 * min
                ELSIF msg.h > max THEN
                    msg.h := max; msg.w := 2 * max
                END
            END
        ELSE (* ignore other messages *)
        END
    END HandlePropMsg;

    PROCEDURE Deposit*;
        VAR v: View;
    BEGIN
        NEW(v); Views.Deposit(v)
    END Deposit;

END ObxViews2.
```

❗ "ObxViews2.Deposit; StdCmds.Open"

We will look at two other preferences in this tutorial. The first one is *Properties.ResizePref.*

```
TYPE
    ResizePref = RECORD (Properties.Preference)
        fixed,
        horFitToPage, verFitToPage,
        horFitToWin, verFitToWin: BOOLEAN    (* OUT *)
    END;
```

The receiver of this message may indicate that it doesn't wish to be resized, by setting *fixed* to *TRUE*. As a consequence, the view will not display resize handles when it is selected, i.e., it cannot be resized interactively. However, the *Properties.SizePref* message is still sent to the view, as the initial size still needs to be determined.

If a view is a *root view*, i.e., the outermost view in a document or window (e.g., if opened with *StdCmds.Open*), then the size of the window, the size of the view, or both might be changed.

However, sometimes it is convenient if the view size is automatically adapted whenever the window is resized, e.g., for help texts which should use as much screen estate as possible. For other views, it may be preferable that their size is not determined by the window, but rather by their contents, or by the page setup of the document in which they are embedded. A view can indicate such preferences by setting the *horFitToWin*, *verFitToWin*, *horFitToPage*, and *verFitToPage* flags in the *Properties.SizePref* message. These flags have no effect if the view is not a root view, i.e., if it is embedded deeper inside a document.

An automatic adaptation of the view size to the actual window size can be achieved by setting the fields *horFitToWin* and *verFitToWin*. The size of the view is then bound to the window, i.e., the user can change it directly by resizing the window.

Note that if the size of the view is bound to the size of the window (either by setting *horFitToWin* or *verFitToWin*), then no *Properties.SizePref* messages are sent to the view, i.e., the constraints specified through *Properties.SizePref* are no longer enforced. Additionally, the view does not display resize handles, regardless of the *fixed* flag.

By setting the fields *horFitToPage* or *verFitToPage*, the width or the height of the view can be bound to the actual printer page size. The width of a text view is usually bound to the width of the page size, and can be changed via the page setup mechanism of the underlying operating system. The following example enforces that the size of a root view is bound to the size of the window:

```
MODULE ObxViews3;

    IMPORT Views, Ports, Properties;
    TYPE View = POINTER TO RECORD (Views.View) END;

    PROCEDURE (v: View)  Restore (f: Views.Frame; l, t, r, b: INTEGER);
    BEGIN
        f.DrawRect(l, t, r, b, Ports.fill, Ports.red)
    END Restore;

    PROCEDURE (v: View) HandlePropMsg (VAR msg: Properties.Message);
        CONST min = 5 * Ports.mm; max = 50 * Ports.mm;
    BEGIN
        WITH msg: Properties.SizePref DO
            IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
                 msg.w := 20 * Ports.mm; msg.h := 10 * Ports.mm
            ELSE
                Properties.ProportionalConstraint(2, 1, msg.fixedW, msg.fixedH,
                                                                msg.w, msg.h);
                IF msg.h < min THEN
                    msg.h := min; msg.w := 2 * min
                ELSIF msg.h > max THEN
                    msg.h := max; msg.w := 2 * max
                END
            END
        | msg: Properties.ResizePref DO
            msg.horFitToWin := TRUE; msg.verFitToWin := TRUE
        ELSE (* ignore other messages *)
        END
    END HandlePropMsg;

    PROCEDURE Deposit*;
        VAR v: View;
    BEGIN
        NEW(v); Views.Deposit(v)
    END Deposit;

END ObxViews3.
```

❗ "ObxViews3.Deposit; StdCmds.Open"

The next preference we look at is *Properties.FocusPref*.

```
TYPE
    FocusPref = RECORD (Properties.Preference)
        atLocation: BOOLEAN;   (* IN *)
        x, y: INTEGER;   (* IN *)
        hotFocus, setFocus, selectOnFocus: BOOLEAN   (* OUT *)
    END;
```

When an attempt is made to activate an embedded view, by clicking in it, then the *Properties.FocusPref* message is sent to the view. If this message is not answered the view is selected as a whole (a so-called *singleton*). This is the behavior of the views implemented in the examples above. To see this, place the caret on the next line and click on the commander below, then on the pasted view:

❗ "ObxViews3.Deposit; StdCmds.PasteView" >< *set the caret here before executing the command*

This behavior is adequate if a view is passive. However, if the view contains editable contents, or if there are menu commands that operate on the view, the user should be able to *focus* the view. The focus is where keyboard input is sent to, where the current selection or caret are displayed (if there are any such marks), and where upon some menu commands operate. In fact, menus can be made to appear whenever a view is focused, and disappear as soon as the view loses focus (more on this below). A root view is always focused when its document window is focus (i.e., is the top window).

If an embedded view wants to become focus, it must answer the *Properties.FocusPref* message. The view can choose whether it wants to become focus permanently or if it wants only be focus as long as the mouse button is pressed. The latter is called a *hot focus*. A typical example of a hot focus is a command button. If a view wants to be a hot focus, it must set the flag *hotFocus*. The focus is then released immediately after the mouse is released. If a view wants to become focus permanently, then the flag *setFocus* must be set instead. *setFocus* should be set for all genuine editors, such that context-sensitive menu commands can be attached to the view. In addition to the *setFocus* flag, a view may set the *selectOnFocus* flag to indicate that upon focusing by keyboard, the view's contents should be selected. Text entry fields are prime examples for this behavior: the contents of a newly focused text entry field is selected, if the user focused it not by clicking in it, but by using the tabulator key.

If the user clicks in an unfocused view, then the *atLocation* flag is set by the framework before sending the message. The receiving view may decide whether to become focused depending on where the user clicked. The mouse position is passed to the view in the focus preference's *x* and *y* fields. Text rulers are examples of views which become focused depending on the mouse location. If you click in the icons or in the area below the scale of a ruler, the ruler is not focused. Otherwise it is focused. Try this out with the ruler below. If you don't see a ruler, execute the *Show Marks* command in the *Text* menu.

A view is not necessarily focused through a mouse click. In forms for example, the views can be selected using the tabulator key. There, the views do not become focused through a mouse click, and the *atLocation* field is accordingly set to *FALSE* by the framework.

The next version of our example will answer the *FocusPref* message and set the *setFocus* flag. This is done by adding the following statements to the previous example's *HandlePropMsg* procedure:

```
| msg: Properties.FocusPref DO
    msg.setFocus := TRUE
```

If an embedded view is focused, the frame of the view is marked with a suitable focus border mark, and view-specific menus may appear while others disappear.

6.4 Controller messages

Other messages that may be interpreted by *HandlePropMsg* will be discussed later. Next we will see how a view can react on user input, e.g., on mouse clicks in the view or on commands called through a menu. For simple views this behavior is implemented in the procedure *Views.HandleCtrlMsg*. It is also possible to define a separate object, a so-called controller, that implements the interactive behavior of a view. Programming of controller objects is not described in this document, since controllers are only recommended for the most complex views.

The*Views.HandleCtrlMsg* handler answers the *controller messages* sent to the view. A controller message is a message that is sent along exactly one path in a view hierarchy, the *focus path*. Every view on the focus path decides for itself whether it is the terminal of this path, i.e., whether it is the current focus, or whether the message should be forwarded to one of its embedded views. It is important to note that all controller messages which are not relevant for a particular view type can simply be ignored.

In order to be able to perform edit operations on the focus view, the framework must allow to somehow perform these operations. Mouse clicks and key strokes can always be issued. However, how the menus should look like may be decided by the view itself. For that purpose, the message *Controllers.PollOpsMsg* is sent to the view. Depending on its current state (e.g., its current selection) and depending on the contents of the clipboard, the focus view can inform the framework of which editing operations it currently supports. The valid operations are elements of the {Controllers.cut, Controllers.copy, Controllers.paste, Controllers.pasteView} set.

```
  TYPE
    PollOpsMsg = RECORD (Controllers.Message)
      type: Stores.TypeName;   (* OUT *)
      pasteType: Stores.TypeName;   (* IN *)
      singleton: Views.View;   (* OUT *)
      selectable: BOOLEAN;   (* OUT *)
      valid: SET   (* OUT *)
    END;
```

The set of valid edit operations is returned in the *valid* field, where *valid IN {Controllers.cut, Controllers.copy, Controllers.paste*}. According to the set of valid operations, the corresponding menu entries in the *Edit* menu are enabled or disabled, e.g., *Cut, Copy, Paste* and *Paste Object...* (Windows) / *Paste as Part* (Mac OS). The field *pasteType* contains the concrete type of the view from which a copy would be pasted, if a paste operation occurred. Depending on this field, the view could decide whether it wants to support the paste operation or not. *PollOpsMsg* is sent when the user clicks in the menu bar. Its sole purpose is to enable or disable menu items, i.e., to provide user feedback.

If a view supports a selection of its contents, then *selectable* must be set to *TRUE*. As a consequence, the menu entry *Select All* will be enabled. The flag should be set regardless of whether a selection currently exists or not.

In the *type* field a type name may be passed. This name denotes a context for the focus view. This context is used to determine which menus are relevant for the focus view. As a convention, a view assigns the type name of its interface base type to *type*, e.g. "ObxViews4.View". A menu which indicates to be active on "ObxViews4.View" will be displayed if such a view is focus. The type name is used because it is easy to make unique, so that name collisions are avoided. The framework doesn't interpret the name.

The *singleton* field is only meaningful for container views. It denotes a container's currently selected view, if the selection consists of exactly one embedded view.

The following example view can become focus, supports the paste operation, and informs the framework that its contents is selectable. As a consequence, the menu entries *Paste* and *Select All* in the *Edit* menu are enabled. Additionally, it defines the context "Obx.Tutorial". Therefore the following menu appears whenever the view is focused; provided that the menu has been installed.

Note that the name of the context, in this case "Obx.Tutorial", by convention starts with the subsystem or a module name followed by a dot, in this case "Obx.". This is highly recommended, in order to make context names globally unique. Often, the name is simply the name of a view type, e.g., "TextViews.View".

A menu can either be defined in the global menu text System/Rsrc/Menus, or more appropriately in its own subsystem's menu text, in this case in Obx/Rsrc/Menus . The global menu text then only needs an *INCLUDE "Obx"* statement to make the Obx menus known.

```
    MENU "New" ("Obx.Tutorial")
      "Beep"    ""    "Dialog.Beep"    ""
    END
```

```
MODULE ObxViews4;

    IMPORT Views, Ports, Properties, Controllers;

    TYPE View = POINTER TO RECORD (Views.View) END;

    PROCEDURE (v: View)  Restore (f: Views.Frame; l, t, r, b: INTEGER);
    BEGIN
        f.DrawRect(l, t, r, b, Ports.fill, Ports.red)
    END Restore;

    PROCEDURE (v: View) HandlePropMsg (VAR msg: Properties.Message);
        CONST min = 5 * Ports.mm; max = 50 * Ports.mm;
    BEGIN
        WITH msg: Properties.SizePref DO
            IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
                 msg.w := 20 * Ports.mm; msg.h := 10 * Ports.mm
            ELSE
                Properties.ProportionalConstraint(2, 1, msg.fixedW, msg.fixedH,
                                                                    msg.w, msg.h);
                IF msg.h < min THEN
                    msg.h := min; msg.w := 2 * min
                ELSIF msg.h > max THEN
                    msg.h := max; msg.w := 2 * max
                END
            END
        | msg: Properties.ResizePref DO
            msg.horFitToWin := TRUE; msg.verFitToWin := TRUE
        | msg: Properties.FocusPref DO
            msg.setFocus := TRUE
        ELSE (* ignore other messages *)
        END
    END HandlePropMsg;

    PROCEDURE (v: View) HandleCtrlMsg (f: Views.Frame;
                                    VAR msg: Controllers.Message; VAR focus: Views.View);
    BEGIN
        WITH msg: Controllers.PollOpsMsg DO
            msg.valid := {Controllers.paste}; msg.selectable := TRUE;
            msg.type := "Obx.Tutorial"
        ELSE (* ignore other messages *)
        END
    END HandleCtrlMsg;

    PROCEDURE Deposit*;
        VAR v: View;
    BEGIN
        NEW(v); Views.Deposit(v)
    END Deposit;

END ObxViews4.
```

❗ "ObxViews4.Deposit; StdCmds.PasteView" >< *set the caret here before executing the command*

Next we discuss how a view can react on actual edit messages. In particular, these are the *Controllers.EditMsg* for edit operations such as cut, copy, or paste, and the *Controllers.TrackMsg* for mouse clicks.

Whenever a key is pressed in a view or when a cut, copy or paste operation is invoked, a *Controllers.EditMsg* is sent to the focus view. The cut, copy and paste operations can only be generated through the environment (menu) if they have been announced with the *Controllers.PollOpsMsg*.

```
TYPE
   EditMsg = RECORD (Controllers.RequestMessage)
      op: INTEGER;   (* IN *)
      modifiers: SET;   (* IN *)
      char: CHAR;   (* IN *)
      view: Views.View;   (* IN for paste, OUT for cut and copy *)
      w, h: INTEGER;   (* IN for paste, OUT for cut and copy *)
      isSingle: BOOLEAN;   (* IN for paste, OUT for cut and copy *)
      clipboard: BOOLEAN   (* IN *)
   END;
```

The *op* field specifies which kind of operation has to be performed. If *op = Controllers.cut* then a copy of the focus view has to be generated. The contents of the new view is a copy of the focus view's selection. The new view is assigned to the *view* field. In addition, the selection is deleted in the focus view.
There is one special case: if the selection consists of exactly one view (a singleton), then a copy of the singleton should be copied to the *view* field, not a copy of the singleton's container. In this case, *isSingle* must be set to *TRUE*.

Except for the deletion of the selection, the same operations have to be performed if *op = Controllers.copy.*

If a key is pressed, then the *op* field has the value *Controllers.pasteChar.* The character to be pasted is stored in the *char* field. The *modifiers* set indicates whether a modifier or control key has been pressed. In the latter case, the *char* field has to be interpreted as a control character.

The paste operation is the inverse of the copy operation: a copy of the contents of the view stored in the *view* field has to be copied into the focus view's contents. This operation is indicated by *op = Controllers.paste*. The view must know the type of the view whose contents is to be pasted and must decide how to insert the pasted view's contents into its own view. For example, a text field view should support copying from a text view only.
If *isSingle* is *TRUE*, or if the contents of *view* cannot be pasted because it has an unknown or incompatible type, a copy of *view* must be pasted. Of course, this is only possible in general containers, which allow the embedding of other views.
When pasting a complete view, the desired width and height of the pasted view are given in the fields *w* and *h*. These values can be treated as hints by the receiving view. If they are not suitable, others can be used.

Whenever the mouse button is pressed in the focus view, a *Controllers.TrackMsg* is sent to the view. The coordinates are specified in the *x* and *y* fields (of the base type *Controllers.CursorMessage).* The *modifiers* set indicates whether modifier keys have been pressed, or whether a double click has been performed. *modifiers IN {Controllers.doubleClick, Controllers.extend, Controllers.modify}*. If the view wants to show a feedback while tracking the mouse, it needs to poll the mouse position in a loop, by calling the *Input* procedure of the passed frame. *Input* also returns information on whether the mouse has been released. Any feedback should directly be drawn into the frame in which the mouse button was pressed.

After the feedback loop, a possible modification of the view's contents need to be broadcast to all the frames that display the same view. Remember that the same document, and consequently all its embedded views, may be displayed in several windows (see Chapter 2). If an embedded view is visible in three windows, there exist three frames displaying the same view. Procedure *Views.Update* causes a restore of the view once in each of its visible frames.

As an example, we extend our view with a cross-hair marker that can be moved around. The coordinates of this marker are stored as additional fields in the view object. Note that if a view contains instance variables, it should implement the *CopyFromSimpleView, Internalize* and *Externalize* procedures in order to work properly. We refer to the next section of this view tutorial, which describes the purpose of these messages. In our example view, the marker can be moved around with the mouse, or through the cursor keys. When a cursor key is pressed, the focus view is informed by sending it an *EditMsg* with *op = pasteChar*.

From time to time a *Controllers.PollCursorMsg* is sent to the focus view. The view may set the form of the cursor depending on the coordinates of the mouse. The *cursor* field can be assigned a cursor out of {*Ports.arrowCursor, Ports.textCursor, Ports.graphicsCursor, Ports.tableCursor, Ports.bitmapCursor*}. The particular form of the cursor depends on the underlying operating system. In our example view, we set the cursor to a graphics cursor.

```
MODULE ObxViews5;

    IMPORT Views, Ports, Properties, Controllers;

    TYPE
        View = POINTER TO RECORD (Views.View)
            x, y: INTEGER
        END;

    PROCEDURE (v: View)  Restore (f: Views.Frame; l, t, r, b: INTEGER);
    BEGIN
        f.DrawRect(l, t, r, b, Ports.fill, Ports.red);
        f.DrawLine(v.x, t, v.x, b, 0, Ports.white); f.DrawLine(l, v.y, r, v.y,
                                                                        0, Ports.white)
    END Restore;

    PROCEDURE (v: View) HandlePropMsg (VAR msg: Properties.Message);
        CONST min = 5 * Ports.mm; max = 50 * Ports.mm;
    BEGIN
        WITH msg: Properties.SizePref DO
            IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
                 msg.w := 20 * Ports.mm; msg.h := 10 * Ports.mm
            ELSE
                Properties.ProportionalConstraint(2, 1, msg.fixedW, msg.fixedH,
                                                                msg.w, msg.h);
                IF msg.h < min THEN msg.h := min; msg.w := 2 * min
                ELSIF msg.h > max THEN msg.h := max; msg.w := 2 * max
                END
            END
        | msg: Properties.ResizePref DO
            msg.horFitToWin := TRUE; msg.verFitToWin := TRUE
        | msg: Properties.FocusPref DO
            msg.setFocus := TRUE
        ELSE (* ignore other messages *)
        END
    END HandlePropMsg;

    PROCEDURE (v: View) HandleCtrlMsg (f: Views.Frame;
                                                            VAR msg: Controllers.Message;
                                                            VAR focus: Views.View);
        VAR x, y, w, h: INTEGER; m: SET; isDown: BOOLEAN;
    BEGIN
        WITH msg: Controllers.PollOpsMsg DO
            msg.valid := {Controllers.paste}; msg.selectable := TRUE;
            msg.type := "Obx.Tutorial"
        | msg: Controllers.EditMsg DO
            IF msg.op = Controllers.pasteChar THEN (* cursor keys *)
                IF msg.char = 1DX THEN INC(v.x, Ports.mm)
                ELSIF msg.char = 1CX THEN DEC(v.x, Ports.mm)
                ELSIF msg.char  = 1EX THEN DEC(v.y, Ports.mm)
                ELSIF msg.char  = 1FX THEN INC(v.y, Ports.mm)
                END;
                Views.Update(v, Views.keepFrames)
            END
        | msg: Controllers.TrackMsg DO
            v.x := msg.x; v.y := msg.y;
            v.context.GetSize(w, h); v.Restore(f, 0, 0, w, h);
            REPEAT
                f.Input(x, y, m, isDown);
                IF (x # v.x) OR (y # v.y) THEN
                    v.x := x; v.y := y; v.Restore(f, 0, 0, w, h)
                END
            UNTIL ~isDown;
            Views.Update(v, Views.keepFrames)
        | msg: Controllers.PollCursorMsg DO
            msg.cursor := Ports.graphicsCursor
        ELSE (* ignore other messages *)
        END
    END HandleCtrlMsg;

    PROCEDURE Deposit*;
        VAR v: View;
    BEGIN
        NEW(v); v.x := 0; v.y := 0; Views.Deposit(v)
    END Deposit;

END ObxViews5.
```

❗ "ObxViews5.Deposit; StdCmds.Open"

There exist many other controller messages which may be sent to a view, but which are beyond the scope of this tutorial. There are messages for the generic scrolling mechanism of the BlackBox Component Framework (*Controllers.PollSectionMsg, Controllers.ScrollMsg, Controllers.PageMsg*), messages which implement drag & drop (*Controllers.PollDropMsg, Controllers.DropMsg, Controllers.TransferMessage*) and messages which control the selection (*Controllers.SelectMsg*). For these and other controller messages we refer to the documentation of the *Controllers* module.

6.5 Properties

We want to close this section with a final extension of our view: the color of the view should be changeable through the *Attributes* menu. The general mechanism to get and set attributes of a view from its environment are *properties.* A view may know about attributes such as font, color, size, but it may also know about arbitrary other attributes. Properties are set with the *Properties.SetMsg* and inspected with the *Properties.PollMsg*. Properties in one of these messages are stored in a linked list. If a view gets a *Properties.SetMsg* it should traverse its property list and adjust those properties it knows about. A *Properties.GetMsg* is sent to the view to get its properties. The view should return all properties it knows about. Properties can be inserted into a message's property list with the *Properties.Insert* procedure.

Every property describes up to 32 attributes. The *known* set defines which attributes are known to the view. The view may also specify which attributes are read-only in the *readOnly* set. The *valid* set finally defines which attributes currently have a defined value. For example, if in a text several characters with different sizes are selected, then the attribute size is known to the view, but currently does not have a valid value. The selection is not homogeneous, and thus there is no single valid value.

```
TYPE
   Property = POINTER TO ABSTRACT RECORD
      next-: Properties.Property;
      known, readOnly, valid: SET;    (* valid and readOnly are subsets of known *)
      (p: Property) IntersectWith (q: Properties.Property; OUT equal: BOOLEAN), NEW, ABSTRACT
   END;
```

A special property is *Properties.StdProp.* This property encompasses font attributes as well as color, and it is known to the BlackBox environment. The supported attributes are *Properties.color, Properties.typeface, Properties.size, Properties.style, Properites.weight.* The fields in the property record hold the corresponding values.

```
TYPE
   StdProp = POINTER TO RECORD (Properties.Property)
      color: Dialog.Color;
      typeface: Fonts.Typeface;
      size: INTEGER;
      style: RECORD
         val, mask: SET
      END;
      weight: INTEGER
   END;
```

The last version of our view only support the color attribute of the standard property. If it receives a *Properties.PollMsg* message, it returns a *Properties.StdProp* object where only the color field is set, and where only the color attribute is defined as known and valid. On a *Properties.SetMsg*, the property list must be searched for a standard property whose color field is valid. When the color has been changed, the view must be updated in all its frames.

```
MODULE ObxViews6;

    IMPORT Views, Ports, Properties, Controllers;

    TYPE
        View = POINTER TO RECORD (Views.View)
            x, y: INTEGER;
            c: Ports.Color
        END;

    PROCEDURE (v: View)  Restore (f: Views.Frame; l, t, r, b: INTEGER);
    BEGIN
        f.DrawRect(l, t, r, b, Ports.fill, v.c);
        f.DrawLine(v.x, t, v.x, b, 0, Ports.white); f.DrawLine(l, v.y, r, v.y, 0, Ports.white)
    END Restore;

    PROCEDURE (v: View) HandlePropMsg (VAR msg: Properties.Message);
        CONST min = 5 * Ports.mm; max = 50 * Ports.mm;
        VAR stdProp: Properties.StdProp; prop: Properties.Property;
    BEGIN
        WITH msg: Properties.SizePref DO
            IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
                 msg.w := 20 * Ports.mm; msg.h := 10 * Ports.mm
            ELSE
                Properties.ProportionalConstraint(2, 1, msg.fixedW, msg.fixedH,
                                                                    msg.w, msg.h);
                IF msg.h < min THEN msg.h := min; msg.w := 2 * min
                ELSIF msg.h > max THEN msg.h := max; msg.w := 2 * max
                END
            END
        | msg: Properties.ResizePref DO
            msg.horFitToWin := TRUE; msg.verFitToWin := TRUE
        | msg: Properties.FocusPref DO
            msg.setFocus := TRUE
        | msg: Properties.PollMsg DO
            NEW(stdProp);
            stdProp.color.val := v.c;
            stdProp.valid := {Properties.color};
            stdProp.known := {Properties.color};
            Properties.Insert(msg.prop, stdProp)
        | msg: Properties.SetMsg DO
            prop := msg.prop;
            WHILE prop # NIL DO
                WITH prop: Properties.StdProp DO
                    IF Properties.color IN prop.valid THEN v.c := prop.color.val END
                ELSE
                END;
                prop := prop.next
            END;
            Views.Update(v, Views.keepFrames)
        ELSE (* ignore other messages *)
        END
```

```
  END HandlePropMsg;

  PROCEDURE (v: View) HandleCtrlMsg (f: Views.Frame; VAR msg: Controllers.Message;
                                        VAR focus: Views.View);
    VAR x, y, w, h: INTEGER; m: SET; isDown: BOOLEAN;
  BEGIN
    WITH msg: Controllers.PollOpsMsg DO
      msg.valid := {Controllers.paste}; msg.selectable := TRUE;
      msg.type := "Obx.Tutorial"
    | msg: Controllers.EditMsg DO
      IF msg.op = Controllers.pasteChar THEN
        IF msg.char = 1DX THEN INC(v.x, Ports.mm)
        ELSIF msg.char = 1CX THEN DEC(v.x, Ports.mm)
        ELSIF msg.char  = 1EX THEN DEC(v.y, Ports.mm)
        ELSIF msg.char  = 1FX THEN INC(v.y, Ports.mm)
        END;
        Views.Update(v, Views.keepFrames)
      END
    | msg: Controllers.TrackMsg DO
      v.x := msg.x; v.y := msg.y; v.context.GetSize(w, h); v.Restore(f, 0, 0, w, h);
      REPEAT
        f.Input(x, y, m, isDown);
        IF (x # v.x) OR (y # v.y) THEN
          v.x := x; v.y := y; v.Restore(f, 0, 0, w, h)
        END
      UNTIL ~isDown;
      Views.Update(v, Views.keepFrames)
    | msg: Controllers.PollCursorMsg DO
      msg.cursor := Ports.graphicsCursor
    ELSE (* ignore other messages *)
    END
  END HandleCtrlMsg;

  PROCEDURE Deposit*;
    VAR v: View;
  BEGIN
    NEW(v); v.x := 0; v.y := 0; v.c := Ports.black; Views.Deposit(v)
  END Deposit;

END ObxViews6.
```

❗ "ObxViews6.Deposit; StdCmds.PasteView"   >< *set the caret here before executing the command*

6.6 Model-View separation

In this section, we present a sequence of five increasingly more versatile versions of a view object which displays a string that may be typed in by the user. This string represents the view-specific data which it displays. We will see that multi-view editing is possible if this string is represented by a separate model object.

Let us start with a first version of this view, where the string is stored in the view itself. Every view displays its own string. A string's length is limited to 255 characters, but no error handling is performed in the following demonstration programs.

```
MODULE ObxViews10;

    IMPORT Fonts, Ports, Views, Controllers, Properties;

    CONST d = 20 * Ports.point;

    TYPE
        View = POINTER TO RECORD (Views.View)
            i: INTEGER;   (* position of next free slot in string *)
            s: ARRAY 256 OF CHAR (* string *)
        END;

    PROCEDURE (v: View) Restore (f: Views.Frame; l, t, r, b: INTEGER);
    BEGIN
        f.DrawString(d, d, Ports.black, v.s, Fonts.dir.Default())
    END Restore;

    PROCEDURE (v: View) HandleCtrlMsg (f: Views.Frame;
                                                VAR msg: Controllers.Message;
                                                VAR focus: Views.View);
    BEGIN
        WITH msg: Controllers.EditMsg DO
            IF msg.op = Controllers.pasteChar THEN (* accept typing *)
                v.s[v.i] := msg.char; INC(v.i); v.s[v.i] := 0X; (* append character to string *)
                Views.Update(v, Views.keepFrames) (* restore v in any frame that displays it *)
            END
        ELSE            (* ignore other messages *)
        END
    END HandleCtrlMsg;

    PROCEDURE (v:View) HandlePropMsg (VAR msg: Properties.Message);
    BEGIN
        WITH msg: Properties.SizePref DO
            IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
                msg.w := 10 * d; msg.h := 2 * d
            END
        | msg: Properties.FocusPref DO
            msg.setFocus := TRUE
        ELSE
        END
    END HandlePropMsg;

    PROCEDURE Deposit*;
        VAR v: View;
    BEGIN
        NEW(v); v.s := ""; v.i := 0;
        Views.Deposit(v)
    END Deposit;

END ObxViews10.
```

! "ObxViews10.Deposit; StdCmds.Open"

As described in the last section, a character typed in is sent to the view in the form of a controller message record (*Controllers.Message*), to be handled by the view's *HandleCtrlMsg* procedure. This procedure is called with a *Controllers.EditMsg* as actual parameter when a character was typed in, and it reacts by inserting the character contained in the message into its field *s*. Afterwards, it causes the view to be restored; i.e., wherever the view is visible on the display it is redrawn in its new state, displaying the string that has become one character longer.

In the above example, a view contains a variable state (the view's string field *s*). This state should be saved when the window's contents are saved to disk. For this purpose, a view provides two procedures, called *Internalize* and *Externalize*, whose uses are shown in the next iteration of our example program:

```
MODULE ObxViews11;
(* Same as ObxViews10, but the view's string can be stored and copied *)

    IMPORT Fonts, Ports, Stores, Views, Controllers, Properties;

    CONST d = 20 * Ports.point;

    TYPE
        View = POINTER TO RECORD (Views.View)
            i: INTEGER;     (* position of next free slot in string *)
            s: ARRAY 256 OF CHAR (* string *)
        END;

    PROCEDURE (v: View) Internalize (VAR rd: Stores.Reader);
        VAR version: INTEGER;
    BEGIN
        rd.ReadVersion(0, 0, version);
        IF ~rd.cancelled THEN
            rd.ReadInt(v.i); rd.ReadString(v.s)
        END
    END Internalize;

    PROCEDURE (v: View) Externalize (VAR wr: Stores.Writer);
    BEGIN
        wr.WriteVersion(0);
        wr.WriteInt(v.i); wr.WriteString(v.s)
    END Externalize;

    PROCEDURE (v: View) CopyFromSimpleView (source: Views.View);
    BEGIN
        WITH source: View DO
            v.i := source.i; v.s := source.s
        END
    END CopyFromSimpleView;
    PROCEDURE (v: View) Restore (f: Views.Frame; l, t, r, b: INTEGER);
    BEGIN
        f.DrawString(d, d, Ports.black, v.s, Fonts.dir.Default())
    END Restore;

    PROCEDURE (v: View) HandleCtrlMsg (f: Views.Frame;
                                                                                VAR msg: Controllers.Message;
                                                                                VAR focus: Views.View);
    BEGIN
        WITH msg: Controllers.EditMsg DO
            IF msg.op = Controllers.pasteChar THEN (* accept typing *)
                v.s[v.i] := msg.char; INC(v.i); v.s[v.i] := 0X; (* append character to string *)
                Views.SetDirty(v); (* mark view's document as dirty *)
                Views.Update(v, Views.keepFrames) (* restore v in any frame that displays it *)
            END
        ELSE                        (* ignore other messages *)
        END
    END HandleCtrlMsg;

    PROCEDURE (v:View) HandlePropMsg (VAR msg: Properties.Message);
    BEGIN
        WITH msg: Properties.SizePref DO
            IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
                msg.w := 10 * d; msg.h := 2 * d
            END
        | msg: Properties.FocusPref DO
            msg.setFocus := TRUE
        ELSE
        END
    END HandlePropMsg;

    PROCEDURE Deposit*;
        VAR v: View;
    BEGIN
        NEW(v); v.s := ""; v.i := 0;
        Views.Deposit(v)
    END Deposit;

END ObxViews11.
```

A few comments about *View.Internalize* and *View.Externalize* are in order here: First, the two procedures have a reader, respectively a writer, as variable parameters. These file mappers are set up by BlackBox itself; a view simply uses them.

Second, *View.Internalize* must read exactly the same (amount of) data that *View.Externalize* has written.

Third, a user interface typically defines some visual distinction for documents that contain modified views, i.e., for "dirty" documents. Or it may require that when the user closes a dirty document, he or she should be asked whether to save it or not. In order to make this possible, a view must tell when its contents has been modified. This is done by the *Views.BeginModification* / *Views.EndModification* calls.

In addition to *View.Internalize* and *View.Externalize*, the above view implementation also implements a *CopyFromSimpleView* procedure. Such a procedure should copy the view's contents, given a source view of the same type. *CopyFromSimpleView* should usually have the same effect as if the source's contents were externalized on a temporary file, and then internalized again by the destination view. There can be exceptions, though. For example, a text copies even temporary embedded views such as error markers, but it doesn't externalize them.

A view that contains (mutable) state should implement *CopyFromSimpleView*, so that it can be printed. The reason that this is necessary is that the framework makes a shallow copy of a view that is being printed, in order to avoid the original view to be changed by pagination, scrolling, or similar modifications that may be performed during printing.

Basically, *ObxViews11* has shown most of what is involved in implementing a simple view class. Such simple views are often sufficient; thus it is important that they are easy to implement. However, there are cases where a more advanced view design is in order. In particular, if a window normally can only display a small part of a view's contents, *multi-view editing* should be supported.

Multi-view editing means that there may be several views showing the same data. The typical application of this feature is to have two or more windows displaying the same document, each of these windows showing a different part of it in its own view. Thus, if a view's data has been changed, it and all the other affected views must be notified of the change, such that they can update the display accordingly.

The following sample program, which is the same as the previous one except that it supports multi-view editing, is roughly twice as long. This indicates that the design and implementation of such views is quite a bit more involved than that of simple views. It is a major design decision whether this additional complexity is warranted by its increased convenience.

```
MODULE ObxViews12;
(* Same as ObxViews11, but uses a separate model for the string *)

    IMPORT Fonts, Ports, Stores, Models, Views, Controllers, Properties;

    CONST d = 20 * Ports.point;

    TYPE
        Model = POINTER TO RECORD (Models.Model)
            i: INTEGER;  (* position of next free slot in string *)
            s: ARRAY 256 OF CHAR (* string *)
        END;

        View = POINTER TO RECORD (Views.View)
            model: Model
        END;


    (* Model *)

    PROCEDURE (m: Model) Internalize (VAR rd: Stores.Reader);
        VAR version: INTEGER;
    BEGIN
        rd.ReadVersion(0, 0, version);
        IF ~rd.cancelled THEN
            rd.ReadInt(m.i); rd.ReadString(m.s)
        END
    END Internalize;

    PROCEDURE (m: Model) Externalize (VAR wr: Stores.Writer);
    BEGIN
        wr.WriteVersion(0);
        wr.WriteInt(m.i); wr.WriteString(m.s)
    END Externalize;

    PROCEDURE (m: Model) CopyFrom (source: Stores.Store);
    BEGIN
        WITH source: Model DO
            m.i := source.i; m.s := source.s
        END
    END CopyFrom;


    (* View *)

    PROCEDURE (v: View) Internalize (VAR rd: Stores.Reader);
        VAR version: INTEGER; st: Stores.Store;
    BEGIN
        rd.ReadVersion(0, 0, version);
        IF ~rd.cancelled THEN
            rd.ReadStore(st);
            v.model := st(Model)
        END
    END Internalize;

    PROCEDURE (v: View) Externalize (VAR wr: Stores.Writer);
    BEGIN
        wr.WriteVersion(0);
        wr.WriteStore(v.model)
    END Externalize;

    PROCEDURE (v: View) CopyFromModelView (source: Views.View;
                                                        model: Models.Model);
    BEGIN
        v.model := model(Model)
    END CopyFromModelView;

    PROCEDURE (v: View) Restore (f: Views.Frame; l, t, r, b: INTEGER);
    BEGIN
        f.DrawString(d, d, Ports.black, v.model.s, Fonts.dir.Default())
    END Restore;

    PROCEDURE (v: View) ThisModel (): Models.Model;
    BEGIN
        RETURN v.model
    END ThisModel;
```

```
    PROCEDURE (v: View) HandleModelMsg (VAR msg: Models.Message);
    BEGIN
        Views.Update(v, Views.keepFrames) (* restore v in any frame that displays it *)
    END HandleModelMsg;

    PROCEDURE (v: View) HandleCtrlMsg (f: Views.Frame;
                                                        VAR msg: Controllers.Message;
                                                        VAR focus: Views.View);
        VAR m: Model; umsg: Models.UpdateMsg;
    BEGIN
        WITH msg: Controllers.EditMsg DO
            IF msg.op = Controllers.pasteChar THEN
                m := v.model;
                m.s[m.i] := msg.char; INC(m.i);
                m.s[m.i] := 0X; (* append character to string *)
                Views.SetDirty(v); (* mark view's document as dirty *)
                Models.Broadcast(m, umsg) (* update all views on this model *)
            END
        ELSE            (* ignore other messages *)
        END
    END HandleCtrlMsg;

    PROCEDURE (v:View) HandlePropMsg (VAR msg: Properties.Message);
    BEGIN
        WITH msg: Properties.SizePref DO
            IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
                msg.w := 10 * d; msg.h := 2 * d
            END
        | msg: Properties.FocusPref DO
            msg.setFocus := TRUE
        ELSE
        END
    END HandlePropMsg;

    PROCEDURE Deposit*;
        VAR v: View; m: Model;
    BEGIN
        NEW(m); m.i := 0; m.s := "";
        NEW(v); v.model := m; Stores.Join(v, m);
        Views.Deposit(v)
    END Deposit;

END ObxViews12.
```

❗ "ObxViews12.Deposit; StdCmds.Open"

Multi-view editing is realized by factoring out the view's data into a separate data structure, called a *model*. This model is shared between all its views, i.e., each such view contains a pointer to the model:

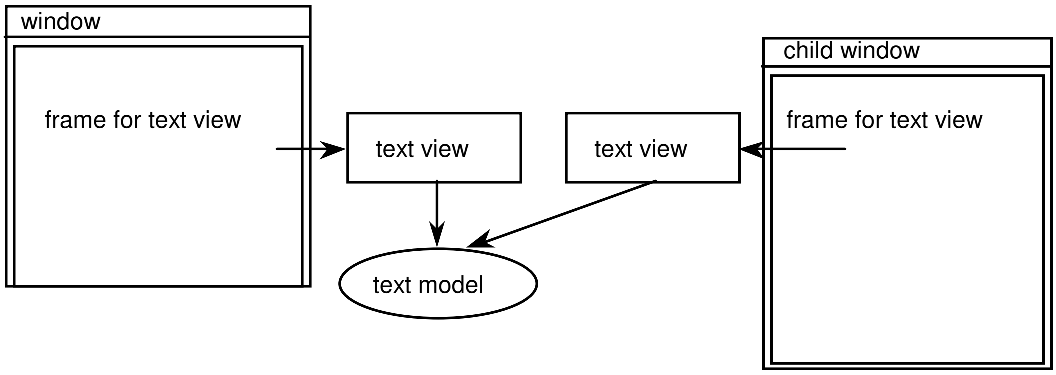


Figure 6.2: Two Views on one Model

A model is, like a view, an extension of a *Stores.Store*, which is the base type for all persistent objects. Stores define the *Internalize* and *Externalize* procedures we've already met for a view. In *ObxViews12*, the model stores the string and its length, while the view stores the whole model! For this purpose, a *Stores.Writer* provides a *WriteStore* and a *Stores.Reader* provides a *ReadStore* procedure.

Stores may form arbitrary graphs. For example, one or several views may refer to the same model, both models and views are store extensions. A graph of stores that may be saved in a file is called a *document*. In order to recognize the boundary of a document upon externalization, stores are bound to a *domain*. All stores of a document refer to the same domain object (*Stores.Domain*).

Domains are also used as containers for operations, i.e., commands performed on some store which can be undone and redone (see next chapter). The operation history is associated with a domain. Furthermore, domains define the set of objects which get notified in a broadcast (either *Views.Broadcast*, *Views.Domaincast* or *Models.Broadcast*).

For more information on stores and domains, see the on-line documentation of module *Stores*.

Now that the model is separated from the view, copying is done by the model, not by the view anymore.

View.ThisModel returns the model of the view. Its default implementation returns *NIL*, which should be overridden in views that contain models. This procedure is used by the framework to find all views that display a given model, in order to implement model broadcasts, as described in the next paragraph.

Views with a model implement the procedure *View.CopyFromModelView* instead of *View.CopyFromSimpleView*. Like *Internalize* and *Externalize*, *CopyFromSimpleView* and *CopyFromModelView* are exported as implement-only procedures, i.e., they can only be implemented, but not called. They are called internally in module *Views*, in its *CopyOf* and *CopyWithNewModel* functions.

When a model changes, all views displaying it must be updated. For this purpose, a *model broadcast* is generated: A model broadcast notifies all views which display a particular model about a model modification that has happened. This gives each view the opportunity to restore its contents on the screen. A model broadcast is generated in the *View.HandleCtrlMsg* procedure by calling *Models.Broadcast*. The message is received by every view which contains the correct model, via its *View.HandleModelMsg* procedure.

This indirection becomes important if different types of views display the same model, e.g., a tabular view and a graphical view on a spreadsheet model; and if instead of restoring the whole view - as has been done in our examples - only the necessary parts of the views are restored. These necessary parts may be completely different for different types of views.

A very sophisticated model may be able to contain ("embed") arbitrary views as part of its contents. For example, a text view may display a text model which contains not only characters (its *intrinsic contents*), but also graphical or other views flowing along in the text stream.

The combination of multi-view editing and hierarchical view embedding can lead to the following situation, where two text views show the same text model, which in turn contains a graphics view. Each text view lives in its own window and thus has its own frame. The graphics view is unique however, since it is embedded in the text model, which is shared by both views. Nevertheless the graphics can be visible in both text views simultaneously, and thus there can be two frames for this one view:

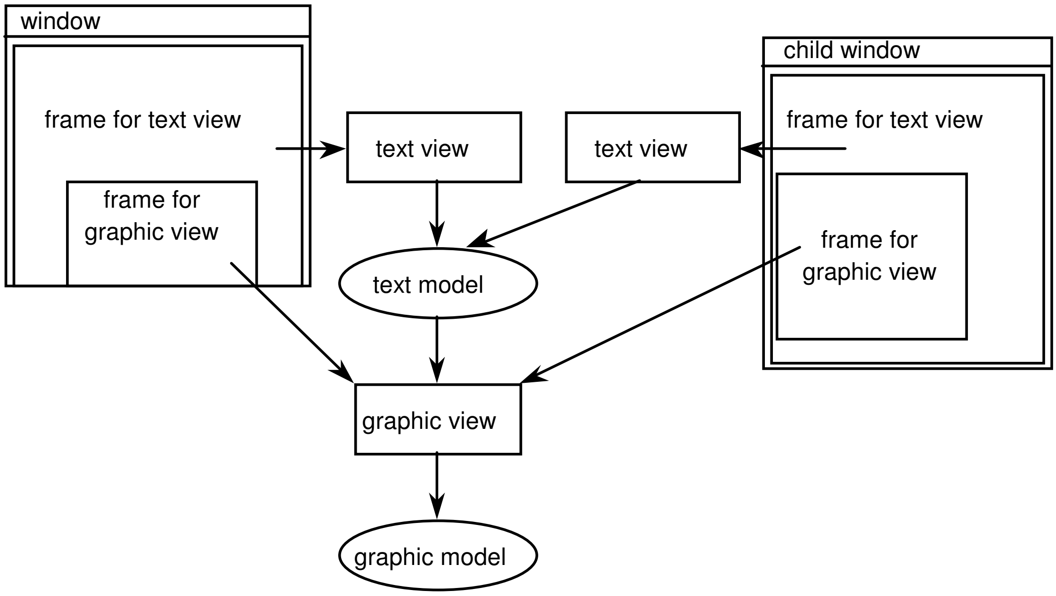


Figure 6-3: Two Frames on one View

As a consequence, one and the same view may be visible in several places on the screen simultaneously. In this case, this means that when the view has changed, several places must be updated: the view must restore the necessary area once for every frame on this view. As a consequence, a notification mechanism must exist which lets the view update each of its frames. This is done in a similar way as the notification mechanism for model changes: with a *view broadcast*. Fortunately, there is a standard update mechanism in the framework which automatically handles this second broadcast level (see below).

We now can summarize the typical events that occur when a user interacts with a view:

Controller message handling:

1) Some controller message is sent to the focus view.
2) The focus view interprets the message and changes its model accordingly.
3) The model broadcasts a model message, describing the change that has been performed.

Model message handling:

4) Every view on this model receives the model message.
5) It determines how its display should change because of this model modification.
6) It broadcasts a view message, describing how its display should change.

View message handling:

7) The view receives the view notification message once for every frame on this view.
8) Every time, the view redraws the frame contents according to the view message.

This mechanism is fundamental to the BlackBox Component Framework. If you have understood it, you have mastered the most complicated part of a view implementation. Interpreting the various controller messages, which are defined in module *Controllers*, is more a matter of diligence than of understanding difficult new concepts. Moreover, you only need to interpret the controller messages in which you are interested, because messages that are not relevant can simply be ignored.

Usually, steps 6, 7, and 8 are handled automatically by the framework: the view merely calls *Views.Update* if a whole view should be updated, or *Views.UpdateIn* if some rectangular part of the view should be updated. Calls of these update procedures cause a *lazy update*, i.e., the framework adds up the region to be updated, but does *not* immediately cause any redrawing. Instead, the command first runs to completion, so that all data structures of the modified views are consistent again. Only the display of the views is still the same as before the command. But then the framework restores all views that have been marked for update. This means that for every frame that displays a view, the view's *Restore* method is called with this frame as argument. If the view is displayed in two (or more) frames in different windows, it is restored two (or more) times; once for every frame. When the framework starts restoring frames of a window, it temporarily redirects the drawing operations of a port into a background pixelmap buffer. When all frames of this window that need updating have been restored, the framework copies the pixelmap buffer to the screen pixelmap, and turns off the redirection of drawing operations.

The effect of this buffering mechanism and of the lazy update mechanism is that minimal flickering occurs. The lazy update mechanism (also used by Apple for the Mac OS, and by Microsoft for Windows) means that even for complex editing operations, a given screen region is only restored once. This can reduce flickering, since flickering can occur when the same area is drawn several times in rapid succession. The buffering mechanism reduces flickering if drawing occurs in layers, from "bottom" to "top" (because this also means drawing several time in rapid succession). For example, you may first draw the background in one color, and then the a filled rectangle on top of it in another color. Thanks to the buffering mechanism, the user never sees the situation after the background is drawn, but before the rectangle is drawn.

The lazy update mechanism also has another advantage: it completely decouples the modification of a model from its drawing. Without this mechanism, it can become rather difficult to correctly perform screen updates for complex editing operations. For example, consider moving a selection of graphics objects in a graphics editor. The selected objects first must be deleted at their old position, this may mean that other objects that have been obscured by the selection now need to be drawn (but not the selected objects themselves!). Then the objects' positions are changed, and then they are redrawn at their new positions (if the source and target positions overlap, drawing has now happened twice). Drawing may have to be done several times, if there are several windows displaying the same data. But of course the change of object positions must only occur once. It is much less tricky to simply call *Views.UpdateIn* for all rectangles that (may) need updating, and to let the framework call the necessary *Restore* methods later!

Normally, the lazy update mechanism is sufficient. Thus a view programmer needs to use explicit view messages only if no complete restore in all frames of a view is desired. This is the case only for marks like selection, focus, or carets, or for rubberbanding or similar feedback effects during mouse tracking. Such marks can sometimes be handled more efficiently because they are involutory: applied twice, they have no effect. Thus marks are often switched on and off through custom view messages *during* a command, not through the delaying lazy update mechanism. But except for such light-weight marks, you should always use the lazy update mechanism described above. Note that there exists a method *Ports.Frame.MarkRect* explicitly for drawing such marks.

6.7 Operations

Now that we have seen the crucial ingredients of a view implementation, a less central feature can be presented. The next program is a variation of the above one, in which a controller message is not interpreted directly. Instead, an *operation* object is created and then executed. An operation provides a *do / undo / redo* capability, as shown in the program below.

The operation we define in this example is the *PasteCharOp*. The operation's *Do* procedure performs the desired modification, and must be involutory, i.e. when called twice, its effect must have been neutralized again. We use the flag *PasteCharOp.do* to specify whether the character *PasteCharOp.char* has to be inserted into or removed from the model *PasteCharOp.m*. In any case, the *Do* procedure has to update all views on the model *PasteCharOp.m* with a model broadcast.

The procedure *NewPasteCharOp* generates a new operation and the procedure *Models.Do* executes this operation, i.e., the operation's *Do* procedure is called and the operation is recorded for a later undo. The name of the operation as specified in the *Models.Do* command will appear in the *Edit* menu after the *Undo* or *Redo* entry respectively.

```
MODULE ObxViews13;
(* Same as ObxViews12, but generate undoable operations for character insertion *)

    IMPORT Fonts, Ports, Stores, Models, Views, Controllers, Properties;

    CONST d = 20 * Ports.point;

    TYPE
        Model = POINTER TO RECORD (Models.Model)
            i: INTEGER;  (* position of next free slot in string *)
            s: ARRAY 256 OF CHAR (* string *)
        END;

        View = POINTER TO RECORD (Views.View)
            model: Model
        END;

        PasteCharOp = POINTER TO RECORD (Stores.Operation)
            model: Model;
            char: CHAR;
            do: BOOLEAN
        END;


    (* PasteCharOp *)

    PROCEDURE (op: PasteCharOp) Do;
        VAR m: Model; msg: Models.UpdateMsg;
    BEGIN
        m := op.model;
        IF op.do THEN  (* do operation's transformation *)
            m.s[m.i] := op.char; INC(m.i) (* insert character into string *)
        ELSE              (* undo operation's transformation *)
            DEC(m.i)     (* remove character from string *)
        END;
        m.s[m.i] := 0X;
        op.do := ~op.do;  (* toggle between "do" and "undo" *)
        Models.Broadcast(m, msg) (* update all views on this model *)
    END Do;

    PROCEDURE NewPasteCharOp (m: Model; char: CHAR): PasteCharOp;
        VAR op: PasteCharOp;
    BEGIN
        NEW(op); op.model := m; op.char := char; op.do := TRUE;
        RETURN op
    END NewPasteCharOp;


    (* Model *)

    PROCEDURE (m: Model) Internalize (VAR rd: Stores.Reader);
        VAR version: INTEGER;
    BEGIN
        rd.ReadVersion(0, 0, version);
        IF ~rd.cancelled THEN
            rd.ReadInt(m.i); rd.ReadString(m.s)
        END
    END Internalize;

    PROCEDURE (m: Model) Externalize (VAR wr: Stores.Writer);
    BEGIN
        wr.WriteVersion(0);
        wr.WriteInt(m.i); wr.WriteString(m.s)
    END Externalize;

    PROCEDURE (m: Model) CopyFrom (source: Stores.Store);
    BEGIN
        WITH source: Model DO
            m.i := source.i; m.s := source.s
        END
    END CopyFrom;


    (* View *)

    PROCEDURE (v: View) Internalize (VAR rd: Stores.Reader);
        VAR version: INTEGER; st: Stores.Store;
    BEGIN
        rd.ReadVersion(0, 0, version);
        IF ~rd.cancelled THEN
            rd.ReadStore(st);
            v.model := st(Model)
        END
    END Internalize;

    PROCEDURE (v: View) Externalize (VAR wr: Stores.Writer);
```

```
BEGIN
    wr.WriteVersion(0);
    wr.WriteStore(v.model)
END Externalize;

PROCEDURE (v: View) CopyFromModelView (source: Views.View; model: Models.Model);
BEGIN
    v.model := model(Model)
END CopyFromModelView;

PROCEDURE (v: View) Restore (f: Views.Frame; l, t, r, b: INTEGER);
BEGIN
    f.DrawString(d, d, Ports.black, v.model.s, Fonts.dir.Default())
END Restore;

PROCEDURE (v: View) ThisModel (): Models.Model;
BEGIN
    RETURN v.model
END ThisModel;

PROCEDURE (v: View) HandleModelMsg (VAR msg: Models.Message);
BEGIN
    Views.Update(v, Views.keepFrames)  (* restore v in any frame that displays it *)
END HandleModelMsg;

PROCEDURE (v: View) HandleCtrlMsg (f: Views.Frame; VAR msg: Controllers.Message;
                                                         VAR focus: Views.View);
    VAR op: Stores.Operation;
BEGIN
    WITH msg: Controllers.EditMsg DO
        IF msg.op = Controllers.pasteChar THEN
            op := NewPasteCharOp(v.model, msg.char); (* generate operation *)
            Models.Do(v.model, "Typing", op) (* execute operation *)
        END
    ELSE                 (* ignore other messages *)
    END
END HandleCtrlMsg;

PROCEDURE (v:View) HandlePropMsg (VAR msg: Properties.Message);
BEGIN
    WITH msg: Properties.SizePref DO
        IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
            msg.w := 10 * d; msg.h := 2 * d
        END
    | msg: Properties.FocusPref DO
        msg.setFocus := TRUE
    ELSE
    END
END HandlePropMsg;

PROCEDURE Deposit*;
    VAR v: View; m: Model;
    BEGIN
        NEW(m); m.i := 0; m.s := "";
        NEW(v); v.model := m; Stores.Join(v, m);
        Views.Deposit(v)
    END Deposit;

END ObxViews13.
```

❗ "ObxViews13.Deposit; StdCmds.Open"

Further examples which use the undo/redo mechanism through operations can be found e.g. in ObxOmosi or ObxLines.

6.8 Separation of interface and implementation

As a last version of our sample view, we modify the previous variant by exporting the *Model* and *View* types, and by separating interface and implementation of these two types. In order to do the latter, so-called *directory* objects are introduced, which generate (hidden) default implementations of abstract data types. A user now might implement its own version of the abstract type and offer it through its own directory object, however he can not inherit from the default implementation and thus avoids the *fragile base class problem*.

Real BlackBox subsystems would additionally split the module into two modules, one for the model and one for the view. A third module with commands might be introduced, if the number and complexity of commands warrant it (e.g., *TextModels*, *TextViews*, and *TextCmds*). Even more complicated views would further split views into views and controllers, each in its own module.

```
MODULE ObxViews14;
(* Same as ObxViews13, but interfaces and implementations separated, and operation directly in Insert
procedure *)

    IMPORT Fonts, Ports, Stores, Models, Views, Controllers, Properties;

    CONST d = 20 * Ports.point;

    TYPE
        Model* = POINTER TO ABSTRACT RECORD (Models.Model) END;

        ModelDirectory* = POINTER TO ABSTRACT RECORD END;

        View* = POINTER TO ABSTRACT RECORD (Views.View) END;

        Directory* = POINTER TO ABSTRACT RECORD END;


        StdModel = POINTER TO RECORD (Model)
            i: INTEGER;  (* position of next free slot in string *)
            s: ARRAY 256 OF CHAR (* string *)
        END;

        StdModelDirectory = POINTER TO RECORD (ModelDirectory) END;
    StdView = POINTER TO RECORD (View)
        model: Model
    END;

    StdDirectory = POINTER TO RECORD (Directory) END;

    PasteCharOp = POINTER TO RECORD (Stores.Operation)
        model: StdModel;
        char: CHAR;
        do: BOOLEAN
    END;


VAR
    mdir-: ModelDirectory;
    dir-: Directory;


(* Model *)

PROCEDURE (m: Model) Insert* (char: CHAR), NEW, ABSTRACT;
PROCEDURE (m: Model) Remove*, NEW, ABSTRACT;
PROCEDURE (m: Model) GetString* (OUT s: ARRAY OF CHAR), NEW, ABSTRACT;

(* ModelDirectory *)

PROCEDURE (d: ModelDirectory) New* (): Model, NEW, ABSTRACT;

(* PasteCharOp *)

PROCEDURE (op: PasteCharOp) Do;
    VAR m: StdModel; msg: Models.UpdateMsg;
BEGIN
    m := op.model;
    IF op.do THEN   (* do operation's transformation *)
        m.s[m.i] := op.char; INC(m.i);
    ELSE                 (* undo operation's transformation *)
        DEC(m.i)      (* remove character from string *)
    END;
    m.s[m.i] := 0X;
    op.do := ~op.do;  (* toggle between "do" and "undo" *)
    Models.Broadcast(m, msg) (* update all views on this model *)
END Do;


(* StdModel *)

PROCEDURE (m: StdModel) Internalize (VAR rd: Stores.Reader);
    VAR version: INTEGER;
BEGIN
    rd.ReadVersion(0, 0, version);
    IF ~rd.cancelled THEN
```

```
        rd.ReadInt(m.i); rd.ReadString(m.s)
    END
END Internalize;

PROCEDURE (m: StdModel) Externalize (VAR wr: Stores.Writer);
BEGIN
    wr.WriteVersion(0);
    wr.WriteInt(m.i); wr.WriteString(m.s)
END Externalize;

PROCEDURE (m: StdModel) CopyFrom (source: Stores.Store);
BEGIN
    WITH source: StdModel DO
        m.i := source.i; m.s := source.s
    END
END CopyFrom;


PROCEDURE (m: StdModel) Insert (char: CHAR);
    VAR op: PasteCharOp;
BEGIN
    NEW(op); op.model := m; op.char := char; op.do := TRUE;
    Models.Do(m, "insertion", op)
END Insert;

PROCEDURE (m: StdModel) Remove;
    VAR msg: Models.UpdateMsg;
BEGIN
    DEC(m.i); m.s[m.i] := 0X;
    Models.Broadcast(m, msg) (* update all views on this model *)
END Remove;

PROCEDURE (m: StdModel) GetString (OUT s: ARRAY OF CHAR);
BEGIN
    s := m.s$
END GetString;


(* StdModelDirectory *)

PROCEDURE (d: StdModelDirectory) New (): Model;
    VAR m: StdModel;
BEGIN
    NEW(m); m.s := "";  m.i := 0; RETURN m
END New;



(* Directory *)

PROCEDURE (d: Directory) New* (m: Model): View, NEW, ABSTRACT;
```

```
(* StdView *)

PROCEDURE (v: StdView) Internalize (VAR rd: Stores.Reader);
    VAR version: INTEGER; st: Stores.Store;
BEGIN
    rd.ReadVersion(0, 0, version);
    IF ~rd.cancelled THEN
        rd.ReadStore(st);
        IF st IS Model THEN
            v.model := st(Model)
        ELSE
            (* concrete model implementation couldn't be loaded->
                an alien store was created *)
            rd.TurnIntoAlien(Stores.alienComponent)
            (* internalization of v is cancelled *)
        END
    END
END Internalize;

PROCEDURE (v: StdView) Externalize (VAR wr: Stores.Writer);
BEGIN
    wr.WriteVersion(0);
    wr.WriteStore(v.model)
END Externalize;

PROCEDURE (v: StdView) CopyFromModelView (source: Views.View; model: Models.Model);
BEGIN
    WITH source: StdView DO
        v.model := model(Model)
    END
END CopyFromModelView;

PROCEDURE (v: StdView) Restore (f: Views.Frame; l, t, r, b: INTEGER);
    VAR s: ARRAY 256 OF CHAR;
BEGIN
    v.model.GetString(s);
    f.DrawString(d, d, Ports.black, s, Fonts.dir.Default())
END Restore;

PROCEDURE (v: StdView) ThisModel (): Models.Model;
BEGIN
    RETURN v.model
END ThisModel;

PROCEDURE (v: StdView) HandleModelMsg (VAR msg: Models.Message);
BEGIN
    Views.Update(v, Views.keepFrames) (* restore v in any frame that displays it *)
END HandleModelMsg;

PROCEDURE (v: StdView) HandleCtrlMsg (f: Views.Frame;
                                                            VAR msg: Controllers.Message;
                                                            VAR focus: Views.View);
```

```
BEGIN
    WITH msg: Controllers.EditMsg DO
        IF msg.op = Controllers.pasteChar THEN
            v.model.Insert(msg.char) (*  undoable insertion *)
        END
    ELSE                (* ignore other messages *)
    END
END HandleCtrlMsg;

PROCEDURE (v:StdView) HandlePropMsg (VAR msg: Properties.Message);
BEGIN
    WITH msg: Properties.SizePref DO
        IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
            msg.w := 10 * d; msg.h := 2 *d
        END
    | msg: Properties.FocusPref DO
        msg.setFocus := TRUE
    ELSE
    END
END HandlePropMsg;


(* StdDirectory *)

PROCEDURE (d: StdDirectory) New* (m: Model): View;
    VAR v: StdView;
BEGIN
    ASSERT(m # NIL, 20);
    NEW(v); v.model := m; Stores.Join(v, m);
    RETURN v
END New;


PROCEDURE Deposit*;
    VAR v: View;
BEGIN
    v := dir.New(mdir.New());
    Views.Deposit(v)
END Deposit;

PROCEDURE SetModelDir* (d: ModelDirectory);
BEGIN
    ASSERT(d # NIL, 20);
    mdir := d
END SetModelDir;

PROCEDURE SetDir* (d: Directory);
BEGIN
    ASSERT(d # NIL, 20);
    dir := d
END SetDir;
```

```
		PROCEDURE Init;
			VAR md: StdModelDirectory; d: StdDirectory;
		BEGIN
			NEW(md); mdir := md;
			NEW(d); dir := d
		END Init;

	BEGIN
		Init
	END ObxViews14.
```

❗ "ObxViews14.Deposit; StdCmds.Open"

A simple example of a BlackBox module which follows this design is *DevMarkers*. The marker views are simple views without a model, therefore only a directory object to generate new views is offered. The second directory object *DevMarkers.stdDir* keeps the standard implementation provided by this module. It might be used to install back the default implementation or to reuse the default implementation (as an instance) in another component.

The separation of a type's definition from its implementation is recommended in the design of new BlackBox subsystems. However, simple view types which won't become publicly available or which are not meant to be extended can certainly dispense with this additional effort.

Note that the subsystem wizard (menu item Tools->Create Subsystem...) helps to generate templates for the different kinds of view. In particular, it is possible to generate separate model and view modules. The tool uses the template texts stored in *Dev/Rsrc/New*. You may want to study the files *Models5*, *Views5*, and *Cmds5* in this directory.

Appendix A: A Brief History of Pascal

Algol

The language Component Pascal is the culmination of several decades of research. It is the youngest member of the Algol family of languages. Algol, defined in 1960, was the first high-level language with a readable, *structured*, and systematically defined syntax. While successful as a notation for mathematical algorithms, it lacked important data types, such as pointers or characters.

Pascal

In the late sixties, several proposals for an evolutionary successor to Algol were developed. The most successful one was Pascal, defined in 1970 by Prof. Niklaus Wirth at ETH Zürich, the Swiss Federal Institute of Technology. Besides cleaning up or leaving out some of Algol's more obscure features, Pascal added the capability to define new data types out of simpler existing ones. Pascal also supported *dynamic data structures*; i.e., data structures which can grow and shrink while a program is running. Pascal received a big boost when ETH released a Pascal compiler that produced a simple intermediate code for a virtual machine (P-code), instead of true native code for a particular machine. This simplified porting Pascal to other processor architectures considerably, because only a new P-code interpreter needed be written for this purpose, not a whole new compiler. One of these projects had been undertaken at the University of California, San Diego. Remarkably, this implementation (UCSD Pascal) didn't require a large and expensive mainframe computer, it ran on the then new Apple II personal computers. This gave Pascal a second important boost. The third one came when Borland released TurboPascal, a fast and inexpensive compiler, and integrated development environment for the IBM PC. Later, Borland revived its version of Pascal when it introduced the rapid application development environment Delphi.
Pascal has greatly influenced the design and evolution of many other languages, from Ada to Visual Basic.

Modula-2

In the mid-seventies, inspired by a sabbatical at the Xerox Palo Alto Research Center *PARC*, Wirth started a project to develop a new workstation computer. This workstation should be completely programmable in a high-level language, thus the language had to provide direct access to the underlying hardware. Furthermore, it had to support *team programming* and modern software engineering principles, such as *abstract data types*. These requirements led to the programming language Modula-2 (1979). Modula-2 retained the successful features of Pascal, and added a module system as well as a controlled way to circumvent the language's type system when doing *low-level programming*; e.g., when implementing device drivers. Modules could be added to the operating system at run-time. In fact, the whole operating system consisted of a collection of modules, without a distinguished kernel or similar artefact. Modules could be compiled and loaded separately, with complete *type and version checking of their interfaces*.
Modula-2 has made inroads in particular into safety-critical areas, such as traffic control systems.

Simula, Smalltalk, and Cedar

Wirth's interest remained with desktop computers, however, and again an important impulse came from Xerox PARC. PARC was the place where the workstation, the laser printer, the local area network, the bitmap display, and many other enabling technologies have been invented. Also, PARC adopted and made popular several older and barely known technologies, like the mouse, interactive graphics, and *object-oriented programming*. The latter concept, if not the term, was first applied to a high-level language in Simula (1966), another member of the Algol language family. As its name suggests, Simula used object-orientation primarily for simulation purposes. Xerox PARC's Smalltalk language (1983), however, used it for about *anything*. The Smalltalk project broke new ground also in user interface design: the graphical user interface (GUI) as we know it today was developed for the Smalltalk system.

At PARC, these ideas influenced other projects, e.g., the Cedar language, a Pascal-style language. Like Smalltalk and later Oberon, Cedar was not only the name of a language but also of an operating system. The Cedar operating system was impressive and powerful, but also complex and unstable.

Oberon

The Oberon project was initiated in 1985 at ETH by Wirth and his colleague Jürg Gutknecht. It was an attempt to distill the essence of Cedar into a comprehensive, but still comprehensible, workstation operating system. The resulting system became very small and efficient, working well with only 2 MB of RAM and 10 MB of disk space. An important reason for the small size of the Oberon system was its component-oriented design: instead of integrating all desirable features into one monolithic software colossus, the less frequently used software components (modules) could be implemented as extensions of the core system. Such components were only loaded when they were actually needed, and they could be shared by all applications.

Wirth realized that component-oriented programming required some features of object-oriented programming, such as *information hiding*, *late binding*, and *polymorphism*.

Information hiding was the great strength of Modula-2. Late binding was supported by Modula-2 in the form of procedure variables. However, polymorphism was lacking. For this reason, Wirth added *type extension*: a record type could be declared as an extension of another record type. An extended type could be used wherever one of its base types might be used.

But component-oriented programming is more than object-oriented programming. In a component-based system, a component may share its data structures with arbitrary other components, about which it doesn't know anything. These components usually don't know about each other's existence either. Such mutual ignorance makes the management of dynamic data structures, in particular the correct deallocation of unused memory, a *fundamentally* more difficult problem than in closed software systems. Consequently, it must be left to the language implementation to find out when memory is not used anymore, in order to safely reclaim it for later use. A system service which performs such an automatic storage reclamation is called a *garbage collector*. Garbage collection prevents two of the most evasive and downright dangerous programming errors: *memory leaks* (not giving free unused memory) and *dangling pointers* (releasing memory too early). Dangling pointers let one component destroy data structures that belong to other components. Such a violation of *type safety* must be prevented, because component-based systems may contain many third-party components of unknown quality (e.g., downloaded from the Internet).

While Algol-family languages always had a reputation of being safe, complete type safety (and thus garbage collection) still was a quantum leap forward. It also was the reason why complete compatibility with Modula-2 was not possible. The resulting revision of Modula-2 was called the same way as the system: Oberon.

Oberon's module system, like the one of Modula-2, provided information hiding for entire collections of types, not only for individual objects. This allowed to define and guarantee invariants spanning several cooperating objects. In other words: it allowed developers to invent higher-level safety mechanisms, by building on the basic *module safety* and type safety provided by a good Oberon implementation.

Orthodox object-oriented programming languages such as Smalltalk had neglected both typing (by not supporting types) and information hiding (by restricting it to objects and classes), which was a major step backwards as far as software engineering is concerned. Oberon reconciled the worlds of object-oriented and modular programming.

As a final requirement of component-oriented programming, it had to be possible to *dynamically load* new components. In Oberon, the unit of loading was the same as the unit of compilation: a module.

Component Pascal

In 1992, a cooperation with Prof. H. P. Mössenböck led to a few additions to the original Oberon language ("Oberon-2"). It became the de-facto standard of the language.

In 1997, the ETH spin-off Oberon microsystems, Inc. (with Wirth on its board of directors) made some small extensions to Oberon-2 and called it Component Pascal, to better express its focus (component-oriented programming) and its origin (Pascal). It is the industrial-strength version of Oberon, so to say.

The main thrust of the enhancements compared to Oberon-2 was to give the designer of a framework (i.e., of module interfaces that define abstract classes for a particular problem domain) more control over the framework's custom safety properties. The benefit is that it becomes easier to ascertain the integrity of a large component-based system, which is particularly important during iterative design cycles when the framework is being developed, and later when the system architecture must be refactored to enable further evolution and maintenance.

BlackBox

Oberon microsystems developed the *BlackBox Component Framework* starting in 1992 (originally it was called Oberon/F). This component-oriented framework is written in Component Pascal, and simplifies the development of graphical user interface components. It comes bundled with several BlackBox extension components, including a word processor, a visual designer, an SQL database access facility, an integrated development environment, and the Component Pascal run-time system. The complete package is an advanced yet light-weight rapid application development (RAD) tool for components, called *BlackBox Component Builder*. It is light-weight because it is completely built out of Component Pascal modules - including the kernel with the garbage collector, and the Component Pascal compiler itself.

Appendix B: Differences between Pascal and Component Pascal

Eliminated Features

• Subrange types
Use a standard integer type instead.

• Enumeration types
Use integer constants instead.

• Arbitrary array ranges
Arrays are always defined over the integer range 0..max-1.

Example

```
   A = ARRAY 16 OF INTEGER    (* legal indices are in the range 0..15 *)
```

• No general sets
Type SET denotes the integer set which may include the elements 0..31.

• No explicit DISPOSE
Memory is reclaimed automatically by the garbage collector. Instead of calling DISPOSE, simply set the variable to NIL.

• No variant records
Use record extension instead.

• No packed structures
Use SHORTCHAR or BYTE types for byte-sized values.

• No GOTO

• No PRED and SUCC standard functions
Use DEC or INC on integer values instead.

• No built-in input/output facilities
No file types. I/O is provided by library routines.

Changed Features

- **Standard procedure ENTIER instead of ROUND**

- **Syntax for REAL constants**

3.0E+4 but not 3.0e+4

- **Syntax for pointer type declarations**

P = POINTER TO R
instead of
P = ^R

- **Syntax for case statement**

"|" instead of ";" as case separator.
ELSE clause.

Example

```
CASE i * 3 - 1 OF
  0: StdLog.String("zero")
| 1..9: StdLog.String("one to nine")
| 10, 20: StdLog.String("ten or twenty")
ELSE StdLog.String("something else")
END
```

- **Procedure name must be repeated**

Example

```
PROCEDURE DrawDot (x, y: INTEGER);
BEGIN
END DrawDot;
```

- **Case is significant**

Small letters are distinguished from capital letters.

Example "proc" is not the same as "Proc".

- **String syntax**

String literals are either enclosed between " or between '. There cannot be both single and double quotes in one string. String literals of length one are assignment-compatible to character variables.

Examples

```
"That's great"    'Write "hello world" to the screen'
ch := "x"
ch := 'x'
```

- **Comments**

Comments are enclosed between (* and *) and may be nested.

- **Set brackets**

Set constants are given between { and } instead of [and].

Example   {0..2, 4, j..2 * k}

- **Function syntax**

Use keyword PROCEDURE for functions also, instead of FUNCTION.
Procedures with a return value always have a (possibly empty) parameter list in their declarations and in calls to them.
The function result is returned explicitly by a RETURN statement, instead of an assignment to the function name.

Example

```
   PROCEDURE Fun (): INTEGER;
   BEGIN
      RETURN 5
   END Fun;

    instead of

   FUNCTION Fun: INTEGER;
   BEGIN
      Fun := 5
   END;

   n := Fun()   instead of   n := Fun
```

- **Declarations**

The sequence of declarations is
{ ConstDecl | TypeDecl | VarDecl} {ProcDecl | ForwardDecl}
instead of
[ConstDecl] [TypeDecl] [VarDecl] {ProcDecl}.

Forward declarations are necessary if a procedure is used before it is defined.

Example

```
   PROCEDURE ^ Proc;
   instead of
   PROCEDURE Proc; FORWARD;
```

- **Procedure types**

Procedures may not only be passed to parameters, but also to procedure-typed variables.

Example

```
   TYPE P = PROCEDURE (x, y: INTEGER);
   VAR v: P;
   v := DrawDot;    (* assign *)
   v(3, 5);    (* call DrawDot(3, 5) *)
```

- **Explicit END instead of compound statement**

BEGIN can only occur before a statement sequence, but not in it. IF, WHILE, and LOOP are always terminated by END.

- **WITH statement**

A WITH statement is a regional type guard, it does not imply a hidden variable and does not open a new scope.
See language reference for more details.

- **ELSIF**

IF statements can have several branches.

Example

```
IF name = "top" THEN
   StdLog.Int(0)
ELSIF name = "bottom" THEN
   StdLog.Int(1)
ELSIF name = " charm" THEN
   StdLog.Int(2)
ELSIF name = "beauty" THEN
   StdLog.Int(3)
ELSE
   StdLog.String("strange")
END
```

- **BY instead of only DOWNTO in FOR**

FOR loops may use any constant value as increment or decrement.

Example

```
FOR i := 15 TO 0 BY -1 DO StdLog.Int(i, 0) END
```

- **Boolean expressions use short-circuit evaluation**

A Boolean expression terminates as soon as its result can be determined.

Example

The following expression does not cause a run-time error when p = NIL:

```
IF (p # NIL) & (p.name = "quark") THEN
```

- **Constant expressions**

In constant declarations, not only literals, but also constant expressions are allowed.

Example

```
CONST
   zero = ORD("0");
   one = zero + 1;
```

- **Different operators**

# is used instead of <> for inequality test.
& is used instead of AND for logical conjunctions.
~ is used instead of NOT for logical negation.

- **Explicit conversion to included type with SHORT**

Type inclusion for numeric types allows to assign values of an included type to an including type. To assign in the other direction, the standard procedure SHORT must be used.

Example

```
int := shortint;
```

```
shortint := SHORT(int)
```

New Features

• **Hexadecimal numbers and characters**

Example

```
100H    (* decimal 256 *)
0DX    (* carriage return *)
```

• **Additional numeric types**
LONGINT, SHORTINT, BYTE, SHORTREAL have been added.

• **Symmetric set difference**
Sets can be subtracted.

• **New standard procedures**
The new standard procedures INC, DEC, INCL, EXCL, SIZE, ASH, HALT, ASSERT, LEN, LSH, MAX, MIN, BITS, CAP, ENTIER, LONG and SHORT have been added.

• **LOOP with EXIT**
There is a new loop statement with an explicit exit statement. See language report for more details.

• **ARRAY OF CHAR can be compared**
Character arrays can be compared with the =, #, <, >, <=, and >= operators.

• **Open arrays, multidimensional arrays**
Arrays without predefined sizes can be defined, possibly with several dimensions.

Examples

```
VAR a: POINTER TO ARRAY OF CHAR;
NEW(a, 16)

PROCEDURE ScalarProduct (a, b: ARRAY OF REAL; VAR c: ARRAY OF REAL);

TYPE Matrix = ARRAY OF ARRAY OF REAL;
PROCEDURE VectorProduct (a, b: ARRAY OF REAL; VAR c: Matrix);
```

• **Pointer dereferencing is optional**
The dereferencing operator ^ can be omitted.

Example

```
root.next.value := 5
instead of
root^.next^.value := 5
```

• **Modules**
Modules are the units of compilation, of information hiding, and of loading. Information hiding is one of the main features of object-oriented programming. Various levels of information hiding are possible: complete hiding, read-only / implement-only export, full export.
See language report for more details.

• **Type extension**

Record types (pointer types) can be extended, thus providing for polymorphism. Polymorphism is one of the main features of object-oriented programming.

- **Methods**

Procedures can be bound to record types (pointer types), thus providing late binding. Late binding is one of the main features of object-oriented programming. Such procedures are also called *methods*.

- **String operator**

The string (sequence of characters) that is contained in an array of character can be selected by using the $-selector.

- **Record attributes**

Records are non-extensible by default, but may be marked as EXTENSIBLE, ABSTRACT, or LIMITED.

- **Method attributes**

Methods are non-extensible by default, but may be marked as EXTENSIBLE, ABSTRACT, or EMTPY. Newly introduced methods must be marked as NEW.