An Investigation into Professional Programmers' Mental Representations of Variables

Jorma Sajaniemi
University of Joensuu
Department of Computer Science
P.O.Box 111, 80101 Joensuu, Finland
jorma.sajaniemi@joensuu.fi

Raquel Navarro Prieto
Universitat Pompeu Fabra
Estacio de la Comunicacio
Ocata 1, 08003 Barcelona, Spain
raquel.navarro@upf.edu

Abstract

Very little is known about professional programmers' mental representations of variables, yet this information is vital in designing effective tools for program comprehension. In order to find out what types of information programmers have about variables and their relations, we conducted a knowledge elicitation study where professional programmers studied programs and the resulting mental representations were elicited using card sorting and interviews. The mental representations were based on fourteen principles that can be organized in four main categories: domain-based, technology-based, execution-based, and strategy-based. Most frequent information types dealt with two execution-based criteria: behavior and utilization.

1. Introduction

Comprehension of programs encompasses the construction of a mental representation, which describes both the program as a technical artefact and its relationship with the application domain [18]. In the case of large programs, the mental representation must be partial and its construction is laborious. This task can be eased by tools which should fit programmers' mental representations and cognitive processes in order to be effective [17, 19].

Programs consist of variables, operations working on variables, control structures steering the execution of operations, and larger constructs—like functions, classes, modules, subsystems—giving a structure to collections of smaller constructs. A central task in program comprehension is to understand what a piece of code does: what is its purpose and how does the code achieve it. Code comprehension means dealing with variables and their tasks: understanding relations between variables and how operations act on them. In a study among professional maintenance programmers [18], information about variables was the most frequent information need. If code comprehension is to be supported by tools, mental representations of variables and their relationships must be known.

However, very little is known about professional programmers' mental representations of variables. Elementary textbooks treat variables as memory boxes holding a value that can be changed in arbitrary ways, and studies in the psychology of programming has seen variables as carrying out some unique task and having some unique data flow relations with other variables. But variables are not used in arbitrary ways and their tasks are not unique: the same tasks and update patterns occur in programs over and over again. It would be highly unexpected, if these tasks and patterns would not be part of programmers' mental representation. Yet, these tasks and patterns have been neglected in research.

In order to find out what types of information programmers have about variables and their relations, we conducted a knowledge elicitation study where professional programmers studied programs and the resulting mental representations were elicited. Research in the last years by Cañas and al. [4] has provided the theoretical and empirical bases for the utilization and interpretation of the results of knowledge elicitation techniques. After a series of experiments they concluded that relationship judgments among concepts were useful to gather knowledge about a user's conceptual model or information stored in the long term memory. Nevertheless, the information gathered is not a complete picture of the conceptual knowledge of the user, but rather a subset that depends on the information that is used for the particular task in the working memory, i.e., a short term store used to perform a task. Because of that Cañas and al. recommend to complete the information obtained from the users' judgments with other methods.

For knowledge elicitation we used card sorting, or grouping, task. This method has been used in a wide range

of areas to extract the mental knowledge of a specific set of people. In interaction design it is used to understand users' mental models [13], in expert systems design it is used to understand how knowledge providers conceptualize the domain elements [5], and in program comprehension it has been applied to understand the difference between the mental representations of experts and novices. In our case, we completed the information gathered with the card sorting task with interviews where the participants explained why they sorted the variables in the way they did. This qualitative information about reasons for grouping things together has proven to be valuable in understanding the criteria used by the participants [6]. Other examples of the use of interviews to study program comprehension is Pennington's work (e.g., [10]).

The rest of this paper is organized as follows. Section 2 provides background by reviewing previous literature on mental representation of variables. Section 3 describes the investigation with results and discussion in Section 4. Section 5 contains the conclusions.

2. Background

Very little is known about programmers' mental representation of variables. For example, Brooks [2, 3] suggests that knowledge about variables comprises its name, type (e.g., real or integer), and its purpose in the program, e.g, to keep track of elapsed time. A salient part of the purpose of a variable, and hence of its representation, is the relationship between a variable and the corresponding application domain concept. Brooks postulates a series of mappings between different levels of a solution to a programming problem: domain level, symbolic domain level, algorithmic level, and program level. In this theory, program creation and comprehension involve the construction of these mappings, and hence also the construction of variables and their corresponding entities within the other levels.

The idea of several related representation levels is central to later models of program comprehension, also. For example, von Mayrhauser and Vans [18] present an integrated program comprehension model containing several representations of programs, each at a different level: domain, program text, and intermediate. Variables belong to the program text level, and they exist in order to implement some need in the application domain level. The model contains also a knowledge base—retained in long-term memory—that describes, e.g., how programming goals can be achieved by specific code fragments. The detailed contents of plans is outside the scope of the model.

Pennington [10] describes several abstractions of programs possessed by expert programmers. Two of these concern variables: the data flow abstraction, and the conditionalized action abstraction. The data flow abstraction describes how the initial data objects given as input to a program are transformed into the outputs. The transformation creates a data flow through each variable; thus knowledge about this flow contains knowledge about data flow relationships between variables. The second abstraction, conditionalized action, is concerned with the states of a program: a certain state triggers an action, execution of the action results in a new state, and so on. A major part of the state consists of the values of all variables; thus conditionalized action knowledge contains knowledge about relationships between variables.

The knowledge types above concern unique variables in unique programs. They are part of program knowledge as opposed to more general programming knowledge that applies to theoretically all programs and is usually referred to as programming plans, or schemas (e.g., [8, 10, 18]). Plans represent stereotypic ways to achieve typical goals in programming. Ehrlich and Soloway [8] suggest that variable plans consist of such aspects as the variable's role in the program, the manner the variable is initialized and updated, and a guard that may protect the variable against invalid updates. As examples of roles they give counter variable, running total variable, and new value variable (that holds the newest number given as input in a loop). Rist [12] has further developed this idea and defines a plan as a set of actions that achieve a goal, and a goal as a state to be achieved, e.g., to calculate a value or a series of values. His basic plans deal with actions related to variables: prompt plan to obtain an input, label plan to produce output, running total plan to accumulate information, found plan to register some event, and loop plan to iterate using a loop counter variable. Neither Ehrlich and Soloway nor Rist claim that their lists of plans would be exhaustive.

The literature cited above has a strong cognitive basis and is supported by empirical experimentation. There are more practical suggestions to categorize variables, also. Green and Cornah [9] proposed a tool intended to clarify the mental processes of maintenance programmers. Among other features, the tool was supposed to reveal roles of variables listed tentatively as: constant, counter, loop counter, most-recent holder, best-of holder, control variable, and subroutine variable. Later, Sajaniemi [15] defined the role of a variable to depend only on the behavior of a variable (as opposed to its use) and found the following roles in novice-level procedural programs: constant, stepper (a generalization of counters), most-recent holder, most-wanted holder, gatherer, follower, one-way flag, temporary, and organizer. Ben-Ari and Sajaniemi [1] conducted an investigation demonstrating that computer science teachers can easily learn roles and assign them successfully in normal cases. However, the psychological existence of roles has not been studied before.

Somewhat related to roles is Hungarian notation [16],

which is a convention to encode information into a variable name about its type, quantity, and use. Hungarian notation is supposed to assist in remembering names, reading code written by others, and speeding name selection, but no studies into these effects have been conducted. Several convention schemes exist and they are used in professional programming.

In summary, programmers have been shown to have program dependent knowledge about variables, like their relationship with application domain entities and position in data flow, and general programming knowledge concerning variables, i.e., variable plans. The existence of knowledge about specific roles or information encoded in Hungarian notation is more speculative. The purpose of the current study was to investigate on a broad range what types of information expert programmers possess about variables.

3. Investigation

In order to elicit expert programmers' knowledge about variables a card sorting investigation was conducted. Professional programmers were instructed to get acquainted with short programs and then asked to sort all variables into groups based on their similarity. Participants were also interviewed about the sorting criterion they used and the interviews were audio recorded. The resulting groups and the interviews were then analyzed.

Participants: Thirteen programmers with a background between 3 and 24 years of professional programming (mean 13.7, mode 15) were recruited from software companies in the Joensuu region in Finland. All participants knew the programming language C well, had been programming in several procedural or object-oriented languages, and were unaware of the researchers' prior work in the area of programming knowledge. All participants were male. The participants or their companies were paid a small fee for participating in the investigation.

Materials: Five C programs having between 19 and 33 non-comment and non-blank program lines (mean 25.4, mode 22) were prepared. All programs had a different application domain, e.g., blood hormone testing or day number format conversion. The beginning comment of each program included an example of the execution of the program: input and corresponding output. Each program had an associated modification task; the tasks were designed to be simple and not to require the introduction of new variables. The materials can be found at http://www.cs.joensuu.fi/~saja/var_roles/materials/exp03/index.htm.

There was a total of 30 global scalar variables in the programs; all having the type `int` except a single `char`. Table 1 gives short descriptions of all the variables. In order to avoid too coarse granularity in the sorting task, participants

Table 1. Variables used in the investigation sorted by program.

Variable	*Description*
base	constant set at variable declaration
currDigit	numeric input read repeatedly
largePowerOfTwo	constant set at variable declaration
numericValue	collects a number from single digits given as input
powerOfTen	controls for-loop; divided by 10 on each round
powerOfTwo	controls while-loop; divided by 2 on each round
smallDigit	smallest value so far in a number series generated in a loop
closest	input value so far closest to a constant
count	descending while-loop counter
maxDelta	greatest difference between two consecutive input values found so far
norm	constant set with an assignment statement
testRes	numeric input read repeatedly
yesterday	previous input
dayNbr	data source generated repeatedly by a random number generator
daysAtEnd	sums up numbers given in an array
daysToBeginning	the previous value of `daysAtEnd`
month	keeps track of the number of rounds in a loop
command	single character input ('i' for incoming ship etc.) read repeatedly
lastShip	previous value of specific inputs
longest	greatest value so far of specific inputs
pierLength	constant set at variable declaration
ship	numeric input read repeatedly and initialized at the variable declaration
used	keeps track of a total amount when items are added and removed
current	numeric input read repeatedly
currentMonth	ascending for-loop counter starting from 4
greatest	greatest sum of three consecutive input values found so far
monthsToProcess	constant set with an assignment statement
preceding	input value before the previous one
previous	previous input value
total	a running total of inputs

were asked to form groups consisting of 3 to 8 variables. The programs were designed so that the group size restriction did not prevent participants from using sorting criteria based on data flow, role, or form of assignment. If the variables are sorted using position in data flow as the criterion, the result consists of 5 groups containing 3–8 variables each; sorting based on roles of variables yields 6 groups containing 4–5 variables; and sorting based on the form of assignment used to update the variable yields 5 groups containing 4–7 variables.

For control purposes, several alternative naming conventions were used for the variables, e.g., `month`, `smallDigit`, `pier_length`, `SHIP`, `INT_greatest`. (For the sake of clarity, all variable names are written throughout this paper using a single convention, e.g., `pierLength`.) Different naming styles were used within each of the above theoretical groups. Even though it was possible to use the naming convention as sorting criterion (5 groups of 4–8 variables), we did not expect anybody to do that.

Procedure: Participants were run individually. A session started with a background questionnaire conducted by the researcher. The participant was then given five sheets of paper, each containing one program and its modification task, and was asked to study the programs so that he understood them well, and to make the modifications using a pencil. The programs were laid side by side on a table and their order was systematically varied. Participants were allowed to study the programs in the order they wanted, but most of them studied the programs in the given order. There was no time limit for this phase, and the durations varied between 22 and 99 minutes (mean 45.9, mode 40).

The researcher then rearranged the program sheets in the order they were originally presented to the participant and laid cards representing the variables in the program on top of each sheet. Each card had the names of the program and the variable written on it. The researcher asked the participant to sort the cards in groups of three to eight variables so that "similar variables will go together", and to apply the same sorting criterion for all groups. The time used for the sorting task was not limited and it varied between 8 and 35 minutes (mean 17.6, mode 15).

When the sorting task was ready, the participant was asked to give a written explanation for each of his groups. This was followed by an interview where the participant explained the sorting criterion he had used, the exact contents of each group, and alternative sorting criteria he had thought of or might consider to be appropriate.

During the interview, the participant was allowed to move cards between groups. This occurred often, as some of the programs were complicated and some variables had for control purposes obscure (but meaningful) names, resulting in obvious slips in sorting. For example, in order to avoid superlatives in all names for extreme values, the variable holding the smallest digit found so far was not named `smallestDigit` but `smallDigit` and this resulted in some cases in false recall of the meaning of the variable. Participants were also asked to further divide groups having more than 8 cards. In some cases the participant was unable to do that; then the group was left unchanged.

Participants were allowed to put a variable in more than one group by cloning its card, and to leave variables outside any group. The number of extra cards varied between 0 and 21. Seven participants used no extra cards; three used 1–2 cards; and and the remaining three participants used 4, 17, and 21 cards. Two participants left one or two cards outside the groups. The total length of the sessions varied between 58 and 144 minutes.

4. Results and Discussion

The goal of the investigation was to elicit expert programmers' knowledge about variables. The study is qualitative and does not attempt to capture, e.g., relative frequencies of various sorting criteria among professional programmers. Instead, we have taken a descriptive approach and used a relative low number of participants. Therefore, in consequence with this approach we will emphasize the descriptive analysis of the data: first, we perform a qualitative analysis of sorting criteria based on the data from the groupings and the interviews; second, we analyze the most common groups made by the participants; third, we analyze the grouping data using cluster analysis.

4.1. Sorting Criteria

We analyzed the groups, their descriptions, and the interviews paying attention to the commonalities and differences both in the contents and in the words used. This way we were able to identify a set of criteria that the participants used in sorting variables. Even though the participants were instructed to use a single sorting criterion for all groups, it turned out that most of them used several criteria either hierarchically or mixed in some more complicated way, e.g., a couple of groups based on one criterion and other groups based on another criterion with subdivision based on yet another criterion. This is in line with the findings of Cañas and al. [4] who found that in knowledge elicitation tasks the limits of working memory cause context effects in the selection between various criteria for judgements. In the following, we will describe each criterion separately from others.

The set of identified criteria is large and based on a wide variety of principles. The only common feature of the criteria is importance: every criterion conveys some information that is important in some certain task. Consider, for

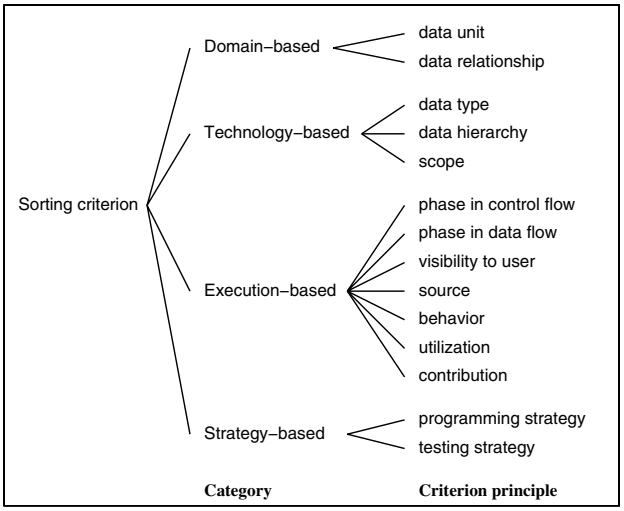

Figure 1. Sorting criterion principles used by the participants or found during interviews.

example, the data type (`int`, `float`, ...) as a sorting criterion. The data type seems to be a simple decision that is made when the need for a variable has raised. It is based on the nature of the data, i.e., how large values are possible and whether decimals are needed, and there seems to be no need to think about it later on. However, when reporting alternative sorting criteria, one participant stated:

> P06: it [the data type] affects the accuracy of results

He thus considered data type as an important information needed in writing expressions that operate on the variable, i.e., data types of variables must be remembered long after the variable declaration has been written and hence they form a sensible basis for sorting.

Figure 1 lists all sorting criterion principles used by the participants or identified in the interviews as possible alternative criteria that the participants thought of using. The criterion principles are meant to be non-overlapping and they are organized in four main categories. Domain-based criteria deal with issues related to the application domain, technology-based criteria deal with the features of programming languages, execution-based criteria are based on activities that occur during the execution of a program, and strategy-based criteria have their origins in the strategies that the programmer applies when working with the program. Each criterion principle, e.g., data unit principle, may give rise to several sorting criteria with differences among details. The rest of this subsection is devoted to a detailed description of the various sorting criterion principles.

Domain-based criteria: Two participants mentioned *data unit* as a possible sorting criterion and one used it to form a subdivision of his primary sorting based on visibility to the user. There were different forms of the data unit criterion: it was based either on the exact data unit (e.g., metres, hours, seconds, ...) or using a coarser division based on the conceptual unit (length, time, ...).

One participant thought of using *data relationship* as a sorting criteria but because the application domains of the programs were different, he abandoned it.

Technology-based criteria: Five participants mentioned *data type* as an alternative sorting criterion. Nobody, however, actually used it because all variables except one were integers. One participant mentioned *scope* (global vs. local) as an alternative sorting criterion but did not use it because all variables were global.

One participant used the naming convention as his sorting criterion. This was against our expectations because the materials were carefully prepared so that variables named using the same convention had nothing in common. The interview revealed that the participant worked in a company that had a strict naming standard where the form of variable names reveal the data hierarchy in a data base, and whether the variable refers to an element in a data base or is a temporary data element. Thus his sorting criterion was actually based on *data hierarchy* and scope even though his technique did not work in the current situation.

Technology-based criteria are easy to use because the required information can be found in the program text. It is therefore obvious that programmers have this type of knowledge about variables. In order to avoid the use of such obvious criteria, the materials were designed to inhibit the use of technology-based criteria. However, the example at the beginning of this subsection demonstrates that technology-based properties, e.g., data type, are important for programmers.

Execution-based criteria: As seen in Figure 1, most sorting criteria used by the participants were based on properties related to the dynamic behavior, i.e., the execution of programs. One participant mentioned *phase in control flow* as an alternative grouping criterion. He described this as:

> P02: ... based on those blocks that are in the program ...
>
> R: What do you mean by blocks?
>
> P02: Entities of the program that do different things X; say lines one to five take care of some task, and lines six to ten take care of another task ... or the textual structure of the program.

Thus, even though the criterion is based on the textual form of the program, the block borders are determined by functional entities or schemas for local goals.

Phase in data flow (used by 5 participants) and *visibility to users* (used by a single participant) are related but

yet different criterion principles. Visibility to users divides variables into two groups: those visible to users (i.e., variables holding input and output), and internal variables. At first this seems to be a coarser version of phase in data flow which has a further division of input and output variables into their own groups. However, participants using phase in data flow criterion did not call the rest of the variables as "internal" but "intermediate" or "auxiliary" and they had problems in deciding what to do with variables that take part in the calculations but are also part of output. In the visibility to users case, this is not a problem: such variables belong simply to the group of variables visible to users. Thus these two criteria are conceptually different.

An interesting detail is the treatment of variables occurring in expressions that are output. For example, one of the programs contained the following statement:

```
printf("Day %d is %d.%d \n",
        dayNbr, dayNbr-daysToBeginning,
        month+1)
```

One participant using the phase in data flow criterion put `month` and `dayNbr` in output variables but `daysToBeginning` in auxiliary variables. When asked during the interview, he reported that `daysToBeginning` is an auxiliary variable because it is used in a calculation to produce the actual output. However, `month` is not output as such, either, but used in an expression to produce output. But in this case the expression is much simpler and only compensates for the difference between month numbers (starting from 1) and C array indexes (starting from 0). Thus, the distinction between output and intermediate variables is not purely syntactical, i.e., it cannot be simply decided based on whether a variable occurs within an expression or is output as such.

A related criterion is the *source* of variables: whether they are obtained from a user or set by the program. This was mentioned by one participant as an alternative criterion. His point was that variables may affect the behavior of programs and it is important to know whose decision the effect is based on: is it based on a user's decision or the programmer's decision.

Sorting criteria classified as *behavior* consider the internal life of variables; not the purposes they are used for but how their own values are obtained. Eight participants used this criterion to form a group for constants, i.e., variables whose value does not change once initialized; five of these participants formed other groups based on this principle, also; and two participants mentioned behavior as an alternative criterion. Groups based on this principle include the following:

- "counters" (P07, P11, P13)
- "total sums" (P03, P11)
- "greatest/smallest" (P03), "extreme values etc." (P04), "extreme values" (P07), "greatest etc." (P11)
- "previous values" (P04, P07), "history data" (P11)

A common behavior-based group was "input", which however can also be obtained by using the phase in data flow criterion.

Two criterion principles—*utilization* and *contribution*—deal with the purpose of a variable in the program. Utilization is more specific and sorts variables according to their own task whereas contribution sorts variables according to the structure they participate in. For example, a group containing loop counters is formed using a utilization criterion while a group containing variables that participate in controlling loops, i.e., loop counters and loop limits, is based on a contribution criterion.

Six participants used utilization principles. Groups based on utilization criterion include:

- "loop counters" (P01, P04), "indexes and control" (P02)
- "current item to limit loops" (P04)
- "intermediate variables for small use" (P03)
- "intermediate variables for important data to be used later" (P03)
- "calculation" (P05)

The group "input" can be considered to be based on utilization but it can be obtained by other criteria, also.

Several participants mentioned or based groups on a coarse utilization criterion dividing variables into two groups: those used to control the execution of the program, and those used for calculations whose results will be output or otherwise needed in further calculations. In the case of input, control variables consist of commands that tell what to do next, and of values that define the number of rounds in some loop; all other input values belong to the calculation group. One participant had a third group in this coarse criterion: variables used for state maintenance.

Two participants used contribution principle: one as a top-level abstraction for utilization-based groups; the other had groups "influences loop control" and "influences conditional statement control".

Strategy-based criteria: Four participants based their sorting criteria on some strategy they used in working with programs. One based his criterion on the overall *programming strategy* approach:

> P09: [I think] first about what is taken in and what is tried to get out and then start to think about the algorithm [...] and if there is iteration then what there must be done, where to start from and where to stop; that's where the edges of loops come from; and then I start to look inside the loop; for example what auxiliary variables are needed

such that their values disappear in every round; and then what intermediate results.

His groups were labeled accordingly: "input", "output", "loop edges" etc.

Three participants based their criteria on their *testing strategy*, i.e., how variables must be attended to when testing and debugging. However, their strategies were not identical. One participant was interested where do variables obtain their values from: through input, through calculation, through both input and calculation, or none of these ways (i.e., constants). Thus he had no group for output variables, because the use of a variable was not interesting to him. On the other hand, another participant had a different testing strategy and different groups: input (because of safety risks), constants (need no attention), loop counters (need to be checked only once), output that can be wrong (where debugging starts; does not include loop counters "[because they] cannot be wrong") etc.

4.2. Common Groups

The previous subsection was based on analysis of groups, their descriptions, and the interviews. The goal of this analysis was to identify all sorting criteria the participants considered meaningful. Another way to look at the data is to find the most commonly occurring groups and analyze what sorting criteria may lead to these groups. We will now turn to this task.

The most often recurring group was "constant". Eleven participants put essentially the same variables in this group, and they described it referring to the behavior of these variables. One of these participants put `count` both in this group and in a loop counter group because, in his opinion, there should have been two variables, a constant that defines the number of loops and a separate loop counter.

Ten participants had a group for inputs. Five of them did not restrict themselves to pure input but considered data items obtained from a random number generator to be input as well. Another difference concerned two variables that obtained their first value through input but later values through assignment. Three participants put these variables in the input group.

The input group can be obtained by looking at the behavior of variables or their phase in the data flow. The latter principle leads to groups "output" and "intermediate (or auxiliary) results" which were common titles of groups. Six participants formed an output group but the contents of these groups are all different. Similarly, five participants had groups for intermediate or auxiliary variables but, again, their contents were different. Thus the participants used behavior-based criteria more consistently than phase in data flow.

Eight participants had a group for loop counters. In addition to counters increasing or decreasing with a step of 1, four participants included also counters traversing through the powers of a base number. One of the participants that did not include these two variables said that "they are kind-of counters". Loop counters can be identified based on their behavior or utilization.

The rest of common groups were all based on the behavior of variables. Six participants had a group for variables storing history data; however, there were small differences among these groups. Four participants had a group for variables holding the greatest or smallest value, and their groups were very similar. Finally, three participants had a group for total sums; one restricted this group to variables obtained through simple addition whereas two participants accepted more complicated operations, also.

All commonly occurring groups are thus based on either behavior or utilization criterion. The utilization criterion yields three common groups: input, output and intermediate results; the latter two groups having the largest amount of variation. The behavior criterion yields all other groups than output and intermediate results: constant, input, loop counter, history data, greatest/smallest, and total sums. Other sorting criteria seem to yield groups with varying names and contents; thus they do not appear among common groups in this study.

4.3. Cluster Analysis

As noted above, the same groups may be obtained by different sorting criteria. In order to obtain a general view of the groups, we applied hierarchical cluster analysis. This analysis method has been used by several researchers to understand if mental representations are organized following a significant set of groups (e.g., [11, 14]). We used the program EZCalc [7] to produce a dendrogram that depicts the frequency of variables occurring in the same groups. This technique does not allow a variable to be included in several groups, so the following adjustments were made:

- For the participant with 21 extra cards, multiple occurrences of variables in the "computing" group were discarded (because the participant said that those should perhaps have been excluded but "I just started to make it this way and did not want to start again").
- The participant with 17 extra cards was excluded totally because we could not find good reasons to exclude the extra cards in any meaningful way.
- In all other cases with a card in two groups, the variable was excluded from both groups and marked "the participant does not understand what this card means"—an option allowed by the analysis program.

Moreover, the participant who used naming convention as his sorting criterion was excluded because this was a

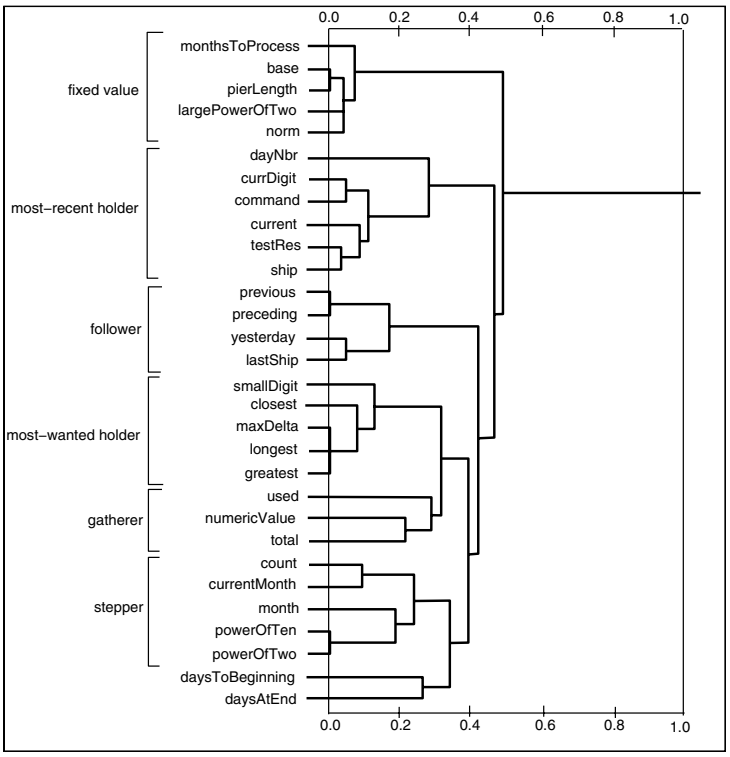

Figure 2. Result dendrogram of the hierarchical cluster analysis.

meaningless criterion for current materials.

Figure 2 depicts the results of the cluster analysis and the relationship of the clusters with the role theory. In a dendrogram, the sooner the lines emanating from variables at the left hand side of the dendrogram are joined, the more frequently the variables occur together in the groups formed by the participants.

The first five variables are constants; `monthsToProcess` is used to control the number of rounds in a loop whereas the others are "magical values" with no special control nature.

The next six variables hold starting values for processing; others are for holding input except `dayNbr`, which is obtained repeatedly from a random number generator.

The next four variables hold past values of input. The variables `previous` and `preceding` come from a program where input data flow goes through both of these variables. Input data flows through the other two variables in the cluster, also, but only some of the values of `lastShip` are used in later processing.

The next six variables find the largest, smallest or closest value. Last three of them (`maxDelta`, `longest`, `greatest`) look for the largest value. It seems odd that the first variable, `smallDigit`, which looks for the smallest value, is more loosely connected to the above three variables than `closest`, which looks for the closest value. However, `smallDigit` occurs in the program that the participants reported to be hardest to understand and its name does not reflect its purpose well. Perhaps some participants did not understand this variable well enough to sort it correctly.

The next three variables—`used`, `numericValue`, and `total`—form a looser cluster. They combine the net effect of input values using various calculations: `used` describes the amount of pier usage when ships are coming in and going out, `numericValue` combines the effect of individual digits when reading a number digit by digit, and `total` is a standard running total. Variables in this group accumulate the effect of all the values in the input flow whereas variables in the previous group pick a single value from the input.

The next five variables do not directly depend on input. The first two are loop counters limiting the number of rounds in the loop whereas the third, `month`, counts how many times the loop is executed. The last two—`powerOfTwo` and `powerOfTen`—occur in the same program and control the execution of a loop by being updated with a division operator, e.g.,

```
for (powerOfTen = 100000000;
     powerOfTen >= 1;
     powerOfTen = powerOfTen / 10) {...}
```

The last two variables—`daysToBeginning` and `daysAtEnd`—occur in a date format conversion program. They keep track of the number of days at the beginning and at the end of the current month in a loop going from January to December. DaysAtEnd is obtained by summing up the number of days in individual months and `daysToBeginning` stores the old value whenever `daysAtEnd` is updated.

Regarding to the sorting criteria, the clusters are mostly based on behavior principles, e.g., the first five variables do not change once initialized etc. To some extent, the clusters are also affected by the utilization principle. For example, the first variable, `monthsToProcess`, is used in a different context—to control the number of rounds in a loop—than the other constants. In the clustering, the other variables are more tightly connected. However, the difference attributed to the utilization criterion is small.

By looking at the knowledge types described in Section 2, the clusters can be best explained by the role theory [15], which is based on the behavior of variables. As depicted in Figure 2, most of the variables are clustered according to the roles fixed value, most-recent holder, follower, most-wanted holder, gatherer, and stepper.

4.4. Validity of the Research

There are some threats to the validity of the investigation. For example, the programs were small, all variables were global, all except one were of the type `int`, and the variables were named using English words. We would have liked to use larger programs, but then the comprehension task would have become too demanding. In order to avoid effects of using a single application domain, several small programs representing several domains were used instead of a single larger program. Global variables representing mainly a single type were used in order to avoid overly use of technology-based criteria, which are self-evident. The lack of data structures and pointers may have prevented the use of some criteria but could not cause the use of artificial criteria. Thus the restrictions of the programs may have resulted in unobserving some criteria but not in false detection of nonexistent criteria.

The naming of variables may affect grouping criteria. To avoid such effects, varying naming conventions were used and the use of similar grammatical constructs were avoided. It would have been possible to use meaningless variable names (`v1`, `v2`, ...) but then the comprehension task would have been harder and misunderstanding of the programs would probably have been frequent. Even in the current form, some of the programs were reported by the participants hard to understand.

In order to increase validity, the materials were designed to support evenly all grouping criteria that were anticipated in advance. The actual number and variability of criteria was, however, so large that the materials could not support all criteria evenly, that is, it was not possible to make groups of three to eight variables using any single criteria alone. On the basis of the results this is not a problem because the participants did use several criteria simultaneously and they were willing to violate group size restrictions when required by the criteria they used.

The identification and analysis of the grouping criteria could be biased by the researchers' intuition and attitude. This problem was tackled by applying several techniques (criteria identification, common groups analysis, hierarchical cluster analysis) on the data. Moreover, the researcher making the analysis has worked as a programmer and has a long background in psychology of programming, which provides him a good ability to interpret the interviews.

The participants were programming experts, which increases the validity of the results. Some of them experienced the modification task unnatural due to the presence of the experimenter and lack of computer support. However, this feeling of unnaturalness disappeared during the sorting task, and the atmosphere of the interviews was open.

5. Conclusions

We have studied professional programmers' mental representation of variables by using a sorting task and interviews. In order to reveal what types of information programmers' have about variables, we identified the sorting criteria used in the sorting task or mentioned as a possible criterion in the interviews. The set of criteria is large and based on 14 different principles that can be organized in four main categories: domain-based, technology-based, execution-based, and strategy-based principles. Each principle gives rise to various criteria differing in details.

All frequent groups are based on two execution-based principles: the behavior and utilization of a variable. Behavior refers to the internal life of a variable, i.e., how its values are obtained. Utilization refers to the task that the variable has in the program, e.g., the variable is a counter controlling the execution of a loop. In the hierarchical cluster analysis, the main clusters are formed according to the behavior of variables; utilization affects subclustering within the main clusters. The variation among utilization criteria is however larger than among the behavior criteria.

Excluding data relationship, none of the criteria deals with unique tasks or patterns in unique programs, which is a central theme in past literature on program knowledge [2, 3, 10, 18]. Instead, the criteria are general in the sense that they apply to variables in all programs. This type of knowledge has earlier been described as programming knowledge in the form of plans [8, 12] or roles [9, 15]. Plans combine behavior with utilization and program code

extensions like guards that protect variables against invalid updates. The earlier notion of roles [9] is a mixture of behavior and utilization, whereas the recent role concept [15] is a clear example of a behavior criterion. The other criteria identified in the present study have not been presented as variable-related knowledge in previous literature.

Program comprehension tools should help programmers to build mental representations by providing meaningful information about programs. We have identified fourteen information types that programmers possess about variables. For the purposes of program comprehension, tools should generate categorization of variables in all of these types automatically. The main categories of the information types require, however, different approaches. Domain-based information types are the hardest to support because they require knowledge about the application domain and this kind of information is hard, if not impossible, to be generated automatically. Hence domain-based information should be made explicit already in the programming phase by the use of appropriate coding and documentation standards. In contrast, technology-based information types are easy to support because such information is explicit in programs, gathered by compilers, and already partially available in current programming environments.

There is a large variety of execution-based information types but behavior and utilization seem to be the most common forms and should be supported by tools. The variation among behavior criteria was smaller than the variation among utilization criteria and thus behavior (e.g., roles) seems to offer more promising possibilities for tool development. Strategy-based information types should also be supported because this type of information directly helps programmers in their every-day routines, especially in testing and debugging. Research into various ways of providing these types of information to programmers is still needed.

6. Acknowledgments

This work was supported by the Academy of Finland under grant number 206574.

References

[1] M. Ben-Ari and J. Sajaniemi. Roles of variables from the perspective of computer science educators. In *The 9th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2004)*, pages 52–56. Association for Computing Machinery, 2004.

[2] R. Brooks. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9:737–751, 1977.

[3] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[4] J. J. Cañas, A. Antoli, and J. F. Quesada. The role of working memory on measuring mental models of physical systems. *Psicologica*, 22:25–42, 2001.

[5] E. S. Cordingley. Knowledge elicitation techniques for knowledge-based systems. In D. Diaper, editor, *Knowledge Elicitation: Principles, Techniques and Applications*, pages 89–178. Chichester, U.K.: Ellis Horwood Ltd, 1989.

[6] S. P. Davies, D. J. Gilmore, and T. R. G. Green. Are objects that important? The effects of expertise and familiarity on the classification of object-oriented code. *Human-Computer Interaction*, 10:227–248, 1995.

[7] J. Dong, S. Martin, and P. Waldo. *A User Input and Analysis Tool for Information Architecture*. http://www-3.ibm.com/ibm/easy/eou_ext.nsf/Publish/410, 2004.

[8] K. Ehrlich and E. Soloway. An empirical investigation of the tacit plan knowledge in programming. In J. C. Thomas and M. L. Schneider, editors, *Human Factors in Computer Systems*, pages 113–133. Norwood, NJ: Ablex Publishing Company, 1984.

[9] T. R. G. Green and A. J. Cornah. The Programmer's Torch. In *Human-Computer Interaction - INTERACT'84*, pages 397–402. IFIP, Elsevier Science Publishers (North-Holland), 1985.

[10] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[11] R. S. Rist. Plans in programming: Definition, demonstration and development. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers*, pages 28–47. Norwood, NJ: Ablex Publishing Company, 1986.

[12] R. S. Rist. Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers. *Human-Computer Interaction*, 6:1–46, 1991.

[13] J. Robertson. Information design using card sorting. http://www.steptwo.com.au/papers/cardsorting/, 2001.

[14] S. P. Robertson and C.-C. Yu. Common cognitive representations of program code across tasks and languages. *International Journal of Man-Machine Studies*, 33:343–360, 1990.

[15] J. Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, pages 37–39. IEEE Computer Society, 2002.

[16] C. Simonyi. Hungarian notation. http://msdn.microsoft.com/library/en-us/dnvsgen/html/hunganotat.asp, 1999.

[17] H. M. Sneed. Program comprehension for the purpose of testing. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, pages 162–171. IEEE Computer Society Press, 2004.

[18] A. von Mayrhauser and A. M. Vans. Industrial experience with an integrated code comprehension model. *Software Engineering Journal*, 10:171–182, 1995.

[19] A. Walenstein. Observing and measuring cognitive support: Steps toward systematic tool evaluation and engineering. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 185–194. IEEE Computer Society Press, 2003.