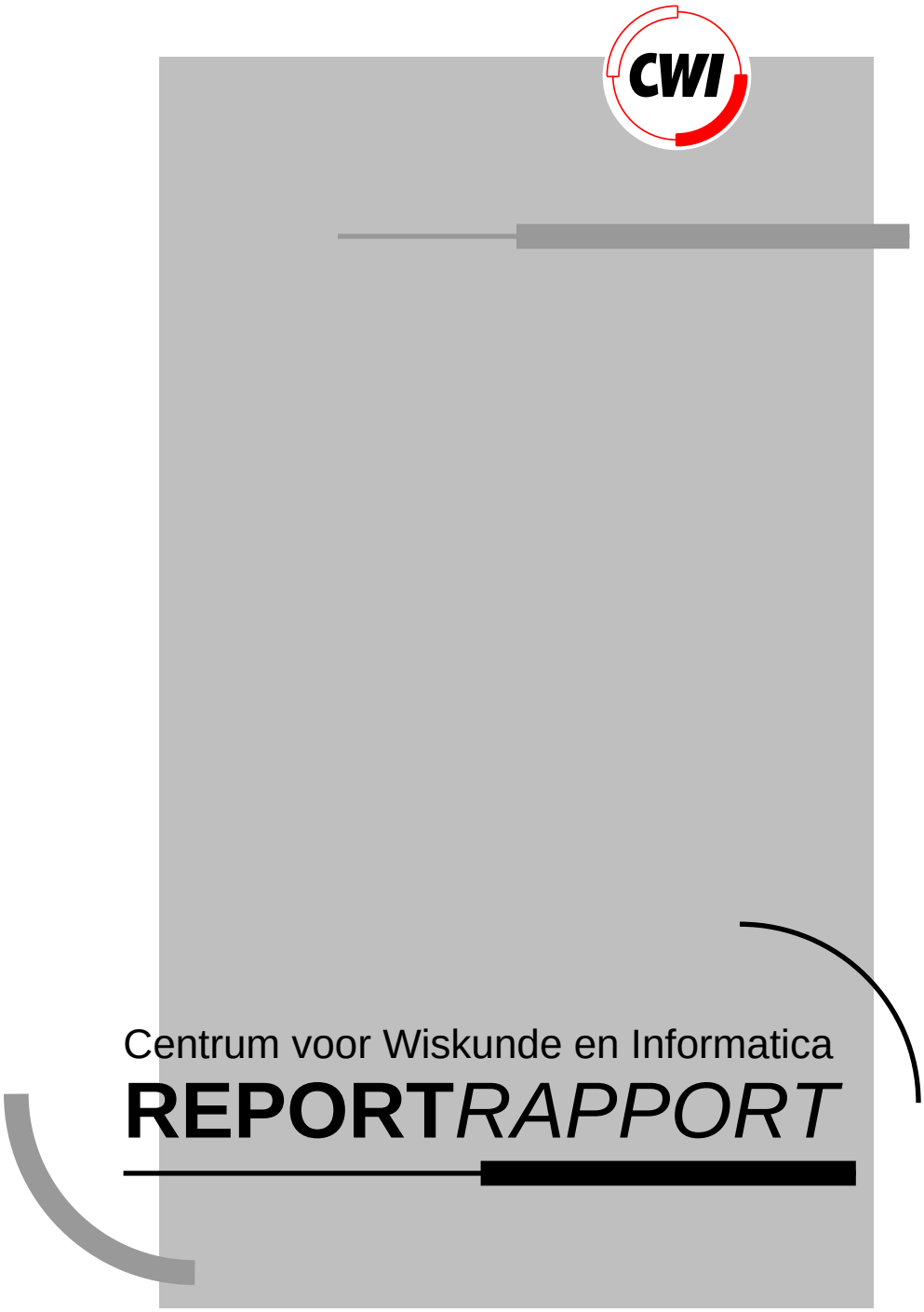

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

The Views application environment

S. Pemberton

Computer Science/Department of Algorithmics and Architecture

CS-R9257 1992

The Views Application Environment

Steven Pemberton

CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Email: Steven.Pemberton@cwi.nl

Abstract

There are a number of problems associated with current windowing environments, such as the lack of consistency both within and between applications, the poor integration between applications, little or no interoperability, 'hard-wired' and inflexible interfaces, and the enormous cost of producing applications. These problems are examined, and a unified approach that addresses them is presented.

1991 Mathematics Subject Classification: 68U99, 68N99.

1991 CR Categories: H.1.2, H.5.2, D39, H.5.0, I.7.2.

Keywords and Phrases: application environments, user interfaces, information interfaces and presentation, document preparation, ergonomics of computer software.

1 Introduction

As computers have become more powerful, and the emphasis has moved from mainframes to personal computers, so the emphasis in software has moved from batch to interactive programs. Accompanying this change of emphasis, attention has focussed on the user-interface of programs: the manner in which one works with a program, and what it is possible to do with it.

And as the number of interactive programs has increased, the users have been faced with an ever increasing number of different user-interfaces, so that to accomplish some tasks you may have to change user-interface several times, leading to confusion, errors, and inefficiency. For instance, from the shell you start up a news reader program, from the news reader you reply to a news article, which starts up a mail program, from which you start an editor: four different user interfaces in quick succession; or to produce a report can involve rapid swapping between text-processor, drawing program, spread-sheet, and file finder.

In order to address some of these problems, various approaches have been tried, with the most prevalent being the ‘toolkit plus guidelines’ approach: the system supplies a ‘toolkit’ of routines that a programmer can call from a program, and a set of guidelines recommending how and where they should be used.

An exemplary instance of this is the Apple Macintosh computer, one of the better examples of a unified approach to user-interfaces. But even with this system, two of the very first applications available – MacWrite and MacPaint – had divergent user-interfaces (for instance different ways to scroll a window), and in general each individual program, while adhering largely to the interface guidelines, has its own peculiarities.

Furthermore, even within the standards of the Macintosh user-interface, the user is confronted with different ways of doing essentially the same thing. For instance, the file-browser as presented by the finder looks to the user very different to the file-browser presented in dialogues (see Figure 1) yet in both cases the task at hand is to select a file to use.

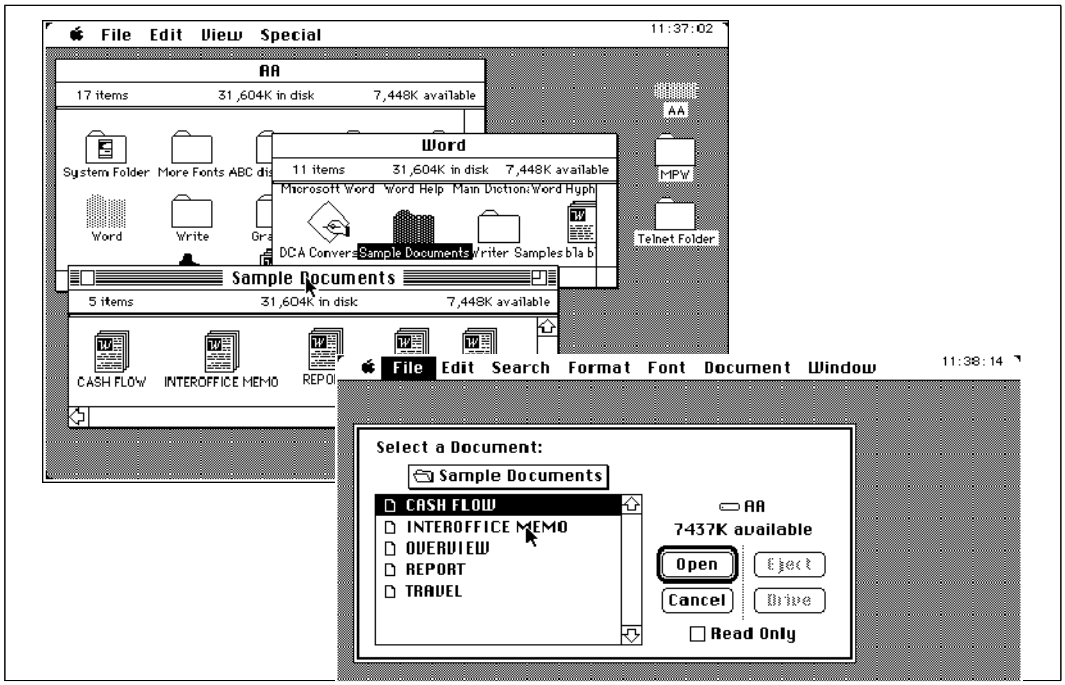

Figure 1. Within a short time, the new Macintosh user can be confronted with two different styles of browser, both with essentially the same task, but not recognisable as similar in appearance, and with different ways of navigating in the file-store

The underlying problem is that while toolkits bring the user-interfaces of programs closer to each-other, they can't guarantee that they will be applied in the same way, since each program must be separately written, interface and all.A better approach is a user-interface layer above the whole system (see Figure 2) so that the individual applications are unaware of user-interface issues [1]. This 'interface independence' (or 'Dialogue Independence' [2]) can be likened to the 'I/O independence' of the 1960's, where the responsibility of device-dependent input-output was taken out of the hands of individual programs, and placed in a central kernel of the system that all programs had access to.

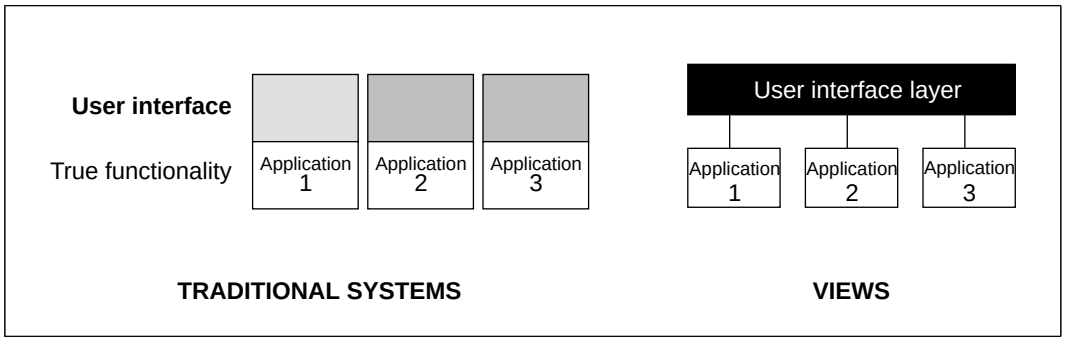

Figure 2. In current systems, each application has to implement its own user interface. Even on systems where a toolbox of user interface tools is supplied, user interfaces of different applications are often different, interfering with a user's optimal use of the computer. In Views, the user interface is a separate layer supplied by the system, offering consistency across applications.

Another good example of the idea is file-name expansion. This is where you can give a program a template pattern of the file names you want, and the system expands this pattern into the true list of file names. For example, the print command '`print *.doc`' asks for all files ending in '`.doc`' to be printed. On systems where this facility is supplied as a central service (such as the Unix shell), all programs have the advantage of the facility without having to know anything about it. On the other hand, on systems where the facility is only part of the 'toolkit' (such as MS-DOS), the only programs that have the facility are those where the individual programmer has taken the trouble to supply it, which leads to confusion and errors for the user, who must try and remember which commands allow it, and which not (see Figure 3). However, even Unix is not immune to this sort of confusion, as can be seen from the extreme example given (see Figure 4) where the user has to change context mid-line.

Another problem with current systems is a lack of integration between applications. Again drawing on the Apple Macintosh as a well-known example, it pioneered multi-media applications, where for instance you can include text and pictures in a single document. But you must use different programs for producing pictures and producing text, so firstly you have to decide whether you want a text with drawings in it, or a drawing with text in. They are both possible, and can produce identical results (Figure 5). Then you have to change context and run a different program to produce your drawing that you want to include in your text (or vice versa). Finally, once you have included the drawing, it is 'frozen': there is no chance to make further changes to it, except to delete it and replace it with a new version.

This last point is an example of the lack of interoperability between applications in windowing environments. In Unix-like environments, for example, many applications can be plugged together to produce new applications and tools. Even though the data-model used (single lines of text) is rather impoverished, a large range of applications can be used in this way. For instance, if you have a large, compressed, file of data, and you want to

Now what was that file called? I know: I'll get a directory listing:

```
C> DIR
(Hundreds of filenames flash past too fast to read)
```

Oh. Well, I think it ended with `.TXT`*, so I'll try that:*

```
C> DIR *.TXT
Volume in drive C is DOCUMENTS
Directory of C:
TDXF34J1 TXT 3612303 20-08-90 11:50a
1 File(s) 4571680 bytes free
```

Aha! Only one file ends with `.TXT`*, that's lucky. Now to read it:*

```
C> TYPE *.TXT
Invalid filename or file not found
```

What? Not found? Lets just check that:

```
C> DIR *.TXT
Volume in drive C is DOCUMENTS
Directory of C:
TDXF34J1 TXT 3612303 20-08-90 11:50a
1 File(s) 4571680 bytes free
```

*Well, it's definitely still there. Perhaps I can't use a * here. Now, what was it called again?*

```
C> TYPE TDXF34J1.TXT
(Output flashes past too fast too read)
```

Oh dear. Well, let's see, `MORE` *lets you read something a page at a time. I'd better use the full name again:*

```
C> MORE TDXF34J1.TXT
(Nothing happens)
```

What? What's happened? Has it crashed? Perhaps the file was too big for `MORE`*. I'd better reboot I suppose.*

Figure 3. Even in text-based applications, the toolbox problem rears its head. Here is an extract from an interactive MS-DOS session. The user wants to read a file, but can't remember what it is called. This involves a sub-task of trying to find the name of the file. In the process, inconsistencies between each command cause confusion and errors.

```
grep b.\\.c b?.c
```

Figure 4. Sometimes even performing a single task can involve context switching. In this Unix example all files whose names are 4 characters long and begin with a `b` and end with `.c` are searched for lines containing the same names. The patterns have to be specified in two different ways, because one of the patterns is addressed to the shell, and the other to `grep`, the program being run. (Actually, the situation is even worse, because the number of backslashes '\' often has to be found by trial and error, since it is unclear for the user which program they are addressed to — context confusion at its worst)

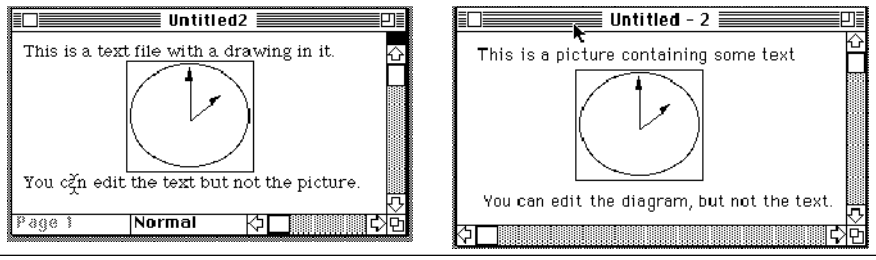

Figure 5. Two similar documents, produced with two different applications. If an application allows you to import documents from other applications at all, once you have done so, the structure of the original is usually lost.

select the second field of each line that starts with the word ‘Total’, and total them up. It does not take much work to be able to write something like:

```
zcat data.Z | grep ^Total | field 2 | total
```

which would uncompress the file on the fly, select the lines, select the field, and cause them to be totalled. In windowing environments, something like this would be impossible; the best you can do is start each separate application up separately, and store intermediate results: it is generally not possible to link applications together to interoperate. This has caused a number of many-headed monolithic applications to emerge, allowing you to at least pass data between a small (fixed) number of applications, such as database, spread-sheet, and text-processor.

And all these problems come with another enormous difficulty: the cost of producing the user-interface. It is said that 90% of the code of an application can be taken up with pure user-interface matters. As an extreme example, the standard C “Hello World” program, 3 or 4 lines long, can expand to 250 lines or more, if you produce a basic version for a windowing environment.

2 Views

Views is an experimental application environment that addresses these problems by supplying a framework that new applications can be added to, offering a consistent and integrated user-interface across applications.

The project has developed from an earlier project at the CWI, the ABC Programming Language [3]. The purpose of ABC is to offer a powerful programming language and environment that is extremely easy to learn and use. As part of this aim, it was wanted to reduce the number of different ‘faces’ that the user would see from the system: traditionally when programming, you must learn the command language of the computing system you are using, the command set of an editor, the programming language itself, how to use the compiler, and so on. With ABC, this set was reduced to two elements: the language and the editor. ABC is not only the programming language, but also the command language, and for tasks that are not carried out with the ABC language, you use the editor: this includes actions like renaming and deleting objects, which is done by ‘direct manipulation’ of the objects involved [4].

It was quickly obvious that this method of working could be generalised to more tasks than programming alone, and thus was born the Views project. Views, then, provides the user with a computing environment that is characterised by:

- What-you-see-is-what-you-get (WYSIWYG)
- Direct-manipulation
- An open-architecture
- A consistent interface across applications.

WYSIWYG refers to systems that try to keep the screen up-to-date with the true state of things. In fact Views is slightly stronger, leading us to coin a new term: TAXATA – things are exactly as they appear. While usually in WYSIWYG systems you are working with a copy of the object in some buffer (in other words, what you see is what you *will* get), in Views you are always working with the object itself.

Open architecture means that it is easy to add new applications to the system, and that the user has much control and choice over individual aspects of the system and its interface.

3 The user's conceptual model

By 'the user's conceptual model' is meant the model or set of rules that the user forms when working with the system. For instance, if you work on a file, do you work on the file itself, or a copy? Since all applications under Views have the same user-interface, the model has to be one that is suitable for all applications.

The basic model in Views is this: every object in the system is editable, every action is carried out by editing; when you edit, you edit the object directly.

An object can be for example a text document, the clock, a diagram, or a menu. Traditionally, editing a file is performed on a copy, which then has to be explicitly written back to the disk. The Views model is closer to what one does in everyday life when changing a document. The one major advantage of editing a copy of a document – that you can retrieve the original version after a disastrous mistake – is offset by an unlimited undo mechanism in Views.

As an example of the basic mechanism, consider document management. Instead of individual commands to list the documents that you have in a directory or folder, to rename them, to delete them, copy them, and so on, you just 'visit' the folder – which is a document in itself in Views – which causes its contents to be displayed on the screen. To rename a document in the folder you just edit its name; to delete it, you just delete its entry; to copy a document, you just use the normal copy and paste facility of the editor.

Similarly, to read electronic mail you just 'visit' your mail box. This causes its contents to be displayed as a list of message headers. Again, you can visit these individual messages (themselves documents), rename them, delete them, copy them, all in exactly the same way.

To print documents, you just copy them to the document representing the printer queue; to cancel printing, you just delete its entry; if you keep the printer queue document open on your screen, you can watch the progress of your printing jobs.

And so on for other tasks: reading news, listing and deleting running processes, editing textual documents, or amending a spread-sheet. Surprising applications, considered from a traditional point of view, include setting the time of day by editing the clock, and rearranging and renaming the menus, and redefining the shortcuts for menu entries, by editing a document describing the menus.

A major advantage that should be emphasised, is that once the user has learnt the basic actions of working with the editor, it should then always be obvious how to deal with a new application that the user hasn't seen before.

4 Data model

Views supplies a data-layer to applications (Figure 6). Views objects are structured, thus containing other objects, and consist only of 'content-full' parts: they contain no details of formatting or other display information, which is added by a separate process when objects are displayed.

Each object has a type, which describes the internal structure of the object, and its external representation, be that as text, as some graphical representation, or even some other medium, such as sound. In general, objects can be viewed in different ways, even simultaneously, for instance as text in one view, but graphically in another (Figure 7). When an object is displayed, the description of its external representation is accessed and used to determine how the object should look.

There is a generic editor which knows about the structure of objects, and allows the user to edit all objects in the same way, regardless of how they are displayed.

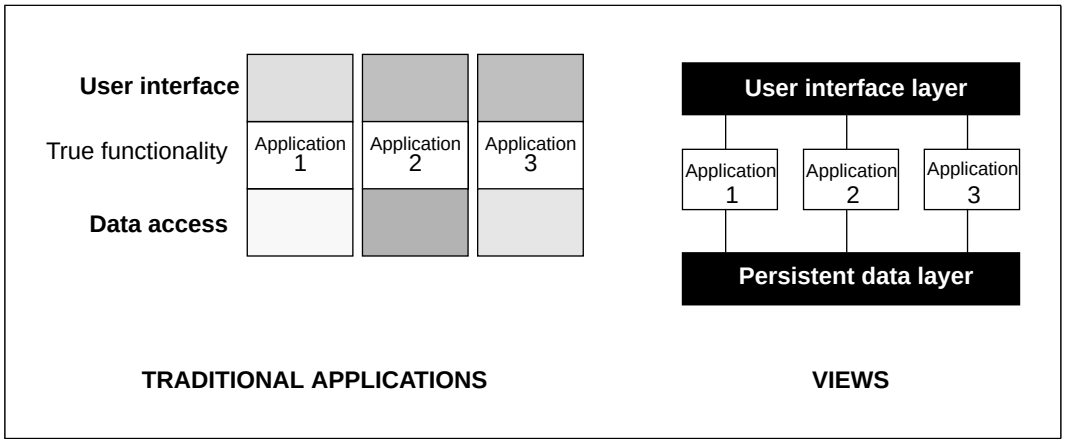

Figure 6. Rather than requiring that each application define its own data-formats as is the case with traditional applications, Views supplies a data-layer that takes care of external data-formats and where objects are stored. This allows any application to import objects from other applications without losing information about their structure and without having to know about files or even the existence of external storage.

Since objects may contain other objects, the distinction between applications blurs. For instance, if the user pastes a graphical object into a text document, the graphic is displayed in the same way, and is editable in the same way, as before.

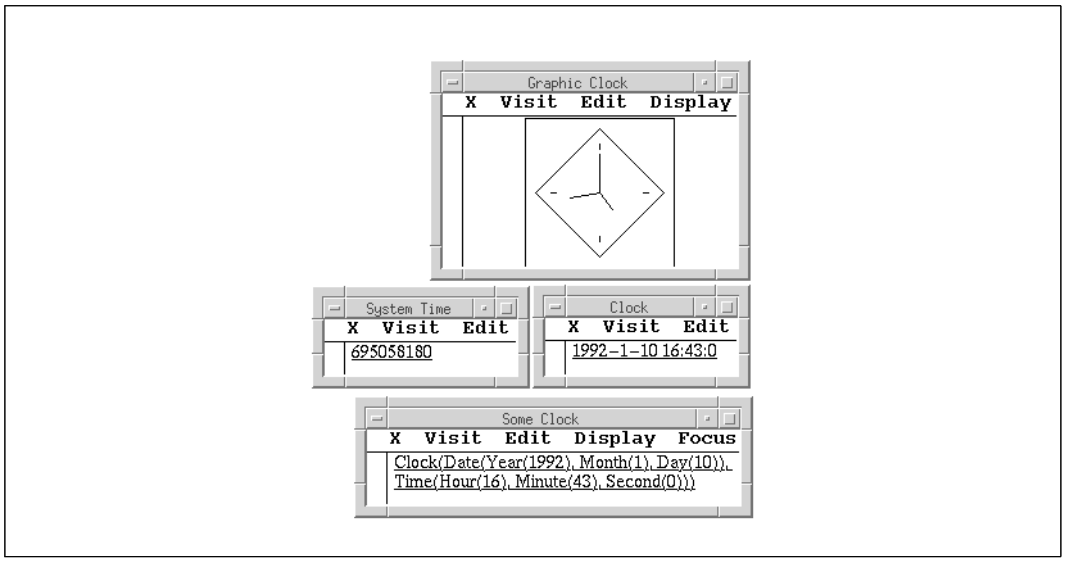

Figure 7. Four views of the time. On the left is the time as represented internally by the computer: the number of seconds since the beginning of 1970. This value is then projected in three different ways: as an analogue-style display of the time, as a digital-style display, and lastly showing the internal structure of the digital-style display, without any added formatting. In this last display, the type of each (sub-) object is given, followed by its value in brackets. Each display also shows its name, and has a ‘menu-bar’ for editing the object and controlling the display.

To the application, all objects are directly accessible: the existence of disks and main-memory is transparent. The data-layer decides on the structure of objects on disk, and takes care of transferring objects from disk to main-memory. This also means that all objects are effectively ‘persistent’: objects only disappear when they are explicitly deleted. Even if you stop running Views, and come back later and restart it, the objects are still there.

5 Implementation model

The main implementation model is that in general there are ‘invariants’ between objects in the system (Figure 8). These invariants state that there is a direct relationship between the contents of an object and one or more other objects. For example, that the profit this year is the difference between income and costs. If an object gets changed (usually by the user editing it), the invariant goes ‘out-of-date’, and has to be re-instated, which is done by calling a related function.

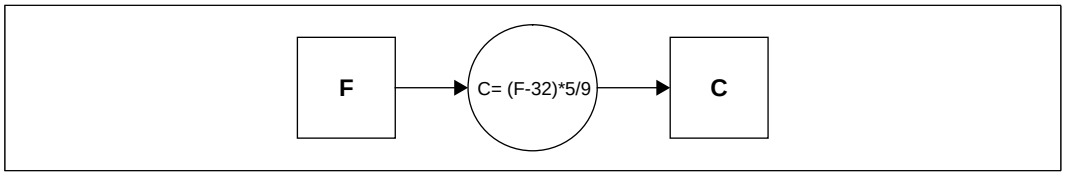

Figure 8. Invariants, here represented as a circle, link objects together. If an object is changed, the invariant is re-instated by changing linked objects. Here an object representing the temperature in degrees Fahrenheit is linked by an invariant to an object representing degrees Celsius. Since the invariants are two-way, if the user edits either one, the other gets updated automatically.

As another example, there is an invariant between the name of a file being displayed and the file itself, so that if the name gets edited the file must be renamed.

In general, the invariants are two-way, so that it doesn’t matter which objects in an invariant get changed. Higher-level invariants, defined in terms of user-defined functions, get broken down into a network of low-level invariants by the system (see Figure 9).

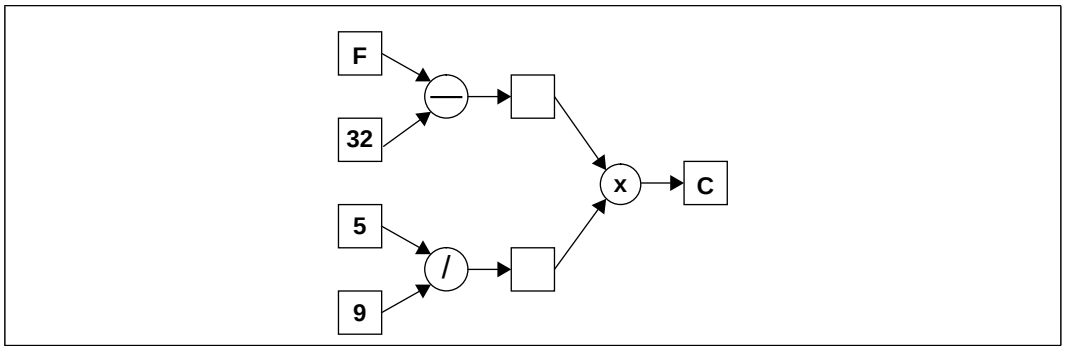

Figure 9. A high-level invariant, such as the one in Figure 8 is broken down by the system into a network of lower-level invariants

In fact, the invariant mechanism used very generally throughout the system, so that for instance, displaying objects on the screen is done by application of the invariant ‘the representation on the screen must match the object’: if the object gets edited, then the screen gets updated (see Figure 10.)

This means that the rest of the system can be completely oblivious of anything to do with output to the screen, or even that it occurs at all. In fact, all that an individual application sees is that its objects somehow change, and that it has to re-instate the invariants.

As a side-note, it should be mentioned that Views is not a full constraint-satisfaction system. As implemented, the invariants are equality constraints solved using local propagation (see [5]).

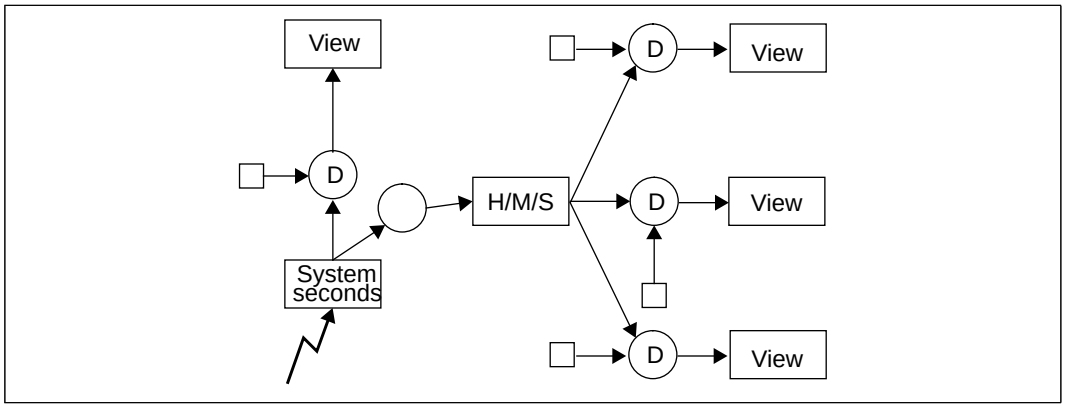

Figure 10. A simplified view of the internal connections of the objects displayed in Figure 7. The oblongs represent objects, and the circles invariants connecting them. If an object changes, any connected invariants are used to update connected objects. The system-time object gets changed automatically by the system each second. The diagram has been simplified to show the main relationships, but the display functions (marked with a D) for instance, are also connected to objects describing how the objects are to be displayed, and giving the details (size, etc.) of the windows in which they are to be displayed.

6 Open architecture

To add a new application to Views, the application designer has available a large collection of built-in types and objects, and types and objects created for other applications. If necessary, new objects can be defined: how they are structured internally, how they are to be displayed, and the relationships between the different objects in terms of invariants. The Views system takes care of the rest: input and changes to the objects, displaying objects, and ensuring that the invariants are kept up-to-date.

The fact that so little has to be done, means that producing a new application is very easy. The “Hello World” application mentioned earlier takes a single line of Views code; the graphical clock in Figure 7 takes a dozen or so lines (compared with several hundred for a traditional windowing environment).

The fact that the whole system is defined in terms of objects that are accessible and changeable means that many aspects of the system which are traditionally hardwired (such as menus, shortcuts, even how information is displayed) are tailorable to the user in Views, even on the fly.

As a small example, suppose that an application is required to display a list of numbers as a histogram. Amongst the tools available to the application builder is the ‘map’ operator `*`. This takes a function, and produces a new function that works on a list. For instance, if `list` is a list of numbers, `sqrt * list` produces a list of numbers whose elements are the square-roots of the corresponding elements in `list`.

This notation is particularly useful for Views, since if you specify that two objects are related by an invariant that includes a map, for instance that `list1 = sqrt * list`, the system splits the invariant into a number of smaller invariants involving each element of the two lists; in this way, if one element of `list` changes, only one element of `list1` need be updated.

For the histogram, we want to take a list of numbers and convert it into a graphical object. Among the graphical primitives is a function `box(h, w)`, which creates a box of height `h` and width `w`. What we want to do is create a number of boxes of constant width, and height proportional to the values in the list of numbers, and then stick them together. Creating the boxes is easy, we just use a map:

```
box(∇, w) * list
```

(The symbol ∇ means that that parameter will be filled in later.) This creates a list of boxes, which we must then stick together. The function `row` does this. So the complete specification of a simple histogram is then:

```
hist = row(box(∇, w) * list)
```

which specifies an invariant between the objects `hist` and `list`. Displaying `hist` will then show the boxes as a graphical picture. If the user edits the boxes (since graphical objects are just as editable in Views as textual ones), `list` will get updated to match, and *vice versa*.

7 Implementation

The basic Views kernel has been constructed. The idea is to build the system incrementally, so that at all times we have a running system, and to add applications one by one.

The window interface is built on the basis of an earlier product of the ABC project, STDWIN, a window management package that allows programs that use windows to be portable between different windowing systems [6]. For example, STDWIN, and thus Views, already runs on top of X-windows on Unix, on the Apple Macintosh, and on the Atari ST.

The basic display mechanism has been constructed, for graphical objects as well as textual ones, and the fundamental data-types and some primitive editing actions have been implemented.

The invariant mechanism, a central part of the system, has been built, and a start made on some applications, such as basic text-editing, file browsing, and message reading.

8 References

[1] Ernest Edmonds (ed.), *The Separable User Interface*, Academic Press, 1992, ISBN 0-12-232150-2.

[2] H. Rex Hartson and Deborah Hix, *Human-Computer Interface Development: Concepts and Systems for its Management*, ACM Computing Surveys, Vol. 21, 1, March 1989.

[3] L. Geurts, L. Meertens and S. Pemberton, *The ABC Programmer's Handbook*. Prentice-Hall, 1990, ISBN 0-13-000027-2.

[4] J. van de Graaf, *Towards a Specification of the B Programming Environment*, CWI report CS-R8408

[5] Wm Leler, *Constraint Programming Languages, Their Specification and Generation*, Addison-Wesley, 1988, ISBN 0-201-06243-7.

[6] G. van Rossum, *STDWIN — A standard window system interface*, CWI Report CS-R8817