General Requirements for a Program Visualization Tool
to be Used in Engineering of 4GL-Programs

Ludwig Coulmann
Technical University at Darmstadt, Germany
Unit Programming Languages and Compilers
and
Software AG, Darmstadt, Germany

Abstract

Program visualization, the two-dimensional graphical representation of a traditional textual program can be used profitably as an aid to help a programmer gain an understanding of the program's meaning and function. In our context this process is called program analysis. *The paper first discusses characteristics of program analysis. It points out that program analysis is highly individual and is influenced by the person involved and by the aim of the process. Secondly, the paper describes what consequences evolve out of the characteristics for a tool which supports program analysis. Five distinct, general properties of a visualization tool are presented, emphasizing the user and his changing interests. Thirdly, concepts are given for the visualization of 4GL-programs and a specific tool is described as an example of how the outlined requirements translate to a real application. A tree is used to represent different structural relations in the program and icons at the nodes facilitate the tree's perception. Finally a discussion of the presented concepts completes the article.*

1. What Program Visualization is Used for

Often it is necessary for a software developer to review an existing old program. Reading source code is very laborious and time consuming because the developer has to follow someone else's thoughts and models. This paper provides concepts for a visualization tool to assist him in understanding the old program. A visualization tool can aid the process of becoming familiar with an existing program by adding a graphical representation to its textual formulation, clarifying its structure and the correlation of its components. Visualization is done with the main purpose of helping the user understand a program better, faster, and more easily.

In the broadest sense the issue can be assigned to the area of software reverse engineering; more precisely (i) to restructuring, which is the transformation from one representation of a program to another one on the same level of abstraction and (ii) to redesigning, the derivation of a higher level of abstraction from an existing representation (Chikofsky, Cross, 1990). The concept of program visualization as it is presented here supports the reverse engineering process in the phases of information collection, analysis, and representation of a program. This support is automated so that a new representation of the program can be generated without manual aid through the programmer.

Here I describe the main requirements of a visualization tool. This paper is an excerpt of Coulmann (1992) which also gives a general introduction to program visualization and available tools and which discusses implementation considerations. Coulmann (1993) presents a detailed description of the visualization tool "NATURAL Visualizer" (NV), which is a concrete example for the concepts presented herein.

The concepts are generally valid for all textual programming languages. An example of how their realization might look follows (sect. 4). It is based on NATURAL, a fourth generation data base access language mainly used for business applications developed by the German software company Software AG, Darmstadt.

According to the classification of program visualization by Myers (1990) the presented concepts are a static visualization of program code.

2. Characteristics of Program Analysis

To show the issue of discussion we give an example (figure 1) which will be explained below in further detail.

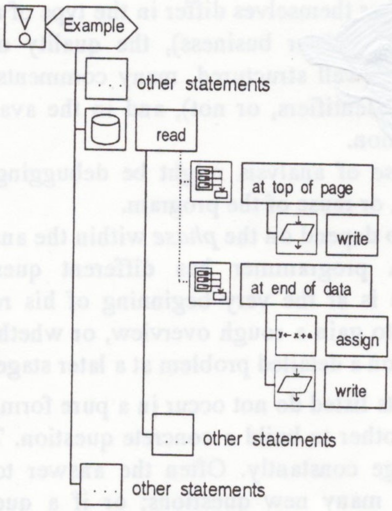


Figure 1. Example of Visualization (Output Aspect)

With the term *program* I refer to all source code belonging to an application system. Normally it consists of many distinct modules and might extend to several hundred pages of program listing. The programmer's task of working through a program to gain an understanding of its functions is a process referred to as *program analysis.*

To find general requirements for a program visualization tool I first describe the main characteristics of program analysis. As will be seen there is no generally valid method according to which a program is analyzed.

2.1. Programs are Analyzed Under Different Aspects

While analyzing a program a programmer directs very diverse questions to the program he is working on. I call these different formulations of questions *aspects.* The different aspects can be categorized as the global aspect, functional aspects, and data aspects.

- The *global* aspect deals with the relationships among the different components of the program. Which components constitute the system? How do they work together?
- The *functional* aspects relate to the functions a program executes. Which functions exist? How are they grouped to modules? What is the purpose of a single function? The flow of control also has to be questioned here, i.e., in what sequence are functions executed?
- The questions regarding *data* aspects include: Which data structures exist? What is the purpose of a single variable? Where, how, when, and why is a variable used? What is the data flow like, i.e., how is the data propagated among the variables?

Which aspect is relevant at a certain time depends on different factors. Some of the factors are the programmer, the program itself, the analysis purpose, and the phase within the analysis process.

- The *programmers* differ, e.g., with respect to their experiences, knowledge about the program, and their working style.
- The *programs* themselves differ in the type of application (technical or business), the quality of the source code (well structured, many comments, self explaining identifiers, or not), and in the available documentation.
- The *purpose* of analysis might be debugging, enhancement, or reuse of the program.
- Factors also depend on the *phase* within the analysis process: A programmer has different questions whether he is at the very beginning of his review and wants to gain a rough overview, or whether he follows up on a detailed problem at a later stage.

All of the aspects listed do not occur in a pure form, they mix with each other to build a concrete question. These questions change constantly. Often the answer to one question raises many new questions; or if a question cannot be answered at the moment the programmer turns to another one.

2.2. The Aspect Determines What is Relevant

When a programmer analyzes a program he is only interested in the topic actually relevant for his present thinking. There are elements in the program which help him to answer his question and others which do not contribute at all. The category an element falls in depends on the aspect under which he is reviewing the program. Considering the fact stated above that program analysis is done under changing aspects, it follows that each element can be relevant at a certain time and be irrelevant during another analysis phase.

An example that demonstrates this fact is taken from the NATURAL programming language. The same piece of source code (see below) can be visualized in two completely different ways.

```
0200 READ EMP-VIEW BY NAME
0210   AT TOP OF PAGE
0220     WRITE "EMPLOYEES BY NAME:"
0230   END-TOPPAGE
0240   PERFORM PRINT-EMPLOYEES
0250   AT END OF DATA
0260     COMPUTE AV = AVER(SAL)
0270     WRITE "AVERAGE SALARY:" AV
0280   END-ENDDATA
0290 END-READ
```

One aspect might be the *output of the program*, a specialization of a functional aspect. To figure out how the report is generated, the programmer might be interested in the statements event-driven executed[1] like AT TOP OF PAGE and AT END OF DATA. A visualization which represents this aspect is already shown in figure 1.

Another aspect might be the program's component structure. Therefore the programmer might want to see a call graph that shows which component is called by which component. This aspect could be visualized as in figure 2.

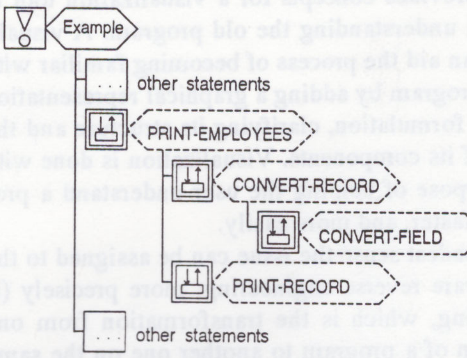


Figure 2. Example of Visualization (Call Graph Aspect)

[1] In NATURAL-terminology "non-procedural" executed

2.3. Program Analysis Means Searching and Navigating

The process of program analysis is a permanent search for certain locations and therefore also a permanent navigation through the program. One asks questions like: What is happening with that variable? Where is that function called? What is happening after that function call? To find answers, the programmer has to search for distinct locations in the program and has to move around within the program. Often the formulation of the questions is not absolutely precise. The programmer hopes to find the answers just by reading the program. The locations which interest him might be widely separated throughout the program, spread over modules.

The precise formulation of a search criterion is often very difficult as only incomplete information is available. One only knows approximately what to search for. For example: the problem of formulating exactly the name of a variable or of a group of variables. It is true that modern word processors or development environments provide comfortable search functions which allow some kind of imprecision (wild characters, regular expressions), but the formulation of the search criterion is always limited to a textual manner. A query "show all locations in the program where variables of data base file X are used" cannot be expressed in that way. But a programmer has many such questions which cannot be formulated by the search result if he wants to understand the program.

Another problem is related to the fact that the programmer has to consider many different modules at the same time. Utilities like the UNIX *grep* command and cross reference lists generated by the compiler try to assist in this situation, but they do not look at the whole program with all its different modules as one unit and cannot represent it in a comprehensive way. To understand the correlation existing across module borders, the analysis must not be limited to single modules.

2.4. Complexity Must be Ruled

Programs are very large and their internal relations very diverse and hard to follow. Thus, program analysis is a difficult task. To understand a program with a hundred pages of source code but only three different variables and no conditions, branches, or loops is a job more for a busy beaver than for an egghead. A complicated program with many variables and different structures is easier to comprehend the shorter it is. However you will not find short programs in daily life. You normally will have long and complicated programs which are impossible for just one person to study and understand in it's entirety.

2.5. Program Analysis is an Individual Process

The process of program analysis depends on the person doing it. Every programmer has his own individual style formed by his own personality, experience, and knowledge (see above).

3. Consequences for a Visualization Tool

The characteristics mentioned in the previous sections require a visualization tool to meet certain properties in supporting the process of program analysis. Lucas (1991) has summarized these properties (translation by the author, emphasis added):

> Because of the high amount of relations contained in programs, it is mandatory that the programmer can request the information important to *him* according to his *interests*. When doing this he must not be distracted by other information which is *irrelevant at that time*.

For the visualization tool in mind the properties may be detailed as follows:

3.1. Property: Selective

To help rule the complexity the visualization tool must only represent a sector of the whole program, i.e., it must be selective. If the visualization tool represented a 1:1-projection of the source code into another more symbolic notation, the programmer would encounter the same problems of complexity he has when directly investigating the source code. All information irrelevant under a certain aspect must be omitted because it distracts and unnecessarily makes the program more complicated.

The requested selection might be to show one kind of information and not to show other kinds. This could be provided by different views of the represented program. The view corresponds to the aspect under which the program is investigated and commonly emphasizes a structural relation within the program. Possible views are flow of control, data flow, call or data structure, or input/output.

Another way to provide selection is to show only a part of the same kind of information; for instance only variables with certain common properties, such as those used by an investigated function. In contrast, other variables are not visualized because they are of no interest in the current formulation of the question.

3.2. Property: Versatile

On one hand the visualization must be restricted to selected parts of the program and on the other hand I have stated above that all program elements be of the same importance. These facts contradict each other. As a consequence the tool must be versatile. The versatility is achieved by providing *different* views and representation modes. The tool must be able to adjust to *varying* formulations of questions and *varying* methods of investigation.

Such methods might be: to work very close with the source code or with the graphical representation; to approach a program via variables and data structures or via the functional flow of control; to read a program from top to bottom to get a general view or first to separate it in its components and to investigate each one after another.

A tool supporting only one fixed format for viewing the program is not versatile. To some extent it forces the user to analyze the program in a certain way, because it provides only this one representation and therefore only one single aspect and method. Either the programmer adjusts his style of working to the prerequisites of the tool or it is of no use to him. In contrast, the currently investigated question should determine the way of representation and the user should be able to choose in what representation he wants to see it. One single appropriate representation which is always valid does not exist.

3.3. Property: Flexible

Because of the fact that program analysis is done under different and quickly changing aspects and that different programmers have different analysis styles, the tool must be flexible. As far as possible all settings and prerequisites of the tool should be easy to change. The flexibility must not cause the user to be left alone with all possible settings. Then the problem would be to find the most appropriate settings and views rather than to understand the program. Therefore, there should be default settings for the viewing modes from which the user can choose. Then he could employ the tool without a big effort to learn how to use it and he would not have to first develop concepts for his approach. However, default settings must only be suggestions for the user and must not limit his freedom, especially if he has used the tool before and already has some experience.

Flexibility also means not to limit the user to only one representation at a time. He should have the possibility of viewing several visualizations at the same time. For instance, he could then compare two different views or have a rough visualization next to a detailed one.

3.4. Property: Interactive

The fact that program analysis is done following changing formulations of questions leads to the requirement that the tool must be interactive. Analysis is a constant alternation between reading the visualization and changing the settings of the tool and scrolling through the representation. Like a programmer with only the source code available is always turning the pages to find the locations of interest to him, a user of a visualization tool uses it to navigate through the program. In addition, there is the possibility to adjust the mode of representation to the current formulation of question by changing the settings. So he only sees what is necessary for him. A static representation (e.g. Pretty Printing (Baecker, Marcus, 1990), Pascal/HSD (Diaz-Herrera, Flude, 1980), Greenprint (Belady, Evangelisti, Power, 1980), Delta (Thurner, 1990)) is not able to accomplish that. Nevertheless, it might be useful to make a hard copy of a representation in order to overcome the limitations of the small size of a computer screen.

An interactive tool requires fast response times or it will not be accepted by the user. *Fast* is a relative term and the performance depends on many factors. The amount of CPU time needed to produce a graphical representation from the source code must not be underestimated.

To quantify the time requirement I consider a response time of one second per source code page processed as sufficient.

3.5. Property: User Oriented

The tool stands or falls by its user interface. This seems to be a trivial statement because it is true for every application. Every application should be fast and easy to use and provide a lot of comfortable functionality at the same time. But when I consider how many applications exist which do not fulfill these basic requirements, I think it is worthwhile mentioning them anyway. In the area of software engineering, there are interesting ideas and methods to support the software development process (structure editors, programming using flow charts or block charts, visual programming, see Myers, 1990). The fact that no approach has been generally accepted yet is also caused by not considering enough a user's needs. A visualization tool faces the same risk.

The property "user oriented" means, for each function, a developer has to evaluate its usefulness for the user while designing a visualization tool. Which functions a tool offers must be directed by the users' requirements.

The consequence is that the user must be able to rule the required flexibility. Too many commands and settings expect too much of the user. If it takes a programmer six weeks to become familiar with a tool before he can employ it effectively, he will probably never employ it at all. A tool is good, if it is beneficial from the first moment on. The user can then learn the more complicated and powerful functions incidentally by using them.

For a tool to be accepted by the user, consistency is important. Not everything which is possible and nice, fits within the global concept. An additional improvement in functionality - when taken by it's self useful and logical – should only be added if it is consistent with the whole system. Otherwise there is the risk of causing more problems than help.

4. A Program Representation Through an Iconized Tree

As a concrete example of how the requirements mentioned above translate to a real representation, I describe a visualization of NATURAL programs[2] and the basic concepts of the visualization tool "NATURAL Visualizer" (NV). The following sections are only a rough outline. A detailed description can be found in Coulmann (1993).

4.1. The Tree

A program is represented through a tree. The outline of a tree structure was chosen because it provides a good general view and it is easy to comprehend. Furthermore trees can be created by fast algorithms. The following elements constitute the nodes of the tree: objects[3], statements, variables, and comments. The edges indicate the elements' hierarchy. For objects this constitutes the calling graph, for statements the nesting (separately for procedural and event driven statements), and for variables the data structure. In addition, the tree shows which elements use or depend on each other. The statements (represented by statement nodes) form the branches of the tree. Other objects appear as operands of statements. The object's statements constitute a subtree of its object node. This leads to a branching from the root to the leaves containing all statements and dependent objects of the root object. The source code represented by a node is visualized additionally by means of a detail box attached to the right side of the node. This provides a link between the visualization and the original source code. Ellipsis nodes indicate program parts not represented in the tree. These nodes appear wherever a code section is omitted in the visualization. They make the programmer aware of the fact that the representation is not complete. It also provides the opportunity to focus on a certain aspect of analysis and to hide program elements irrelevant to the current aspect.

The visualization example (see Figure 3) represents a fictitious program. It is supposed to provide an impression of how a NATURAL program might look, when it is visualized.

4.2. The Icons

Icons are the key visualization concept in NV. They provide the possibility of expressing a lot of symbolism in a little space through a small image. Therefore, all elements appearing in the visualization are provided with an icon which represents the element's type. The user recognizes "at first glance" what kind of element it is. Without icons this information would have to be given through text; but cognitive reception of textual information is more strenuous for the viewer than the perception of visual information. An alternative would have been to choose other symbols such as geometric shapes (rectangle, circle, and others, with differently shaped borders) to denote an element's type. But shapes are not nearly as expressive as icons and their meaning is harder for a user to memorize, because he does not associate anything with the symbols. Tanimoto (1987) contributed concepts and ideas on how to design icons. Figure 4 shows some examples of the icons used.

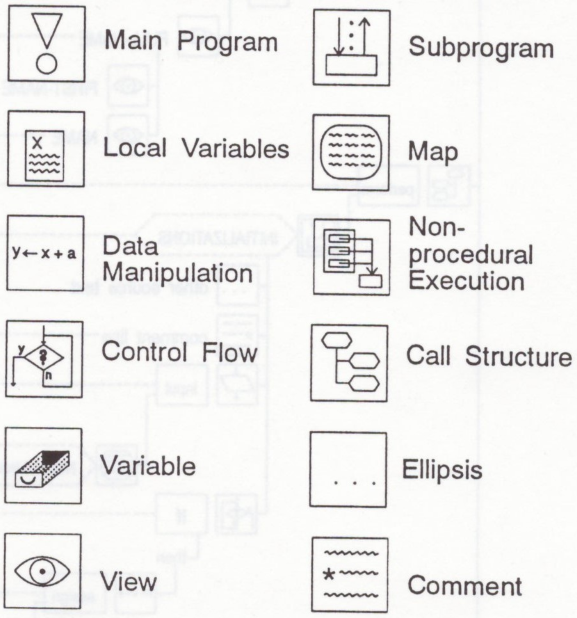


Figure 4. Icon Examples

4.3. The Actions

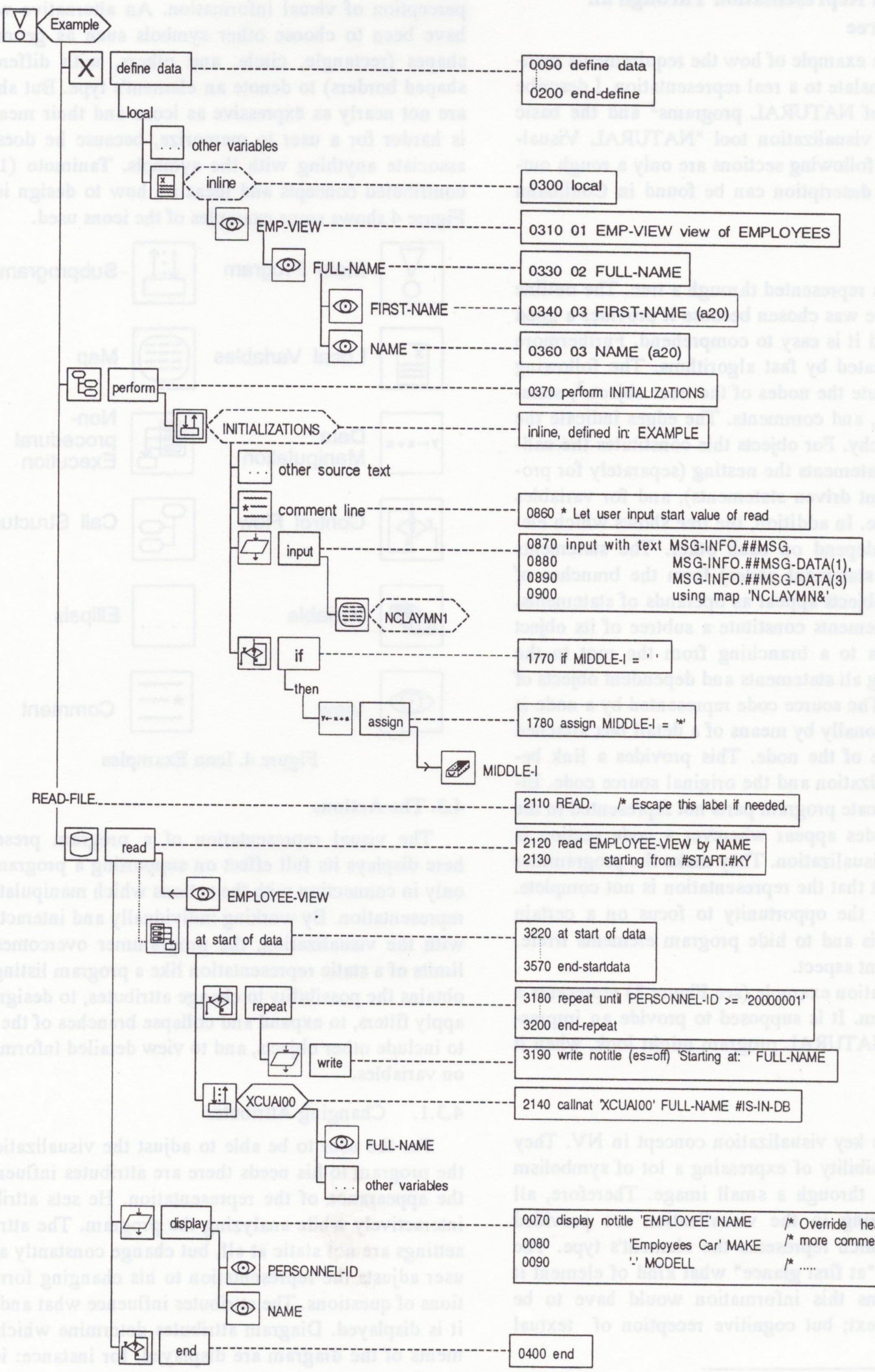
The visual representation of a program presented here displays its full effect on supporting a programmer only in connection with the actions which manipulate the representation. By working individually and interactively with the visualization, the programmer overcomes the limits of a static representation like a program listing. He obtains the possibility to change attributes, to design and apply filters, to expand and collapse branches of the tree, to include other objects, and to view detailed information on variables.

4.3.1. Changing Attributes

For the user to be able to adjust the visualization of the program to his needs there are attributes influencing the appearance of the representation. He sets attributes interactively while analyzing the program. The attribute settings are not static at all, but change constantly as the user adjusts the representation to his changing formulations of questions. The attributes influence what and how it is displayed. Diagram attributes determine which elements of the diagram are displayed, for instance: icons, detail boxes, operand trees, labels, the extent of the comments shown. In addition they decide on which objects and under which circumstances they are included in

[2] A complete description of NATURAL is contained in the NATURAL reference manual (see References).

[3] Object is the NATURAL term for module.



```
0090 define data
0200 end-define

0300 local

0310 01 EMP-VIEW view of EMPLOYEES

0330 02 FULL-NAME

0340 03 FIRST-NAME (a20)

0360 03 NAME (a20)

0370 perform INITIALIZATIONS

inline, defined in: EXAMPLE

0860 * Let user input start value of read

0870 input with text MSG-INFO.##MSG,
0880                 MSG-INFO.##MSG-DATA(1),
0890                 MSG-INFO.##MSG-DATA(3)
0900                 using map 'NCLAYMN&'

1770 if MIDDLE-I = ' '

1780 assign MIDDLE-I = '*'

2110 READ.     /* Escape this label if needed.

2120 read EMPLOYEE-VIEW by NAME
2130       starting from #START.#KY

3220 at start of data

3570 end-startdata

3180 repeat until PERSONNEL-ID >= '20000001'
3200 end-repeat

3190 write notitle (es=off) 'Starting at: ' FULL-NAME

2140 callnat 'XCUAI00' FULL-NAME #IS-IN-DB

0070 display notitle 'EMPLOYEE' NAME        /* Override  header
0080           'Employees Car' MAKE         /* more comments..
0090           '-' MODELL                   /* .....

0400 end
```

Figure 3. Visualization Example

the visualization. Statement attributes determine which statements or statement groups are displayed. Other attributes influence the representation of variables, session parameters, and system functions.

4.3.2. Applying Filters

NV provides filters to enable the programmer to concentrate on what he is currently interested in. If he selects a filter, he does not see the whole program, but only the aspects that can come through the filter. Filters are implemented by setting the appropriate visualization attributes. Each setting of the attributes constitutes a filter. This means that specifying filters is simple and manipulating them is easy.

4.3.3. Expanding and Collapsing Branches

To keep an overview of a large tree and to easily navigate through the tree, the user can expand and collapse single branches or subtrees of the tree. For instance by collapsing the whole branch of an object he is not interested in, he obtains a compact representation of what he is interested in. On the other hand, he can easily display parts that are hidden behind a collapsed branch by expanding it as soon as it becomes necessary for his analysis. A collapsed branch is indicated by an ellipsis node.

4.3.4. Including Other Objects

To regard a program consisting of many objects as a whole, NV can include all objects which depend on any of the displayed ones into the visualization. To include means to represent several different objects of a program within one visualization at the same time. They are displayed as object nodes with their statements constituting a subtree. A typical example is a main program calling subprograms. The object nodes for the subprograms appear as operands of the calling statements.

4.3.5. Information on Variables

To understand the meaning of a program it is important to know which variables are used in what way. Therefore a programmer wants to know where and how a variable is defined when he encounters it somewhere in the program. He can request this information easily for every visible variable through simple command no matter where it is displayed. A pop-up window will be displayed providing the background of the variable (where it is defined, what type it is, or what structure it is a part of, if it is a structure variable; also all comments appearing at its definition are presented). A programmer does not have to scan all modules to find this information.

5. Evaluation

In the following I describe how the required properties listed above are matched by the NATURAL visualization.

5.1. Selective ⇒ Filters/Ellipses

Filters give the possibility of selecting the kind of information the user is interested in; other information is filtered out. This allows the user to concentrate on the aspect relevant to his current formulation of question and he does not have to deal with the whole program and the full amount of complexity.

The mechanisms of collapsing and expanding branches and of setting attributes are also a means of selecting the desired information out of the whole. Ellipsis nodes allow the user to hide what he is not interested in.

5.2. Versatile ⇒ Different Aspects

By filters and settings of the visualization attributes, the programmer can create the appropriate views of the program to reflect the different aspects he has during the process of analysis. The tool is not limited to just one representation of the program and just one method of program analysis but can adjust to the versatile and changing needs of the programmer.

5.3. Flexible ⇒ Attributes

The visualization attributes provide an easy way to form the representation according to the user's requirements. Attributes can be set by just switching them on or off in a dialog box. Sets of attribute settings can be saved as a configuration file, so the user can create his own pool of visualization views. Filters are implemented as appropriate settings of attributes and therefore can be manipulated in a handy way.

5.4. Interactive ⇒ a Workstation Tool with GUI

NATURAL Visualizer is designed as an interactive tool which runs on a workstation with a graphical user interface. This provides the ease of use and direct response to the user's actions on the visualized program. If fast response times are achieved, they guarantee a convenient utilization of the tool, and the user will always obtain the desired representation of the program supporting his analysis efforts.

5.5. User Oriented ⇒ Individual Filters and Attributes

NV does not dictate any method or representation. Everything can be designed by the user to fit his individual demands. Filters and attributes are the flexible framework with which the user can build his personal and individual solutions.

6. Conclusions

The visualization presented here offers the possibility of both a detailed and a condensed representation. The detail boxes with the source code provide all the details necessary to comprehend a program in depth. By collapsing branches of the tree the programmer condenses the representation to the parts he is currently interested in. Because the visualization is not limited to a single object but displays all objects of a program with their correlation, the programmer obtains a complete and comprehensive representation of the program. In addition, the filters provide him with a means of only looking at certain aspects.

Because of these properties the user is supported while searching and navigating through a program. A comprehensive representation makes it easier for him to get an overview at the beginning. At a later time he can investigate specific detailed problems with a detailed representation. By collapsing everything he is not interested in, he can concentrate on his specific question without being distracted by other information. Therefore it is easier for him to become familiar with the program and he can more quickly develop an understanding of its meaning.

The flexible and interactive representation provides the user with the security to control the whole process of program analysis, because he can intervene quickly when the representation is not appropriate. He does not feel he is at the mercy of the tool and its representation of the information. He can adjust the representation according to his needs. The selection of information is more flexible as with pure textual search and is closer to the vague ideas a programmer has during program analysis. This concept matches tendency which also occurs in other areas. Often an application tries to make visible and controllable processes which perform automatically. With that the user gets the security of being able to intervene quickly when something unwanted happens. (Example: While copying files all filenames are displayed and the user has the possibility to abort the process before it is finished.)

There are two extreme possibilities to represent a program. In the area of visual programming some systems try to represent a program *only* through pictures and symbols. Examples are PICT (Glinert, Tanimoto, 1984) and HI-VISUAL (Hirakawa et al., 1987). In contrast traditional programming environments provide *exclusively* text oriented tools. The combination of a visual and textual representation in the presented concept combines these two extremes. It uses both the intuitive approach to a program through pictures and symbols and the precise and compact representation through text. Representations which try to be more abstract than the source code, can be created only with the aid of the user. Furthermore, such representations contain the danger to force the programmer to a view that might not be related to his specific needs. Therefore, a very close relation between the source code and its visual representation was chosen. Every node in the visualization is directly equivalent to a source code section.

I want to express my thanks to Software AG, specifically to Christian Lienert and Guido Falkenberg, for their support of this work. Prof. Hans-Jürgen Hoffmann provided me with important ideas. This is appreciated.

7. References

Baecker, R., A. Marcus (1990): **Human Factors and Typography for More Readable Programs.** Addison Wesley, Reading, MA.

Belady, L.A., C.J. Evangelisti, L.R. Power (1980): GREENPRINT: A graphic representation of structured programs. **IBM Systems Jounal, 19, 4,** 542-553.

Chikofsky, E.J., J.H. Cross II (1990): Reverse Engineering and Design Recovery: A Taxonomy. **IEEE Software,** January 1990, 13-17.

Coulmann, L. (1992): **Visualisierung von NATURAL-Programmen.** Konzepte und Implementierung als Prototyp. Diploma thesis, Unit Programming Languages and Compilers, Department of Computer Science, Technical University of Darmstadt, Germany.

Coulmann, L. (1993): **Programmvisualisierung bei Sprachen der 4. Generation.** Paper submitted to a conference.

Diaz-Herrera, J.L., R.C. Flude (1980): Pascal/HSD: A Graphical Programming System. **IEEE Proceedings COMPSAC 1980.** The Institute of Electrical and Electronics Engineers, Inc., 723-728.

Glinert, E.P., S.L. Tanimoto (1984): Pict:An Interactive Graphical Programming Environment". **IEEE Computer, 17,** 7-25.

Hirakawa, M., S. Iwata, I. Yoshimoto, M. Tanaka, T. Ichikawa (1987): HI-VISUAL Iconic Programming. **IEEE 1987 Workshop on Visual Languages,** Linköping, Sweden, 305-314.

Lucas, J. (1991): Ein Visualisierungswerkzeug für die Wartung modularer Programme. in J. Encarnação (Ed.): **Telekommunikation und multimediale Anwendungen der Informatik.** Proceedings 21st Annual Conference of the German Society on Computer Science (GI) 1991. Springer Verlag, Berlin, 405-414.

Myers, B. A. (1990): Taxonomies of Visual Programming and Program Visualization. **Jounal of Visual Languages and Computing, 1,** 97-123.

NATURAL 2.2 Reference Manual (1991). Manual Order Number: NAT-221-030. Software AG, Darmstadt.

Tanimoto, S.L. (1987): Visual Representation in the Game of Adumbration. **IEEE 1987 Workshop on Visual Languages,** Linköping, Sweden, 17

Thurner, R. (1990): Reengineering mit Delta. in H. Balzert (Ed.): CASE: Systeme und **Werkzeuge.** BI-Wissenschaftsverlag, Mannheim, Vienna, Zürich, 135-166.