Informatics in Education, 2005, Vol. 4, No. 1, 123–142

Delphi Study of the Cognitive Skills of Experienced Software Developers

Sami SURAKKA, Lauri MALMI
Helsinki University of Technology, Laboratory of Information Processing Science
P.O. Box 5400 (Konemiehentie 2), FIN-02015 HUT, Finland
e-mail: sami.surakka@hut.fi, lauri.malmi@hut.fi

Received: December 2004

Abstract. In the present paper, a qualitative research of the cognitive skills of experienced software developers is presented. The data for the research was gathered using the Delphi method. The respondents were 11 software developers who have worked at least five years after their graduation. Two questionnaire rounds were conducted. In the first round, the respondents mentioned altogether 32 different skills. In the second round, 10 of the respondents answered and evaluated the difficulty of these 32 skills (e.g., does the skill efficiently differentiate experts from novices). The results are divided into two categories: composition and comprehension. Approximately 40% of the skills were classified into the category "comprehension." For each skill, the evaluated degree of difficulty of the skill is presented. In the category comprehension, skills related to comprehension of a program or a problem as a whole were evaluated as the most difficult.

Key words: cognitive skills, experts, psychology of programming.

1. Introduction

What are cognitive skills? According to ERIC Thesaurus (2004), the term *thinking skills* should be used for the term *cognitive skills*. The description for the term thinking skills is the following:

> Interrelated, generally "higher-order" cognitive skills that enable human beings to comprehend experiences and information, apply knowledge, express complex concepts, make decisions, criticize and revise unsuitable constructs, and solve problems – used frequently for a cognitive approach to learning that views explicit "thinking skills" at the teachable level.

In the present research, the goal has been to identify cognitive skills that are important for expert software developers' work. Our research origins from the need to better understand what kind of topics and skills should be included in the master's level education of software systems specialists in the Helsinki University of Technology. Typical sources for such curriculum development work include various model curriculums such as Computing Curricula 2001 (Engel and Roberts, 2001). However, they mostly concentrate on listing topics to be covered in the curriculum. The skills to be achieved during the education are covered more vaguely. Since programming is a high-level cognitive skill,

we wanted to find out in some more detail about what kind of cognitive skills should be trained in the education.

We decided to search for high-level software development experts and ask from them which topics in computer science they consider important for their work. Moreover, we were interested in identifying tacit knowledge needed in software development. Since such information is difficult to be grasped with simple questionnaires we decided to apply the Delphi method (Wilhelm, 2001) in which people in the same focus group are queried two or more times. After each time a summary of results is presented for them followed by more closely defined questions of the topic of interest. Delphi is a qualitative research method, where the quality rather than the number of respondents is the more important factor. The statistical reliability of the results is therefore not the general goal, and thus the number of respondents need not be very large. In the present research, we selected 11 respondents among a group of recommended 59 experts. Two questionnaire rounds were performed, and the second round concentrated especially on the tacit knowledge of software development. In the present paper, we concentrate on the results of the second questionnaire round.

The present paper is an extended version of our previous conference paper (Surakka and Malmi, 2004). The structure of the present paper is the following. First, we consider some related work in Section 2. In Section 3, we describe the research method in some detail. The results are presented and analyzed in Section 4. A discussion including some implications to education and evaluation of the present research summarizes the paper.

2. Related Work

We did not find any research papers where the Delphi method has been used in the field of psychology of programming. This is understandable because it is not common to use even questionnaires as a research instrument in this field.[1] Because the lack of similar research, some more general references are presented next. At the end of this section it is explained how these issues relate to our research. Greeno and Simon (1988) wrote "Computer programming may be characterized 'as a whole' as a design task." Brooks (1983) wrote about design task domains:

> . . ., two fundamental activities in design task domains are composition and comprehension. Composition is the development of a design and comprehension results in an understanding of a design. The essence of the composition task in programming is to map a description of what the program is to accomplish, in the language of real-world problem domains, into a detailed list of instructions to the computer designating exactly how to accomplish those goals in the programming language domain. Comprehension of a program may be viewed as the reverse series of transformations from how to what.

[1] We found only seven articles where questionnaire has been used, for example (Capretz 2003). However, none of these articles is really related to our research beside the use of questionnaires.

Stanislaw et al. (1994) divided expertise in computer programming into two components that were time-based expertise and multiskilling expertise. They wrote (p. 351): "*Time-based* expertise corresponds to the conventional notion of expertise, and is a function solely of the time spent on programming. *Multiskilling* expertise, by contrast, accrues through exposure to a variety of programming languages and tasks, and is related to the cognitive development of higher-level programming schemata." Détienne (2002, p. 35) wrote that one of the characteristics that distinguishes "super experts" or "exceptional designers" from other experts is: "a broader rather than longer experience: the number of projects in which they have been involved, the number and variety of the programming languages they know." In addition, Détienne (2002, p. 35) wrote that experts carry out some aspects of programming tasks completely automatically. She referred to Wiedenbeck (1985, p. 383) who found that experts were faster and made fewer mistakes than novices when both groups had to do a series of timed true/false decisions about short, textbook-type program segments. Perhaps, for example, the following skills are automated gradually when the programming experience increases: (a) using the basic commands of an editor (such as Emacs) and the programming system frequently used, and (b) knowing the details of the syntax and the code conventions of a certain programming language such as C. The previous issues relate to the present research as follows: (a) We have used two activities, composition and comprehension, to interpret and divide our results. (b) The division time-based expertise vs. multiskilling expertise was used so that we required that at least half of the respondents should be characterized as multiskilled experts. (c) The concept of skill automation was used with the questions about cognitive skills: the first question concerned higher-level skills and the second question concerned skills that might be partially or totally automated.

One aspect of cognitive skills is different design strategies. Détienne (2002, p. 26) wrote that experts have a broader range of more versatile strategies than novices. In addition, she wrote (p. 26): "Design strategies can be classified along several axes: top-down vs bottom-up, forward vs backward development, breadth-first vs depth-first, procedural vs declarative." See Appendix A for the explanation of these strategies. This division of strategies was used during the second questionnaire round as will be explained later in Section 3.2.2. Originally, Visser and Hoc (1990, pp. 241–244) presented these design strategies and used somewhat different names. However, in the present research the names presented by Détienne (2002) were used.

3. Method

An overview of the Delphi method can be found, for example, from (Wilhelm, 2001). The method was originally used to forecast the future; the name originates from "the oracles of Delphi" where Delphi refers to an ancient Greek island. However, in the present research, estimating the future was only a small part. Some basic properties of the method are the following. First, there are several questionnaire rounds. Second, the results from the previous round are used as material for the next round. Thus the respondents may

change or tune their previous answers. One of the main reasons for using Delphi was that it allows group communication without gathering all respondents to the same place at the same time, which in this case would have been very difficult to achieve. Moreover, in this way the respondents had more time to consider their answers and make their views more explicit. Originally, consensus building has been an important part of the Delphi method. In the present research, however, the second questionnaire round was not used for building consensus on the whole issue but targeted more to refining the results of an interesting part of the first questionnaire; that is, cognitive skills. The first questionnaire had three open questions about cognitive skills required by a software specialist. Based on the answers in total 36 different skills were identified. In the second round the respondents defined the level of these skills, that is, how long learning and experience is needed before such a skill is mastered. The questionnaires are presented in more detail in Section 3.2.

The decision of limiting the second questionnaire to only one area of interest was based on several reasons: (a) The results from the other areas of the first questionnaire were satisfactory enough. Thus, the need to conduct a second questionnaire round for the sake of the other areas was low, (b) The respondents thought that the questions about cognitive skills were the most difficult to answer. We interpreted this as a hint to explore more this area, (c) Regardless of the answering difficulties, some respondents thought cognitive skills as interesting or promising area for this kind of research. This was our own opinion as well, and finally, (d) In the beginning of the research we promised to the respondents that participating would take 1–3 hours, and we wished not to break this promise.

After the cognitive skills were chosen as the topic for the second questionnaire round, the goal was set to evaluate how demanding or difficult the different cognitive skills that were mentioned during the first round are.

3.1. *Finding Respondents*

The goal was to find 10–20 especially good software developers. The respondents were found using recommendations. Thus, they are not a statistically representative sample of all software developers but more like a focus group. Probabilistic sampling was not used because it was difficult to identify the target group using properties such as age, education, and title. For example, the title and working years are not enough to separate especially good software developers from poor or intermediate developers. Kitchenham and Pfleeger (2002, p. 19) wrote that one reason for using a non-probabilistic sample is that the target population is hard to identify. Our decision fits well with this guideline.

The minimum criteria were a degree, five years working experience after graduation, at least half of time used to programming during these five years, and at least 100,000 lines of self implemented code. In addition, at least half of the respondents should have versatile software development experience. Here, versatile means different kind of projects, for example various programming languages and application domains. Two extra criteria were that (a) maximum of three respondents can be included from the same organization and (b) only one respondent can work full-time at the Helsinki University of Technology, where the authors work themselves. The degree could be from other

programs than computer science and engineering. For example, some older respondents had the degree from electrical engineering. The title of the respondent did not need to be programmer, software developer or software engineer, since the important issue was only that their work included enough programming.

Altogether, 59 persons were recommended. 40 of them were not asked because of several different reasons (e.g., the person was graduated less than five years ago). Thus, 19 persons were asked to participate. From these 19 persons, 11 promised to participate.

In most cases, the criterion of any degree was checked from the student register of the Helsinki University of Technology or the personal WWW pages of the candidates. The criteria of at least five years of working experience after graduation, at least 100,000 lines of self-implemented code, and enough programming experience during the last five years were checked when the person was asked to take part. On other words, we simply asked the candidates if they fulfilled the criteria. Some candidates declined because of these three conditions. The criterion of at least half of the respondents should have versatile software development experience was controlled with the first questionnaire. More than half of the respondents had versatile software development experience (see Section 4.1). Thus, no respondents were excluded because of this criterion.

3.2. *Questionnaire Rounds*

Two questionnaire rounds were conducted. The first questionnaire was answered between November 2003 and January 2004, the second questionnaire between January and February 2004. During the first round, most respondents answered so that the first author of the present paper was present during they answered. Thus, the respondents were able to make questions. The authors of the present paper were not present during answering on the second round. The mean answering time for the first round was one hour and six minutes, and 54 minutes for the second round. The original questionnaires are available in Finnish only at (Surakka, 2004). However, their main properties are presented in the following two subsections.

3.2.1. *First Questionnaire*

The first questionnaire had 14 open questions and 14 multiple-choice questions. The topics were (a) background information about the respondent, (b) the importance of various subjects and skills for software development, such as discrete mathematics and concurrent programming, (c) cognitive skills, (d) problem solving techniques, and (e) software quality. For brevity, only results about the background information, cognitive skills, and problem solving techniques are presented in the present paper.

The questions about background information were the title, the proportion of time used to programming, the number of employees under the respondent, lines of code implemented by the respondent, the number of different groups involved, the number of different projects, personal skills in various subjects (42 subitems such as discrete mathematics and object-oriented programming), skills in various programming languages, and knowledge of various operating systems.

Instead of cognitive skills, the concept "tacit knowledge" was used because we assumed that it would be easier to understand for the respondents. An explanation of the concept including initial division into cognitive skills and technical skills was given before the questions. Three questions were:

- For a top-level software developer, what are important mental models, beliefs, and understanding that belong to the cognitive element of tacit knowledge?
- For a top-level software developer, which topics and skills belong to the technical element of tacit knowledge? These can also be called skills that are located in the fingertips.
- Do you believe that some area of tacit knowledge will be more important in the future?

3.2.2. *Second Questionnaire*

The second questionnaire was based on the respondents' answers and comments to the first questionnaire. These were analyzed to identify and separate different skills mentioned in the comments. Comments clearly denoting the same skill were joined. Typing skill was included in the list, based on researcher's observations, even though the respondents did not mention it. Finally, we had a list of 36 comments each identifying at least one skill, for the next round. In the second questionnaire, the respondents had to evaluate the level of these comments according to the following categories:

1) very low-level skill that even novices can learn quickly (during a 1–4 credits basic course);
2) somewhat low-level skill that requires working experience of 3–6 months to be learned, for example;
3) somewhat high-level skill that starts to differentiate good programmers from less good programmers;
4) very high-level skill that takes usually several years to learn and typically only top-level programmers have this skill.

In the question about problem solving techniques, the respondents were asked to read or browse three pages of the book (Détienne, 2002, pp. 26–28). The text was about a strategy-centred approach to software design. After this, the respondent had to answer the following question:

> During the designing of software, have you used these techniques or strategies (Top-down vs. Bottom-up, Forward vs. Backward Development, Breadth-first vs. Depth-first, Procedural vs. Declarative, Mental Simulation)? How often? Do you think they are good? Has the use of these techniques or strategies changed when you have gained more experience in software development (as it is described in the book)? Do you think these skills are tacit or explicit knowledge? What do you think the level of these skills is (the scale is the same as previously: 1 Very low-level skill . . . 4 Very high-level skill)?

The second questionnaire had questions about typing skills and the use of editor as well. These questions are not presented here because they were so simple that just presenting the results is enough.

4. Results

First, some background information about respondents is presented. Second, the results about respondents' opinions from cognitive skills are presented.

4.1. *Background Information of Respondents*

Background information is presented for 11 respondents who answered the first questionnaire round. All respondents were male and mean of respondents' ages was 37.1 years. Their degrees were as follows: one college degree in computer science and engineering (9%), five masters in computer science and engineering (45%), three masters in other engineering disciplines (27%), one doctor from applied mathematics (9%) and one doctor from computer science and engineering (9%). The respondents' positions were distributed into following groups: senior software engineers and developers 45%, researchers 27%, and managers or directors 27%.

Each respondent was asked how many projects he had participated and how well he can program with various programming languages. We classified a respondent to have versatile software development experience if he had participated to at least three projects and had good or excellent skills in at least three programming languages from different programming paradigms (typically C, C++, and Lisp). All respondents had participated to more than three projects but apparently some knew well only one or two programming paradigms. According to the answers, six (55%) respondents were classified as having versatile software development experience.

Each respondent was asked to give himself a grade in 42 subjects or skills related to various fields of computer science, or other sciences (mathematics, physics), and software development phases of the waterfall model. In Table 1 are shown the means from respondents' answers divided into the same four categories that were used in the questionnaire. Inside each category, the rows are ordered first according to the mean and second according to the name of the subject or skill.

Table 1

Means (M) and standard deviations (*SD*) to question "Give yourself a grade in the following subjects or skills" (scale: 1 poor . . . 4 excellent)

Subject or skill	M	SD
Mathematics, physics, and theoretical computer science:		
Logic (in particular, propositional and predicate logic)	2.5	0.8
Other areas of theoretical computer science (e.g., automata)	2.5	0.9
Discrete mathematics	2.4	0.8
Physics	2.1	0.5
Mathematics for continuous systems	1.8	0.8

To be continued

Continuation of Table 1

Subject or skill	*M*	SD
More technical or part of the operational systems:		
Procedural programming	3.8	0.4
Data structures and algorithms	3.5	0.5
Script programming	3.5	0.5
Object-oriented programming	3.4	0.8
Operating systems (operating principles in general)	3.1	0.7
Functional programming	3.0	1.0
Internet protocols	2.8	0.9
Systems programming	2.8	1.1
Compilers	2.7	1.2
Computer/data security	2.6	0.8
Implementing techniques of WWW systems	2.6	0.9
Software architectures	2.6	0.8
Computer architecture	2.5	0.7
Implementing techniques of user interfaces	2.5	0.9
Embedded systems	2.4	0.9
Artificial intelligence and knowledge engineering	2.3	1.3
Concurrent programming	2.3	0.9
Database managements systems	2.3	0.6
Distributed systems	2.3	0.9
Logic programming	2.2	0.6
Computer graphics	2.1	0.7
Extensible Markup Language (XML) techniques	2.1	0.5
Other telecommunications techniques than Internet protocols	2.1	0.8
Real-time systems	2.0	0.6
Software engineering (different phases of life cycle):		
Implementation	3.8	0.4
Design	3.4	0.7
Testing	3.1	0.7
Requirements	2.9	0.7
Concept exploration	2.6	0.7
Approval	2.3	1.0
Operation and maintenance	2.3	0.9
Installation and checkout	2.1	0.8
Packaging and delivery	1.7	0.6
Retirement	1.5	0.7
Software engineering (possible in several phases):		
Version and configuration management	3.1	0.9
Project management	2.8	0.8
Documenting	2.7	0.5

The sample size is 10 for Computer/data security and 11 for the other items.

There are two issues that are worth noticing. First, script programming skills are ranked very high. This obviously correlates with the heavy use of Unix/Linux environment in their work. We did not ask more questions on scripting on the second round. However, our interpretation of this phenomenon is that for this target group scripting is a regular method for solving simple computational problems, for example, filtering and manipulating data files, or building auxiliary tools for them. This is strongly related with the important cognitive skills of recognizing the need for building new tools and choosing a suitable tool for each purpose.

The second observation is that functional programming is ranked much higher than the general use of functional programming languages in software production would indicate. We believe that this is related to multiskilling. A plausible explanation is that many of the respondents have used functional programming during the career and/or hobby programming. Based on answers to the open question about working experience at least four (36%) respondents had actually used Lisp in some work project.[2]

4.2. *Respondents Opinions about Cognitive Skills*

In the second questionnaire, the statements of skills were divided according to the division used in the first questionnaire. However, for the present paper we reclassified the results into two categories: composition and comprehension. We also combined some comments. Two comments are not presented in the tables because they are not related only to software development. These two comments and their means were Being systematic 2.1 and Ability to type using ten fingers 2.1. Thus, Tables 2 and 3 contain together only 30 (17 and 13, respectively) items whereas the second questionnaire contained 36 items. First, the results related to composition are presented in Table 2. The comments are ordered according to the means. The numbers in the leftmost column are used for commenting on the items.

Even though statistical analysis was not our main purpose, we were curious to see, whether the observed differences are significant or not. We assume that a typical reader is not familiar with nonparametric tests but knows the Student t test that is often used to compare means. In the present research, the Student t test was not suitable because the samples were so small. As far as we know, there are no statistical tests available for comparing means in the situation of this kind and the Mann–Whitney test (Conover, 1999, pp. 271–275) is the most suitable alternative for the Student t test. *Note that the Mann–Whitney test compares the ranks (the order of items) as a whole, not the means*. The test is called as a nonparametric test because “parameters” such as means and standard deviations are not used. For comparison, the Student t test uses means and standard deviations.

[2]Nine (82%) respondents have graduated from the Helsinki University of Technology where Scheme was the language of the first compulsory programming course in the degree program of computer science and engineering (CSE) during 1989–2003. However, this is not a suitable explanation because all these nine respondents were admitted before 1989 or were from other degree programs than CSE. That is, the course in question was not compulsory for them.

Table 2

Comments classified into category "Composition": Means (M) to question "What do you think the level of this skill is?" Scale was: 1 very low-level skill. . . 4 very high-level skill

Number	Comment	M
1	A good programmer has always a model. The code itself comes from the spine and brains operate only the model.	3.6*
2a	Automating one's own work using scripts, keyboard macros etc.	3.5*
2b	The mastery of a certain programming language or a certain environment	3.5*
4	Writing code so well that it is not even necessary to comment	3.4
5	Design of interfaces	3.3
6	Choosing as optimal data structures and algorithms as possible	3.1
7a	Ability to find right abstractions	3.0
7b	Mastery of the structures and idioms that are characteristic of each language or environment	3.0
9	Ability to write code clearly and shortly	2.9
10a	Choice of the programming language	2.8
10b	Implementing programs as independent of the operating environment as possible	2.8
12	Isolating the implementation behind well-defined (and documented) interfaces	2.7
13	Changing lower level cognitive models/design patterns to code. For example, table field in C/C++ object and its memory management get/set/constr/destr	2.6
14	Identifying concepts	2.4
15a	Ability to find existing Open Source solutions from Net and being familiar with libraries	2.3
15b	Procedural or object-oriented way of thinking about programming	2.3
17	Documenting code	1.9*

A star (*) indicates that the difference of the corresponding *ranks* is statistically significant ($p < 0.01$). For brevity, the means of ranks are not presented.

For brevity, we present the results of the Mann–Whitney test in the same column with the means and do not present the means of ranks at all. The ranks of single items were compared with the ranks of all items. A star (*) indicates that the difference is statistically significant ($p < 0.01$). If the star is missing, the difference is not statistically significant.

In Table 2, there are a few observations which need to be commented. First, the high mean of item "2a Automating one's own work using scripts, keyboard macros etc." obviously does not indicate the time needed to learn such skills. Instead, it indicates the time needed to use them efficiently as one's personal tools, when necessary. Our assumption is that this is a skill which is analogous to bottom-up software design, where the programmer recognizes the need for general-purpose procedures and data structures. Thus, it has a role in differentiating excellent developers from others. Second, the items "Design of interfaces" and "Isolating the implementation behind well-defined (and documented) interfaces" are kept separate. The first one is more associated with *designing* and the latter one with *using* interfaces. It is obviously easier to learn to use ready-made interfaces properly than actually designing interfaces that support good software architecture. Third, comments 2b and 7b are similar but we think that 2b is broader than 7b. Comment 2b in-

Table 3

Comments classified into category "Comprehension": Means (M) to question "What do you think the level of this skill is?" Scale was: 1 very low-level skill . . . 4 very high-level skill

Number	**Comment**	M
1	Ability to see all possible alternatives from the source code (this comment was related to debugging)	3.9*
2	Ability to notice isomorfisms with some known problem	3.6
3	Ability to evaluate how the system will operate even before its implementation has been started	3.5
4a	Ability to see esthetic values in solutions	3.4
4c	Ability to see the big picture. What is the core of the problem and how it is connected to the environment around it?	3.4
6a	Ability to distinguish essential matters	3.2
6b	Interpreting the program as the whole	3.2
8a	Ability to change fluently • abstraction level (e.g., single line of code vs. procedure or big picture vs. details) • perspective (e.g., is the control flow or the data flow of the program examined) • concepts (e.g., are the concepts of program or the concepts of application domain considered) • view (e.g., users needs vs. maintenance vs. development speed)	3.1
8b	Ability to debug	3.1
10	Ability to see symmetries	3.0
11	Exploring the architecture of the existing systems	2.9
12	Ability to see a big problem as several partial problems	2.7
13	Understanding the functioning of programming languages and computer (e.g., parameter passing, the order of execution, and concurrency)	1.8*

A star (*) indicates that the difference of the corresponding *ranks* is statistically significant ($p < 0.01$). For brevity, the means of ranks are not presented.

cludes also low-level knowledge, for example knowing language's keywords by heart. Forth, we think that the low ranked items 15a and 17 are not really cognitive skills, but other kind skills or knowledge. However, we have not omitted these items from the table because they are related to composition.

In Table 3 we present the results related to category "Comprehension". As a general note, it is interesting that the respondents have used words like "see" and "notice" often to describe these skills. We think that item "13 Understanding the function of programming languages and computer (e.g., parameter passing, the order of execution, and concurrency)" is rather explicit than tacit knowledge.

4.3. *Respondents Opinions on Problem Solving*

Table 4 presents means to the question where the difficulty level of different development strategies was asked. The strategies are presented in Appendix A. Some respondents did not answer this question or did not give the level for all strategies. Some respondents

Table 4

Sample sizes (n) and means (M) to question "What do you think the level of these skills is?" Scale was: 1 very low-level skill . . . 4 very high-level skill

Strategy	n	M
Breadth-first vs. depth-first	6	3.3
Procedural vs. declarative	6	3.3
Top-down vs. bottom-up	8	3.1
Forward vs. backward	4	3.1
Mental simulation	4	2.9

gave different values for single strategies, for example, one respondent gave 2 for top-down and 3 for bottom-up. In this case, the mean 2.5 was used for strategy "Top-down vs. bottom-up." It can be noticed that the differences between the means are small. No statistical tests were conducted to analyze the results because the differences and the subsamples were so small.

The question had other parts as well. These answers were so mixed that no statistics are presented. Most respondents commented that they have used all or most of the strategies and they have often used a combination of strategies (e.g., "top-down + verification by mental bottom-up simulation" or "bottom-up + declarative"). Some respondents commented that single strategies are explicit knowledge but choosing a suitable strategy and changing fluently between the strategies is tacit knowledge that takes years to achieve. One respondent commented that the list of strategies did not mention iterative technique and another that "design by aesthetics" was missing. He explained that design by aesthetics is a kind of extreme bottom-up where first, central parts are programmed as the developer wants them to be. Second, it is considered what has to be done so it is really possible to use these central parts as they were coded.

4.4. *Typing Skills and Use of Editor*

To get some data on the lower level practical skills of the respondents, the questionnaire included a few questions on their typing skills and the editors they use in practical work. Four (40%) respondents have taken a typing course. Five (50%) respondents could type with ten fingers, four (40%) used less than ten fingers but did not look at the keyboard during programming, and only one (10%) respondent had to look at the keyboard while typing.

We wanted to compare the previous results against some other group. However, we did not find any records or previous publications how common typing skills are among general population or among CS graduates in particular. Therefore, we asked similar question from the students of a basic programming course. The questions were presented as part of the normal feedback questionnaire at the end of the course. 13% of the students ($N = 216$) had taken a typing course and 25% could type using ten fingers. 18% answered that he or she did not have to look at the keyboard during one minute of constant typing, 47% had to look 1–5 times, and 33% more than five times.

According to the Z test for proportions (Milton and Arnold, 2003, p. 324), the difference between the experienced software developers and the students is statistically not significant ($p \geqslant 0.05$) for the proportions of ability to type using ten fingers (50% and 25%, respectively). However, we think that looking at the keyboard is more important than using ten fingers because looking at the keyboard might interrupt thinking. If the answers are interpreted so that only 10% of experienced software developers had to look at the keyboard and the corresponding proportion for the students was 82%, then the difference is statistically very significant ($p < 0.001$). Thus, there is some evidence that the typing skills of experienced software developers are better than of students – as one would expect.

An editor is the basic tool in programming. Good knowledge about the versatile features allowed by advanced editors can significantly improve the coding speed. Based on the background of the respondents, we deduced (even though we did not ask this) that most if not all had used mainly Emacs in Unix environment in their student days. According to the answers, most respondents have also continued to use Emacs after graduation, which is well understandable knowing its wide variety of available operations and support for editing different languages. Six respondents used mainly Emacs, one respondent Epsilon (an Emacs clone), and one respondent used vi. One respondent has changed from Emacs to Source-Navigator because he thought that Source-Navigator was more suitable for editing and browsing large programs. Five respondents answered that they had programmed macros for Emacs, and two answered that they knew basic commands of Emacs but had not programmed macros. At the time of answering, one respondent programmed mainly in Windows environment and used Visual Studio. Two respondents answered that they do not work in Windows environment. Others answered that they have installed Emacs in Windows when necessary.

5. Discussion

In this section, the research is evaluated, conclusions are drawn, implications to education are presented, and possibilities for future research are considered.

5.1. *Evaluation of the Research*

The present research would have been very different if the original main goal was to gather information from the cognitive skills of software developers. Questionnaires are used seldom in the psychology of programming where experimental research setting is dominant. One source of criticism is that questionnaires measure opinions, not observable behavior. However, in the present research the purpose was to measure especially the opinions of experts. In addition, we think that Delphi method was suitable for this type of research. The follow-up questionnaire round was necessary to describe more explicitly tacit knowledge that is a more or less vague concept. During the first questionnaire round, most respondents commented that the questions about the tacit knowledge were the most difficult to answer. A possible interpretation could be that the used research method was

not suitable or the questions were poorly designed. However, we interpreted that the answering difficulties were mainly due from the topic itself; that is, the topic is genuinely difficult. It is possible that the respondents do not remember or cannot describe skills that have been automated already several years ago. For example, adults often have difficulties in describing how a bicycle is ridden or a car is driven. We tried to minimize this problem by dividing the questions in two parts and adding an explanatory text before the questions.

5.2. *Conclusions and Implications to Education*

The skills listed can be divided into two main categories: skills associated with composition and skills associated with comprehension. The composition category obviously includes skills that are related to the mastery of the programming languages and environments used. Other important skills associate with having an inherent model of the goal in one's mind, designing interfaces and abstractions, mastering and developing one's own working process, for example. The comprehension category includes skills such as understanding the program as a whole and ability to notice isomorfisms with other known problems.

On a general level, the results confirm that different comprehension-related tasks are an important part of the cognitive skills of software developer. Approximately 40% of the items mentioned by the respondents can be classified as comprehension-related tasks. Obviously, this is not at all surprising result because according to the definition presented in the very beginning of the present paper, cognitive skills enable human beings to comprehend information. It is obvious that many of the skills listed above cannot be taught directly in the courses. They are highly related with a long experience gathered when programming solutions to different problems. The challenge for education is to design project assignments where students will face problems that require the mentioned skills, and find a way to present guidelines for adopting such skills.

On a more general level, we assume that the deployment of the results of the present research might increase the proportion of time used into concept exploration, requirements analysis, and design phases but decrease the proportion of time used into implementation phase. In the following, we mention a few course examples of such development.

Refactoring course

This example would be an advanced course that emphasizes comprehension. During the refactoring course, a student should repair and/or partly rewrite a program (maybe 2000–3000 lines) that contains different kinds of mistakes and bad planning choices. During the task, a student has to read and thus interpret the structures and the operation of a program written by others. Moreover, he/she should argue about the findings made, and how the code should be improved.

Software design workshop

This course would emphasize the composition viewpoint, including analyzing and decision-making skills related to design. The course would contain an open or

semi-open design problem that can be solved using several different strategies and tools. The student group should compare various options, argue their pros and cons, and finally evaluate the result.

Project course (customization/tailoring)

In a customization/tailoring course the group faces a problem of designing a program for a variety of customers with slightly different needs. They should analyze the needs in the specification phase and argue what kind of architectural solution would enable generating different versions of the basic program, and argue their decisions on the program design. To make the project more challenging they should implement the first version, and thereafter get the requirements for new customers, and then analyze how their initial design works in the new situation.

Project course (the combination of student projects)

The main goal here is to force students to read and understand the designs and implementation of other students, and continue their own work based on these. For example, a group should split a task into appropriate subgoals and assign a number of other groups a task to design and/or implement solutions for the subgoals. Thereafter the original group should compare a number of submitted designs/implementations and choose one or two of them as a part of their own project. They would have to use and modify the design/code to match their needs and argue about the process; that is, reflect on their own decisions when assigning the subgoals and when comparing the results.

In general, the students should be faced with problems where they have to understand and modify code written by others, and not necessarily the best quality code with good documentation. This would promote both comprehension and analysis skills as well as composition skills. In our opinion, programming education typically relies too much on implementing everything oneself.

5.3. *Automation of Low-Level Skills in Programming*

In the second questionnaire, the respondents were asked about typing skills and the use of an editor as well. This might seem odd because these are so low-level time-based skills; that is, fast typists who can use their editor well might be faster programmers. In our opinion, it is not so interesting if software developers were 10% or even 50% faster but we wonder if learning problems in low-level skills might cause difficulties when students are learning higher-level programming skills. Based on earlier findings about skill automation and learning in general, this might be the case. For example, according to Wiedenbeck (1985, p. 384):

> Studying children, Perfetti and Hogaboam (1975) and Perfetti and Goldman (1976) found that less-skilled readers were significantly slower at low-level skills, such as letter and word encoding. This lack of automation of low-level skills led to inadequate discourse understanding, since memory for sentence wording decayed while the reader was trying to encode words.

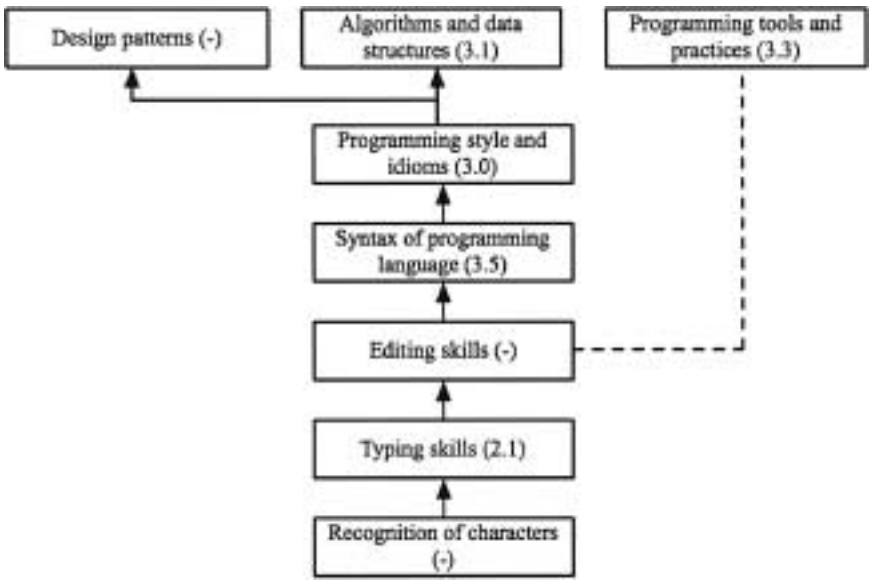

Fig. 1. Possible hierarchy of some low-level and intermediate-level programming skills or areas of knowledge. Numbers are means from respondents' answers in related skills.

In Fig. 1, we propose one possible hierarchy of some low-level and intermediate-level programming skills or areas of knowledge. On purpose, very high-level skills are not presented in Fig. 1 because we assume that these skills cannot be automated. We have added related means from the respondents' answers into Fig. 1. Next, the means are explained starting from the bottom of the figure: (a) In the boxes "Recognition of characters," "Editing skills," and "Design patterns" a dash (-) indicates that none of the results was really related. (b) The mean of the box "Typing skills" was presented in the body text in the beginning of Section 4.2. (c) The mean of the box "Syntax of programming languages" refers to Comment 2b in Table 2. (d) The mean of the box "Programming style and idioms" is a combined mean from Comments 2b, 7b, and 13 of Table 2. (e) The mean of the box "Algorithms and data structures" refers to Comment 6 in Table 2. (f) The mean of the box "Programming tools and practices" is a combined mean from Comments 2a, 2b, and 9 of Table 2, and Comment 8b of Table 3.

At the moment, we do not have a proper evidence to support the figure; we just give our arguments, which are based on our experience. First, it is obvious that all students of an undergraduate programming course know alphabets, and learning the necessary special characters (brackets, an asterisk, a tilde. . .) is not difficult for them. Second, good typing skills, especially the ability to concentrate on looking continuously on the screen, reduces interrupts in thinking. We believe that frequent interruptions in order to look at the keyboard distract the reasoning process when reading and constructing programming language idioms.

Third, an editor can significantly reduce the work load of programming by automating a number of issues. For example, Emacs has different modes for various programming

languages such that recognize the syntax, support automatic indentation, show the pairs of braces or brackets, and provide a number of ready-made keyboard commands to create various syntactical constructions. An experienced programmer can greatly benefit from these features if he/she knows them and frequently applies them. For example, automatic indentation not only saves the time to write the appropriate number of spaces but also easily points out possible errors when the indentation does not seem to work properly. Moreover, the keyboard commands are beneficial if a programmer wants to use a mouse as little as possible. We did not find any really relevant publications about typing or editing as part of programming but Fry (1997, p. 63) has written: "Switching between mouse and keyboard is bad. Most hackers I know think in terms of keyboard commands that perform equivalent mouse operations, so they don't have to switch to and from the mouse." This is just an anecdotal observation[3] but anyhow, we agree with this observation.

The fourth and fifth levels are concerned with knowledge on programming languages. Experienced programmers know the syntax and semantics of several languages by heart, which reduces their need to look at manuals and the number of syntactical and semantic errors they face when processing programs. What is more important is that they know how the language should be used to implement commonly appearing structures such as building linked lists.

In the box "Programming tools and practices" tools refer to editing, debugging, compiling, and version management tools such as Emacs, GDB, Make, and CVS. Practices refer to, for example, debugging and unit testing. We placed the box at the top part of the figure because some practices might be intermediate-level or high-level skills. The dashed line indicates that editors are related to editing skills.

Finally, the skills related to design patterns, and the selection of data structures and algorithms are intermediate-level or high-level skills. These skills are based both on knowledge about these issues and experience about what works well in practice.

5.4. *Future Research*

Next are mentioned three possible research settings that we think as interesting for a follow-up research. On purpose, only research settings that use the Delphi method are mentioned because our research is a Delphi study.

- The researchers of the psychology of programming could be asked as respondents, not experienced software developers. For example, the editors of the book Psychology of Programming might be possible candidates. It would be interesting to compare the results of these two respondent groups. It is possible that researchers of this field can mention some skills that software developers cannot – and vice versa. An experienced researcher of psychology of programming might mention, for example, 10–30 cognitive skills when a respondent of the present research mentioned only 3–5 skills.

[3]The questions concerning typing skills and use of editor were added to the second questionnaire as a consequence of another anecdotal observation. The first author of the present paper noticed that some respondents typed fast while they answered to the WWW version of the first questionnaire.

- The respondents could live in another country than in Finland because there might be some cultural differences. We assume that cultural differences related to the cognitive skills of software developers are small. However, it would be interesting to explore if this is the case.
- Also a third questionnaire round could be organized. In our research, we stopped after the second questionnaire round because we had promised to the respondents that participating would take 1–3 hours. We did not stop because we though that nothing interesting could be found during a third questionnaire round.

If a similar research will be repeated in the future, we suggest that (a) the division composition versus comprehension, and (b) the definition of cognitive skills that is given at the beginning of Section 1 will be used also in the questionnaires. In addition, we suggest that the first questionnaire will concentrate completely or mainly on cognitive skills. In our research, the questions about cognitive skills were only a small part of the first questionnaire.

Acknowledgements

We thank emeritus professor V. Meisalo from the University of Helsinki for suggesting the use of the Delphi method and PhD S. Kujala from the Helsinki University of Technology for commenting on a manuscript of the present paper.

Appendix A: Software Development Strategies

The following texts are quotations from (Détienne, 2002, pp. 26–28):

> **Top-down vs Bottom-up**
>
> A solution may be developed either top-down or bottom-up, that is from the more abstract to the less abstract or vice versa. In the first case the programmer develops the solution at an abstract level and then refines it, progressively adding more and more detail. In the second case, the solution is developed at a very detailed level before its more abstract structure is identified.
>
> **Forward vs Backward Development**
>
> A design strategy is described as forward development when the solution is developed in direction of execution of the procedure. It is described as backward if it is developed in the direction opposite to that of the execution of the procedure.
>
> **Breadth-First vs Depth-First**
>
> A breadth-first strategy means developing all the elements of the solution at one level of abstraction before proceeding to the next, more detailed, level of abstraction. A depth-first strategy means that one element of the system is developed to all levels of abstraction before any other element is developed.

Procedural vs Declarative

The development of a solution is said to be procedural when it is the structure of the procedure that controls the solution; the solution is then based on aims or procedures. The development is said to be declarative when static properties, such as objects and roles, control the solution.

Mental Simulation

Simulation can be used to evaluate a solution. In fact, designers often use mental simulation on a partial or complete solution at a higher or lower level of abstraction or on passages of code that they are seeking to understand. Simulation provides a way of verifying that a solution meets the desired objectives and a way of integrating partial solutions by controlling their interactions.

References

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, **18** (6), 543–554.

Capretz, L. (2003). Personality types in software engineering. *International Journal of Human-Computer Studies*, **58** (2), 207–214.

Conover, W. (1999). *Practical Nonparametric Statistics*, 3rd edition. John Wiley and Sons, New York.

Détienne, F. (2002). *Software Design – Cognitive Aspects*. Springer, London.

Engel, G., and E. Roberts (2001). *Computing Curricula 2001. Computer Science*. Final report, December 15, 2001. IEEE Computer Society and Association for Computing Machinery. Retrieved on October 28, 2004, from the IEEE Computer Society web site:
`http://www.computer.org/education/cc2001/cc2001.pdf`

ERIC Thesaurus (2004). Retrieved on April 27, 2004, from the Educator's Reference Desk web site:
`http://www.ericfacility.net/extra/pub/thessearch.cfm`

Fry, C. (1997). Programming on an already full brain. *Communications of ACM*, **40** (4), 55–64.

Greeno, J., and H. Simon (1988). Problem solving and reasoning. In R. C. Atkinson, R.J. Herrstein, G. Lindzey and R.D. Luce (Eds.), *Stevens Handbook of Experimental Psychology*, vol. 2.

Kitchenham, B., and S. Pfleeger (2002). Principles of survey research. Part 5: Population and samples. *Software Engineering Notes*, **27** (5), 17–20.

Milton, J., and J. Arnold (2003). *Introduction to Probability and Statistics*, 4th edition. McGrawHill, New York.

Stanislaw, H., B. Hesketh, S. Kanavaros, T. Hesketh and K. Robinson (1994). A note on the quantification of computer programming skill. *International Journal of Human-Computer Studies*, **41** (3), 351–362.

Surakka, S. (2004). *Supplementary Material for Article "Cognitive Skills of Experienced Software Developer: Delphi Study"*.
`URL: http://www.cs.hut.fi/u/ssurakka/papers/Delphi2/index.html`

Surakka, S., and L. Malmi (2004). Cognitive skills of experienced software developer: Delphi study. In A. Korhonen and L. Malmi (Eds.), *Kolin Kolistelut–Koli Calling 2004. Proceedings of the Fourth Finnish/Baltic Sea Conference on Computer Science Education*. Koli, Finland, pp. 37–46.

Visser, W., and J-M. Hoc (1990). Expert software design strategies. In Hoc, J.-M., T.R.G. Green, R. Samurçay and D.J. Gilmore (Eds.), *Psychology of Programming*. Academic Press, London.

Wiedenbeck, S. (1985). Novice/expert differences in programming skills. *International Journal of Man-Machine Studies*, **23** (4), 383–390.

Wilhelm, W. (2001). Alchemy of the Oracle: The Delphi technique. *The Delta Pi Epsilon Journal*, **43** (1), 6–26.

S. Surakka is a doctoral student at the Helsinki University of Technology where he received master's degree in 1993. The topic of his doctoral thesis is the evaluation of software systems education.

L. Malmi is a professor of computer science in the Helsinki University of Technology (HUT). He received his doctor of technology diploma in the HUT in 1997. His main research area is computer science education including software visualization, automatic assessment, new educational methods, and evaluating how they improve learning.

Patyrusių programinės įrangos kūrėjų kognityvinių įgūdžių tyrimas remiantis *Delphi*

Sami SURAKKA, Lauri MALMI

Straipsnyje aptariami patirtį turinčių programinės įrangos kūrėjų kognityvinių įgūdžių kokybinio tyrimo rezultatai. Tyrimui reikalingi duomenys surinkti pasinaudojant *Delphi* metodu. Respondentais buvo atrinkti 11 programuotojų, turinčių, – skaičiuojant nuo mokslų baigimo, – mažiausiai penkerius metus programuotojo darbo patirties. Tyrimu siekta išnagrinėti profesionalių programuotojų įgūdžius, todėl atrenkant respondentus buvo atsižvelgta į rekomendacijas. Tokiu būdu šis tyrimas nelaikytinas statistiškai reprezentatyviu visų programuotojų atžvilgiu, veikiau į jį reikėtų žvelgti kaip į konkrečios programinės įrangos kūrėjų grupės įgūdžių analizę. Apklausa vykdyta dviem ciklais. Atliekant pirmą apklausos ciklą respondentai paminėjo iš viso 32 skirtingus programuotojui reikiamus turėti įgūdžius. Atliekant antrąjį ciklą, 10 respondentų turėjo pateikti savo vertinimus apie kiekvieną iš šių 32 įgūdžių. Gauti rezultatai buvo suskirstyti į dvi kategorijas – programinės įrangos kūrimui ir supratimui reikiami įgūdžiai. Kiekvienas iš paminėtų įgūdžių buvo klasifikuotas pagal sudėtingumo laipsnį (pavyzdžiui, ar ši įgūdžių sudėtingumo gradacija gali būti veiksminga atliekant ekspertų ir pradedančiųjų diferenciaciją).