The Design and Application of Visually Oriented Tools For Use During Software Development.

by Caroline Humphreys,
Computer Studies Dept., Loughborough University.

Focal Areas of Interest

Software Design, Programming Behaviour and Debugging Methods; Human Factors, Human Information Processing, Visual Perception and User Interface Design. Exploiting Beneficial Factors to Enhance Program Comprehension and Debugging.

I have always been interested in the ways and means of making programming and debugging easier, faster and more efficient; and in investigating the factors that affect them. Thus one of my primary goals is to design tools that emphasize and take advantage of the factors that aid in the programming and debugging tasks. Especially those that alleviate cognitive processing during the edit-compile-debug cycle.

Effects of Experience on Personal Perspective

I spent 3 years as a Software Engineer - designing, writing, testing and debugging Process Control software. Thus I have strong personal experience of debugging real-life software destined for actual customers, as well as noting how the other 5 Software Engineers in the department tackled similar tasks on different projects. I have also spent time, during the last 4 years devoted to research, observing the next generation of graduate programmers; and asking them about their attitudes to software tools and debugging methods. The views I express in this paper are a distillation of experience, observation, consultation and discussion with others programmers of varying levels.

The Programmer's Main Problem

As you can see from Fig. 1, the programmer usually only has the code and his own mental processes to aid in the program development and debugging tasks; since both activities require the comparison of what the program code should say (as defined by the programmer's own mental model of the code, and/or the code's specification) as opposed to what it actually does say/do in the editing environment. Program code has 2 principal interacting elements :

- declaration of various program and procedural variables; and
- statements and program constructs using these variable values.

There are 2 other factors associated with program text : visual appearance - the way that the code is laid out; and control flow - the order in which individual statements are executed at run-time. The former affects program comprehension, and the latter is determined by the input data processed, which in turn determines those bugs that are revealed. Testing only gives evidence of bugs that are present, but it may not reveal all bugs. Debugging is the art of eliminating bugs, without introducing others.

Principle Aims of Research

Catering for the solo programmer who only has access to the minimum programming toolbox, consisting of a screen editor (or editing environment like MacPascal) and compiler. Designing tools that fit the programming task, and programmers' needs more closely than existing tools; filling the gap with new tools and/or "extrapolating" existing tools and concepts.

- Using typographic effects to focus visual attention and alleviate those programming tasks dependent on visual processing (spotlighting).
- Reducing information processing burdens by providing essential information in alternative formats (summary tables/menus).
- Supporting individual aesthetic requirements (layout aids).

In effect, increasing programmer satisfaction and productivity by reducing the frustration and mental burdens created by the inadequate tools provided for the programming task.

NB. Spotlighting could be applied to other forms of electronic text, whereas summary tables and layout aids refer mainly to procedural programming languages, such as Pascal, which I have used for demonstration, since it is my preferred language. I'm not sure whether they could be applied to logical languages with any beneficial effects!

Layout aids

Provide a selection of alternative layout patterns for each programming construct, and let each user define his or her own matrix of layout patterns, and the number of spaces for indenting each level. This would enable any program to be laid out according to personal preferences. Thus enhancing comprehension and visual (as well as mental) rapport with that program.

Layout factors : indentation and the relative disposition of programming construct sub-elements, such as if-then-else (3 sub-elements)

Could only alter indentation - leaving all other aspects alone, except for overflow lines (eg. complex conditions occupying 2+ lines) - thus placing of comments and overall layout pattern would retain its original features

Placing of AND & OR in complex conditional statements, reflects personal preference

Summary Tables

Provide alternative views of all (or selected) data declarations within the program. Possible ordering permutations include : original (ie. in order of declaration), alphabetic, by data type, by parent procedure or function. For example, a list of all variables within procedure "average" alphabetically.

Summary tables are intended to answer questions of the class: "What has variable X been used for and where?".

Summary aid as an on the spot reference so you can get it right the first time. Summaries of all data definitions used as information tables or as selection menus. User control of updates for spotlighting and summary tables means removal of problems of deciding when to update.

Summary tables - reserved word list could be useful when you need to refresh your memory after previous working with another language or lull from this one.

Spotlighting

Provides "automatic" highlighting of any given word throughout an electronic text, using inverse video or colour. Thus spotlighting enables the user to see "all" instances of the selected word situated within the current screen window, at one glance.

It is also be useful to know how many instances of the given word there are all together, and to know which "position" the "current" word holds.

The use of such a "current instance/total instance count" indicator could be used as a strategic (planning-wise) and/or pure orientation aid.

Spotlighting implies additional "movement" commands, such as, go to 6th spotlight, or go forwards (or backwards) 4 spotlights.

Debugging

Most bugs can be associated with a specific variable. Thus the simplest way of finding the bug is to examine all statements involving that variable.

This technique is called variable trail following, and is usually achieved using the search mechanisms. I hope to demonstrate that Spotlighting makes variable trail following much easier to do, and hopefully less frustrating!

Search Mechanism Principles

Existing principle - "sequential, show/visit only one at a time, top-down search"

Spotlighting principle - "random access, show all at once, visit any using forwards and/or backwards search"

Spotlighting & Debugging

Seeing all instances at the same time has many advantages :
- you can quickly pick out the best place to start looking;
- freedom of movement : you can employ forward or backward search as needed;
- you can check individual statements in the instance sequence;
- you don't have to keep a mental map of "instance locations", as it is provided for you; thus cutting down on memory load;
- using 2 spotlights as boundaries, you can investigate intermediate statements without "getting lost".

Correlating Spotlighting & Summary Tables With Errors

Common Errors During Coding	Spot	Summ
undeclared variables	y	y
redundant variables	y	n
misspelt variable names	?	y
infinite loops	y	n
redundant loops	?	n
inappropriate initialisation or modification of variable values	y	n
incorrect sequencing of "dependent" variable assignments	y	n
missing/mis-matched comment brackets	y	n
incompatible format & content of procedure parameter lists	n	y
inappropriate passing/return of variable values via proc calls	y	y

• undeclared variables - if a variable has not been declared then its spotlight will appear in the main body of the procedure, but not in the declaration area - thus it can be detected by omission.

• redundant variables - the only spotlight will appear in the declaration area

• misspelt variable names - either you can detect this by checking what did not get spotlighted, or by comparing the declared names list with the undeclared names list.

• infinite loops - terminating condition of the loop is never met, either the exit condition is incorrectly defined, or the variable(s) that triggers the exit condition has not been modified correctly within the loop

• redundant loops - the initial condition of the loop is never met, either the entry condition is incorrectly defined, or the variable(s) that triggers the entry condition has not been modified correctly beforehand

• missing or non-initialisation of a variable before use - spotlighting this variable's trail, and tracking it forwards or backwards should soon give an idea of where the missing initialisation statement should go

• inappropriate initialisation or modification of variable values - putting the spotlight on a variable, and then checking each of the assignment statements, quickly points you to the cause of the error. In some cases the wrong operator or function is applied, and in others the wrong (variable) value has been fed into the equation.

• incorrect sequencing of "dependent" variable assignments - in the simplest case, one variable's value is modified before being fed into the equation that modifies another variable (or itself). Getting these assignment statements in the wrong order can cause all sorts of trouble

• missing/mis-matched comment brackets - if you arrange to spotlight anything that appears between contiguous '{' and '}' symbols, disregarding any surplus '{' symbols, then you should only spotlight "comment text", but if a '}' symbol has gone missing, then it will be obvious where, since all intervening comment and program text will be put into inverse video

Combining spotlighting and summary table methods

Knowing which names/words are undeclared makes it easy to choose which to spotlight.

Comparing the undeclared list with the declared list(s) should point out any discrepancies in spelling eg. declaring "time", and misspelling it as "ttime", "yime" etc. in the text.

Comparing Different Typographic Effects on the Signal Variable

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;
begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Total time=', time, 'secs');
  writeln('No. of vehicles=', vehicles);
  writeln('Max wait=', maxwait, 'secs');
end.
```

Italicising

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;
begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Total time=', time, 'secs');
  writeln('No. of vehicles=', vehicles);
  writeln('Max wait=', maxwait, 'secs');
end.
```

Emboldening

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;
begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Total time=', time, 'secs');
  writeln('No. of vehicles=', vehicles);
  writeln('Max wait=', maxwait, 'secs');
end.
```

Underlining

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;
begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Total time=', time, 'secs');
  writeln('No. of vehicles=', vehicles);
  writeln('Max wait=', maxwait, 'secs');
end.
```

Reverse Video

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;
begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span=', time, 'secs');
  writeln('Vehicle-count=', vehicles);
  writeln('Max-wait=', maxwait, 'secs');
end.
```

Fig 1 Plain text

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;
begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span=', time, 'secs');
  writeln('Vehicle-count=', vehicles);
  writeln('Max-wait=', maxwait, 'secs');
end.
```

Fig 2 Signal variable

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;
begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span=', time, 'secs');
  writeln('Vehicle-count=', vehicles);
  writeln('Max-wait=', maxwait, 'secs');
end.
```

Fig 3 time + vehicles

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;
begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span=', time, 'secs');
  writeln('Vehicle-count=', vehicles);
  writeln('Max-wait=', maxwait, 'secs');
end.
```

Fig 4

time + vehicles + signal

Spotlighting Different Variables

Siddiqi's (1985) signal problem (Program designer behaviour, People & Computers 1, p377) is stated as follows :
A traffic survey is conducted automatically by placing a detector at the road side connected by data-links to a computer. Whenever a vehicle passes the detector, it transmits a signal consisting of the number 1. A clock in the detector is started at the beginning of the survey, and at one second intervals thereafter it transmits a signal consisting of the number 2. At the end of the survey the detector transmits a 0. Each signal is received by the computer as a single number (ie. it is impossible for two signals to arrive at the same time). Design a program which reads such a set of signals and outputs the following :
(a) the length of the survey period;
(b) the number of vehicles recorded;
(c) the length of the longest waiting period without a vehicle.

Fig 1 shows a complete, commented solution to Siddiqi's signal problem - this can be used for reference and comparison of the subsequent partial solutions, and the variety of errors that spotlighting emphasizes in each case.

```
program survey(input, output);
var
  signal : 0..2;
   { 0 indicates end of survey period,
     1 indicates another vehicle has passed the detector,
     2 indicates another second has passed. )

  time, { length of survey period in seconds )
   vehicles, { no. of vehicles detected so far )
    wait, { time in seconds since last car was detected )
     maxwait : integer; { maximum waiting period so far )

begin { initialise )
  time := 0;
  vehicles := 0;
  wait := 0;
  maxwait := 0;
  repeat { read and process signals until end of survey period )
    read(signal);
    if signal = 2 then { another second has passed, so increment time counters )
      begin
        time := time + 1;
        wait := wait + 1;
        if wait > maxwait { adjust maxwait to new maximum wait value )
          then maxwait := wait;
      end;
    if signal = 1 then
     { a vehicle has passed, so reset wait counter, and increment vehicle count )
      begin
        wait := 0;
        vehicles := vehicles + 1;
      end;
  until signal = 0; { end of survey period )
  { Print out required data )
  writeln('Length of survey period is ', time, 'secs');
  writeln('No. of vehicles recorded is ', vehicles);
  writeln('Longest waiting period is ', maxwait, 'secs');
end.
```

Final Solution to Signal Problem

In Fig. 2, the signal variable has been spotlighted - this shows up a variety of associated bugs. As can clearly be seen, (with/without referring to the complete solution in the Appendix) there is no declaration of the signal variable. Also, the "read(signal);" statement is on the wrong line - it should be the first statement inside the repeat loop - as it is the repeat loop forms an infinite loop (unless the first signal value is 0).

Fig 3 : If the time and vehicles variables are spotlighted together, then it is easy to check that each of the variables is incremented in the appropriate sequence, and that they are independent from each other.

Fig 4 : If the signal, time and vehicles variables are spotlighted together, then this makes the sequencing dependencies even more obvious, and subsequently easier to detect.

Thus, multiple spotlighting can be used to check for dependence between the selected variables.

Figures 5 & 6 show a slightly different (partially developed) solution to the signal problem, where some comments have been added and the code that deals with the timer variables (time, wait and maxwait) has been made into a subprocedure called from within the main program loop.

Fig 5 shows the effect of spotlighting, when the global (main program) variable "wait" is selected - the declaration, initialisation, re-initialisation and procedure call statements involving "wait" have all become highly visible. However, the "wait" variable statements in the subprocedure remain camouflaged, because they are associated with the local "wait" variable belonging to procedure inc_timers, which is not the same as the global (main program) variable of the same name. If the procedural parameter list for inc_timers had not included the "wait" variable, then the references would have referred to the global variable (in this particular case) and the spotlighting would have emphasized these instances of the "wait" variable as well.

Fig 6 shows the effect of spotlighting when it is applied to the task of matching comment brackets - that is spotlighting all text that occurs between contiguous '{' and '}' symbols. It is obvious that the '}' symbol is missing from the "main program" comment, since all the following statements have become spotlighted, until a matching '}' is found, terminating the next comment.

Thus, the brevity of the examples give an indication of the interpretational power afforded by spotlighting - however, it must be remembered that in longer texts, this power will increase as the (potential) number of selected item instances increases. If the selected item has a low density (few instances within a large chunk of text), then it becomes increasingly easier, especially with unassisted visual scanning, to overlook some instances. The same is true for high densities, where the same effect occurs due to information overload and confusion between successive statements (Card et. al. 1983).

The spotlight effect could also be used as a memory jogger, to guard against uncompleted variable name changes eg. changing 'i' to 'index' but not checking that all appropriate changes have been made. This would be particularly useful where the scope of a variable extends across a large section of text, with a "blank area" in the middle. For example, where a variable is spread across 3 screen "pages", occurring on the first and third pages, but not on the second page. The wider the gap - the more useful the reminding effect.

One of the most frequent errors is the undeclared variable error and also the non-initialisation error. These can quite easily be picked up with spotlighting. In the case of the undeclared error, if you spotlight the variable name that hasn't been declared, then it won't appear in the declaration list. It will appear throughout the program or procedure text, but it won't actually occur in the declaration, so you can detect that by omission.

A simpler way, of course, of detecting undeclared variables is to compile a list of all the different variables and which procedures they belong to. Then any variable or any word which does not occur in the declared variables list, or is not a reserved word or reserved procedure/function name is obviously undeclared, and you can note it that way.

For the uninitialised variable error, all you have to do is to spotlight the required variable and it will appear within the text, and then all you need is to check where the first use of this variable appears, and decide where to put the initialisation statement, just before it (the program counter) gets to that point.

The next tool I conceptualized is the summary menu system - an automatic data dictionary inventory, that is viewable from different perspectives. The purpose of this tool is to collect all the different variable names, that you've declared throughout the program text, and to arrange them in different ways so that you can see them at a glance, by looking at the summary menus. So, for example, you could call up a variable name, and see which procedures it appeared in, because sometimes you use the same variable name, and just pass it across as a parameter. Or perhaps if it is just a simple counting variable, you might use the same variable name across different procedures for simplicity.

(That's also an interesting point, because programmer's have their own pet names for counting variables. I tend to use i and j for my counting variables in "for", "while" and "repeat-until" loops - anything which needs an intermediate incrementer which is simple and of no particular importance.)

Having a declaration list available as a menu is very useful, because that way you can look up whichever variable name you're interested in and check the data type to make sure that you're using the right functions and operators to manipulate it, and also to check that if you're modifying a value and passing it to another variable, that it is assigned to a variable of the correct data type. For example if you create or modify a real value on one side of an assignment statement, and is assigned to an integer variable on the other side, then you are going to lose value across the operation, because a real value will truncate to an integer value.

Some people would argue that you don't need a summary menu tool, because you've got the declaration list. Well that is true, but why should you expend time and (mental???) energy scrolling back to the declaration area, and searching through for whatever variable you're interested in. It's much simpler just to call up the chosen variable name on the summary menu and have it tell you what it is. That way there are no errors such as you think it's one data type and then finding out much later (when debugging perhaps) that it is something else. Also summary tables provide other possibilities for checking. For example you can find out which other procedures use the same variable name, and if they are declared as the same or different data types. Perhaps in one procedure you've declared it as an integer, and in another you've declared it as a real data type. Now you may have done that on purpose, or you may have wanted them both to be of the same data type, and this way it is much easier to check that you are using them consistently, or to your required plan.

All these things that I'm suggesting are ways of making the actual programming and debugging tasks easier for the programmer, because there is such a lot you have to remember, and obviously the more you have to remember and deal with, the more mistakes you are going to make. Obviously, anything that is going to make the actual burden lighter is a bonus, to be wished for.

Another very useful mechanism, I think, for the summary table system to work on is the user-defined procedure names and their parameter lists themselves so that for example, a quite common fault is for programmer to get the parameter list that goes with a procedure call wrong, or alternatively to get the order of the parameters themselves mixed. This way you can call up the procedure name that you're interested in, and then also call up it's parameter list as it was declared originally. That way it makes it absolutely clear which parameters are variable and which are actual parameters, and which type each one is. That way there should be no problem in assigning the right variables into the procedure call's parameter list - which again is a useful thing.

The foregoing applies to the user's own procedures, but it is perhaps even more useful for the predefined procedures/functions which you are not familiar with, when you need to find out the parameter list. Usually to do this you have to go and look at the manual, which is a chore that nobody likes doing. So in this case it's much simpler to call it up on the menu, find out it's parameter list and then just fill it in - instead of having to go through the aggravation of getting the manuals out and trying to find out more about the required procedure. Also it should be error free, because you will have all the procedural parameter list there and possibly if its a predefined procedure there may even be some additional information on the actual use of the procedure. This all goes to making life easier for the programmer.

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;

  procedure inc_timers
    (var time, wait, maxwait : integer);
  begin
    time := time + 1;
    wait := wait + 1;
    if wait > maxwait
      then maxwait := wait;
  end;

begin { main program
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat { process signal }
    if signal = 2 then
      inc_timers(time, wait, maxwait);
    if signal = 1 then
      begin
        wait := 0;
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span=', time, 'secs');
  writeln('Vehicle-count=', vehicles);
  writeln('Max-wait=', maxwait, 'secs');
end.
```

Fig 5 : Wait (global variable only)

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;

  procedure inc_timers
    (var time, wait, maxwait : integer);
  begin
    time := time + 1;
    wait := wait + 1;
    if wait > maxwait
      then maxwait := wait;
  end;

begin { main program
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat { process signal }
    if signal = 2 then
      inc_timers(time, wait, maxwait);
    if signal = 1 then
      begin
        wait := 0;
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span=', time, 'secs');
  writeln('Vehicle-count=', vehicles);
  writeln('Max-wait=', maxwait, 'secs');
end.
```

Fig 6 : Matching Comment Brackets

Fig 7

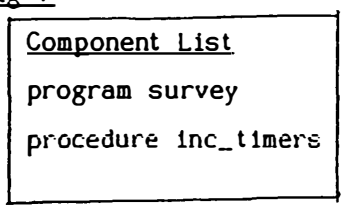


Fig 8

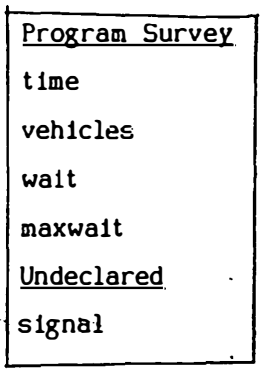


Fig 9

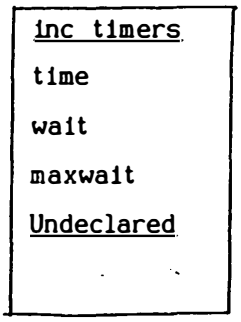


Fig 10

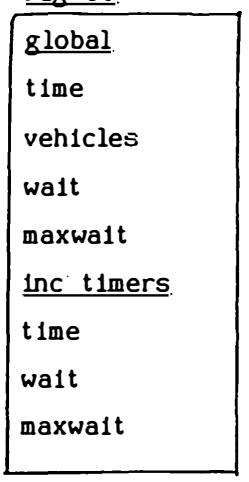


Fig 11

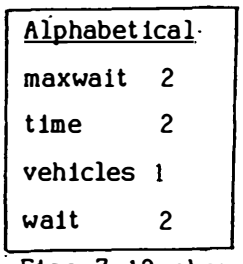
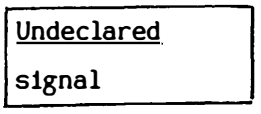


Fig 12



Figs 7-12 show summary lists that could be produced after interpreting the structure produced through interrogation of the program text of Figs 5 & 6. Fig 7 shows the component list - the full list of all named procedures and functions, including the program name. Selecting a name shown on the component list would cause the associated child lists to become available - either in declaration or alphabetical order; with or without the associated undeclared variable lists.

Thus Fig 8 results when selecting the declaration ordered variable list of "program survey" from Fig 7; and Fig 9 results when selecting the declaration ordered variable list of "procedure inc_timers" from Fig 7. Note that the lower portion of both Figs 8 & 9 is devoted to undeclared variables.

In contrast, Fig 10 defines the list of declared variables that are accessible and can be used, in terms of global and local variables, when seen from within inc_timers.

Fig 11 shows the (entire) alphabetical list of variables declared throughout the program. Notice that each variable is associated with a number, if it is declared more than once - selecting any individual variable name would cause a list of its "parental" procedure names (denoting declaration origin), to pop up, with or without an accompanying definition of the variable's type status (depending on the viewer's requirements).

There are 2 ways of doing the entire declaration list - either listing all declarations and allocate them as given, and list all undeclared item separately in a "floating" list; or list everything in association with its parent list, noting declared items first and undeclared items second, so it is easy to tell where each item appeared, and hence to allocate it.

Bibliography

Cakir A, Hart D J & Stewart T F M
Visual Display Terminals
John Wiley & Sons 1980

Card S K, Moran T P & Newell A
The psychology of human-computer interaction
Lawrence Erlbaum, 1983

Davies S P
Skill levels and strategic differences in plan comprehension and implementation in programming
People and Computers V, 1989, p487-502

Green T R G
Programming as cognitive activity
in Human Interaction with Computers, edited by Smith H T & Green T R G
Academic Press : London, 1980

Green T R G, Sime M E & Fitter M J
The art of notation
in Computing Skills and the User Interface, edited by Coombs M J & ALty J L
Academic Press : London, 1981

Hulme
Extracting information from printed and electronically presented text
in Fundamentals of human-computer interaction, edited by Monk A
Academic Press, 1985, 35-42

Monk A
Personal Browser
Interacting with Computers 1(2) Aug 1989

Siddiqi J I A
Program designer behaviour
People & Computers I, 1985, p369-379

Suchman L
Plans and situated actions : the problem of human-machine communication
Cambridge, 1987

Thompson P
Visual perception : an intelligent system with limited bandwidth
in Fundamentals of human-computer interaction, edited by Monk A
Academic Press, 1985, 5-33

Treisman A
Perceptual grouping and attention in visual search for features and objects
Journal of Experimental Psychology : human perception and performance, 1982, 8(2) 194-214

van Laar
Evaluating a colour coding support tool
People and Computers V, 1989, p215-230

van Nes F L
Space, colour and typography on visual display terminals
BIT 1986, 5(2) 99-118

Watkinson N S
The evaluation of dynamic human-computer interaction
Unpublished Ph.D. Thesis, Computer Studies Department, Loughborough University, 1988.

Winfield I
Human Resources and Computing
Heinemann : London, 1986

Wright P & Lickorish A
Colour cues as location aids in lengthy texts on screen and paper
BIT 1988, 7(1) 11-30