First publ. in: Cambridge handbook of expertise and expert performance / K. Anders Ericsson ... (eds.). Cambridge: University Press, 2006, pp. 373-387

Expertise in Software Design

Sabine Sonnentag, Cornelia Niessen, & Judith Volmer

Keywords: software design, programming, computer program, debugging, communication, knowledge representation, Professionals.

Introduction

In this chapter, we review research evidence on expertise in software design, computer programming, and related tasks. Research in this domain is particularly interesting because it refers both to rather general features and processes associated with expertise (e.g., knowledge representation, problem-solving strategies) and to specific characteristics of high performers in an economically relevant real-world setting. Therefore, in this chapter we draw on literature from various fields, mainly from cognitive psychology, but also from work and organizational psychology and from the software-design literature within computer science.

Our chapter is organized as follows: In the first main section we provide a brief description of the domain and give an overview of tasks in software development. Next, we briefly describe the expertise concept and distinguish between a conceptualization of expertise as years of experience and expertise as high performance. The third main section is the core part of this chapter. In this section, we review empirical research on expertise in tasks such as software design, programming, program comprehension, testing, and debugging. Moreover, we describe how expert performers differ from non-experts with respect to knowledge as well as communication and cooperation processes. In the final section, we present directions for future research and discuss some practical implications.

Historical Context

Extensive research on expertise on software design and programming started in the early 1980s. For example, Jeffries, Turner, Polson, and Atwood (1981) as well as Adelson (1984) published influential studies on how experts

differ from novices. These studies stimulated subsequent research and were often cited also in more general publications on expertise (e.g., Ericsson & Smith, 1991; Sternberg, 1996). Generally, research activity in the domain of software design and programming has been very lively and intensive (Hoffman, 1992). For many years, research on expertise in this domain was an important topic in the workshops on Empirical Studies of Programmers (Olson, Sheppard, & Soloway, 1987; Soloway & Iyengar, 1986). Since the 1990s, research interest seems to have shifted to investigations in more complex organizational settings (e.g., Campbell, Brown, & DiBello, 1992; Sonnentag, 1995; Turley & Bieman, 1995).

Studies on expertise in software design and programming do not aim only at scientific insight into the processes associated with expertise in a complex domain. Nowadays, as many countries' economy and people's life largely depend on information technology, the development and maintenance of high-quality software systems are of crucial relevance. Therefore, also with respect to practical implications it is important to examine how expert performance in software design and programming is achieved.

The Field of Software Design and Programming

The domain of software design and programming comprises distinct tasks, such as requirement analysis, software design, programming, testing, and debugging. It is the main goal of requirement analysis to specify the demands a new computer program or software system should meet. Ideally, future users of the computer system – or their representatives – should be involved in requirement analysis because they often have detailed knowledge about the tasks and processes the computer system should support. Software design aims at the description of the basic features of the future computer system and prescribes the functions the system should perform. During programming, the functions specified during software design are translated into a computer program, that is, a set of grammatical rules that refer to calculations, procedures, or objects. A main challenge for software design is to ensure that a computer program meets the requirements and runs without errors. This asks for extensive testing and debugging (i.e., error correction).

Traditionally, software design and programming has been conceptualized as a rather linear process, starting from requirement analysis and moving then to testing and debugging, with design and programming as distinct intermediate phases. However, such a linear approach was not feasible in practice because often not all requirements can be specified in advance. Therefore, nowadays a more iterative view dominates the field (Sommerville, 2001).

From a more psychological perspective, many software-design tasks can be described as ill-defined problems (Simon, 1973). Ill-defined problems imply that problem specifications are incomplete and have to be decided on during the design process. Thus, ill-defined problems have no single correct solution. Software design and programming draws largely on syntactic, semantic, and schematic knowledge (Détienne, 2002). A broad range of different design and programming strategies have been distinguished, such as top-down versus bottom-up, forward versus backward development, or breadth-first versus depth-first (Détienne, 2002). Therefore, psychological research on software design and programming has focused mainly on designers' and programmers' cognitive processes.

However, it has been argued that professional software design and programming ("programming in the large") takes place in teams and other cooperative settings and, therefore, an exclusive focus on the cognitive processes of individuals can fall short when describing software design and programming (Curtis, 1986; Riedl, Weitzenfeld, Freeman, Klein, & Musa, 1991). To understand fully how high-quality software is produced research should also address communication and cooperation processes.

Before describing characteristics of expert software designers and programmers in detail, we introduce two conceptualizations of expertise.

Conceptualizations of Expertise in Software Design and Programming

As in many other domains, researchers in the domain of software design and programming have relied on various conceptualizations and operationalizations of expertise. Although at the conceptual level most researchers would probably agree with Ericsson and Smith's (1991) definition of expertise as "outstanding performance" (p. 2), in most empirical studies, expertise has been operationalized as years of experience. For example, Sonnentag (2001b) reported that 84 percent of all quasi-experimental studies on expertise in software development and programming published between 1981 and 1997 used an operationalization of expertise that referred to months and years of experience. In these studies, beginning students have been compared with graduate students or professional programmers (e.g., Jeffries et al., 1981). With respect to relatively inexperienced persons, it is plausible to equate years of experience with increasing expertise and to assume that expertise develops as a function of time spent within the domain.

However, when it comes to more-advanced software designers and programmers, long years of experience are not necessarily related to a high performance level (Sonnentag, 1995, 1998; Vessey, 1986). This result mirrors findings on expert performance in other domains where years of experience have been shown to be poor predictors of high performance (Ericsson, Krampe, & Tesch-Römer, 1993). Thus, at the professional level, high performers are not necessarily more experienced than moderate performers. Therefore, it is important to differentiate between the conceptualization of expertise as *(long) experience* and of expertise as *high performance*. Differences found between novices on the one hand and more-experienced programming students or software professionals on the other hand may not easily generalize to differences between highly performing and moderately performing software professionals (Sonnentag, 2000). Of course, there are also large variations within the group of inexperienced persons (ranging from novices, who are undergoing introductory training, to apprentices and journeymen) and within the group of moderate performers.

After now having defined the expertise concept, in the next section we will summarize research evidence on differences between experts and non-experts.

Empirical Studies on Software Design and Programming

In this section, we give an overview over results from empirical studies on expert software design and programming. More specifically, we describe differences between experts and non-experts in five areas: (1) requirements analysis and design tasks, (2) programming and program comprehension, (3) testing and debugging, (4) knowledge representation and recall, and (5) communication and cooperation. Major findings are summarized in Table 21.1.

We focus on studies that used a contrastive design comparing a group of experts with a group of non-experts (Voss, Fincher-Kiefer, Greene, & Post 1986) and on other studies in which study participants differed with respect to the level of expertise or performance. We excluded papers from our review that provided case studies of single experts without comparing them to non-experts (e.g., Campbell et al., 1992).

Requirement Analysis and Software Design

Requirements analysis aims at identifying the demands a future software system should meet. It includes tasks such as analysing the problem requirements, setting design goals, and decomposing the goals to come up with a software design representing

Table 21.1. Comparison between experts and non-experts in software design and programming

	Experienced persons – as compared to inexperienced persons	*High performers – as opposed to moderate performers*
Requirement analysis and design	• Spend more time and effort on problem comprehension • Spend more time on clarifying program requirements • Decompose design problems in small components in a top-down breadth-first manner	• Spend less time on problem comprehension • Adequate problem representation early in the design process
Program comprehension and programming	• Pursuit of abstract programming goals • Program comprehension based on abstract concepts	• Pursuit of abstract programming goals • Cross-referencing strategy
Testing and debugging	• Test for inconsistency • Hypothesis-driven procedure	• Active search for problems
Knowledge	• Knowledge organized in greater and more meaningful chunks • Knowledge of abstract concepts • Superior meta-cognitive knowledge	• Broader and more detailed knowledge base
Communication and cooperation	<not studied>	• Spend more time on communication and cooperation

a solution (Koubek, Salvendy, Dunsmore, & LeBold, 1989; Malhotra, Thomas, Carroll, & Miller, 1980). Early studies showed that there are considerable individual differences in software design and that activities tend to vary according to the design experiences of programmers (e.g. Jeffries et al., 1981; Weinberg & Schulman, 1974). Studies focused on the comparison of inexperienced programmers (e.g., students) and experienced programmers (e.g., professionals; Agarwal, Sinha, & Tanniru, 1996), rather than on the comparison of specific performance levels (e.g., Sonnentag, 1998).

In real-world design situations, an important part of the design process consists of analysing the requirements the new software system should comply with. These requirements are often ill defined and not clearly stated by clients or potential users. Thus, the first task of the software designer is to set and refine the design goals (Chevalier & Ivory, 2003; Malhotra et al., 1980). Similar to findings from early studies in physics problem solving (e.g., Chi, Feltovich, & Glaser, 1981), studies have indicated that experienced software designers spend more time on clarifying the program requirements compared to students (Batra & Davis, 1992; Jeffries et al., 1981). Moreover, within the web-design domain, professionals included more requirements in their web design than less-experienced designers (Chevalier & Ivory, 2003). Even when specific clients' requirements were not explicitly stated, professionals inferred more requirements of the client on the basis of their prior experience. In a study with performance level as expertise criterion, Sonnentag (1998) found no differences between high and moderate performers in analysing requirements in early phases of the design process. Moderate performers spent even more time analyzing requirements in later phases of the design process

compared to high performers. High performers might have developed an adequate problem representation early in the design process, whereas moderate performers had to do additional requirement analysis on the later stages of the design process.

A lot of research has focused on the ability to decompose the design problem into smaller components. Here, a consistent result across many studies, albeit with very small samples, is that experts decomposed the design problem in a top-down and breadth-first manner (Adelson & Soloway, 1985). For example, Jeffries et al. (1981) described that experts decomposed a problem in major parts through a number of iterations. Also, advanced novices relied on such a top-down and breadth-first solution strategy, but with fewer iterations and therefore less detail.

A common assumption is that the decomposition process is guided by a knowledge representation, which is built up with experience and stored in long-term memory (Jeffries et al., 1981). When such a knowledge representation is not available, experts develop plans through a bottom-up, backward strategy (Rist, 1991). According to Rist (1986), experts have stored a broad range of programming plans that comprise knowledge of abstract and specific, language-dependent code fragments. This idea is supported by the study of Sonnentag (1998). When software designers were asked about strategies they would recommend to an inexperienced programmer working on the same design problem, high performers reported twice as many strategies as moderate performers. The studies of Davies (1991) showed that experienced programmers did not necessarily have more plans available, but focused on the most salient parts of the plan and flexibly switched between plans. In contrast, novices preferred a strategy with which plans are implemented in a more linear fashion.

With the emergence of new programming approaches (e.g., object-oriented programming), studies examined the role of knowledge in the design activity of experts using a new programming approach. These studies indicated that design strategies were influenced toward the specific language experiences (Agarwal et al., 1996). Though experts tried to obtain a deep understanding of a new language, they still used analogies to languages they knew when generating a program within a new framework (Campbell et al., 1992; Scholz & Wiedenbeck, 1992). Furthermore, research showed that when participants were experienced in one language (in object-oriented programming), complex plans were developed on the basis of the deep structure of this programming language. Inexperienced designers in object-oriented programming developed plans on the basis of functional similarity and showed more plan revisions (Détienne, 1995).

It thus seems that the knowledge base of experts is highly language dependent. However, experts also have abstract, transferable knowledge and skills. Agarwal et al. (1996) compared 22 computer scientists (more than two years of experience in process-oriented approaches) and 24 business students (limited knowledge in process-oriented modeling). When solving a design problem within a new non-process-oriented programming approach (here, object-oriented programming), experts did not differ from novices in the quality of solutions (structure and behavior of the program), and they showed even poorer performance on a finer level (processing aspects). Also Collani and Schömann (1995) examined the acquisition of a new programming language. In contrast to the study of Agarwal et al. (1996), the criterion for expertise was the programmers' performance. Their results supported the hypothesis that high performers possess language-independent abstract programming skills and knowledge that can be transferred faster and with less errors to the new programming framework.

The contrary results of the studies that defined expertise in terms of the length of experience versus performance can be explained by the different kinds of expertise: routine and adaptive expertise (Hatano & Inagaki, 1986). Individuals characterized by routine expertise show superior performance in well-known, often highly

routinized tasks, whereas individuals characterized by adaptive expertise are able to master novel tasks and are successful in transferring existing knowledge and skills to unknown tasks or settings. For example, participants in the study of Agarwal et al. (1996) could have developed routine expertise through extended practice with process-oriented design, which has rigidifying effects on learning new skills and knowledge. On the other hand, the upper performance group in the study of Collani and Schömann (1995) might be characterized by the concept of adaptive expertise. Such experts have a deeper conceptual understanding of the domain that allows for a transfer of knowledge and skills to novel tasks (Hatano & Inagaki, 1986; Kimball & Holyoak, 2000). This might also lead to a faster in-depth requirement analysis that the high performers in the above mentioned study of Sonnentag (1998) have shown.

Most studies have focused on one aspect of the design activity. One exception is the study of Sonnentag (1998) that analysed a broad range of design activities: requirement analysis, feedback processing, planning, task focus, visualizations, and knowledge. According to the results, expertise in software design was characterized by more local planning activities, more feedback processing, less task-irrelevant cognitions, more solution visualizations, and more knowledge about design strategies experts would recommend to an imagined inexperienced colleague.

After the requirements have been analyzed and the software design has been produced, the concepts have to be implemented in computer programs. Therefore, in the next paragraphs we review research findings on differences between experts and non-experts with respect to programming and program comprehension.

Programming and Program Comprehension

Programming refers to the process of "translating" specifications of a calculation or a procedure into a computer program. Most studies on programming have compared relatively inexperienced students with more-experienced programmers (Bateson, Alexander, & Murphy, 1987; Davies, 1990; Soloway & Ehrlich, 1984). These studies clearly showed that more advanced programmers outperformed the relatively inexperienced students with respect to performance quality (Bateson et al., 1987; Soloway & Ehrlich, 1984) and solution time (Davies, 1990). When comparing high performers with moderate performers within a group of professional programmers, the high performers followed more abstract goals but did not differ with respect to other process features (Koubek & Salvendy, 1991).

Program comprehension is a necessary prerequisite for successfully completing programming tasks. Moreover, also testing and debugging tasks cannot be satisfactorily accomplished without understanding the program. In addition, program comprehension is of crucial importance in the context of professional software maintenance and software reuse. Nearly all empirical studies that have examined program comprehension have focused on the comparison between inexperienced and more-experienced persons (Adelson, 1984; Guerin & Matthews, 1990; Rist, 1996; Widowski & Eyferth, 1985; Wiedenbeck, Fix, & Scholtz, 1993). These studies showed that experienced persons perform better on program comprehension tasks and appear to be superior with respect to specific aspects of comprehension. For example, Adelson (1984) found that experienced programmers provided better answers to abstract questions than did students. Similarly, Rist (1996) reported that experienced graduates categorize programs according to programming plans. Thus, program comprehension of more-experienced persons is based on more-abstract concepts.

Pennington (1987) compared highly and poorly performing professional programmers. When trying to understand a program, high performers showed a "cross-referencing strategy" characterized by systematic alterations between systematically studying the computer program, translating it to domain terms, and subsequently verifying domain

terms back in program terms. In contrast, poorer performers exclusively focused on program terms *or* on domain terms without building connections between the two "worlds." Thus, it seems that connecting various domains and relating them to each other is crucial for arriving at a comprehensive understanding of the program and the underlying problem.

Designing and programming software is only half of the story of software development. Finding and correcting errors in the software before introducing the software to the market is crucial. In the next paragraphs we will therefore summarize research evidence in the area of testing and debugging.

Testing and Debugging

Software testing aims at identifying inconsistencies and errors within computer programs. Debugging refers to the process of removing errors, so-called *bugs*, from a computer program. Empirical studies on testing and debugging have focused on comparisons between relatively inexperienced and more-experienced persons (Nanja & Cook, 1987; Teasley, Leventhal, Mynatt, & Rohlman, 1994; Weiser & Shertz, 1983). Not surprisingly, these studies illustrated that experienced students and professionals found more errors and detected the errors more quickly than did novices (Law, 1998; Weiser & Shertz, 1983). With respect to the testing processes, studies showed that experienced persons tested more for inconsistency (Teasley et al., 1994) and proceeded in a more hypothesis-driven and concept-oriented way (Krems, 1995). Thus, as students' programming experience increased, they seemed to work in a more systematic way.

Few studies have compared highly performing software professionals with those who perform at a moderate level. A thinking-aloud study showed that high performers needed less time for debugging, made fewer errors, searched more intensively for problems, and showed more information-evaluation activity (Vessey, 1986). Thus, it seems that high performers have a better representation of the program and potential problems, which helps them to actively search for problems and to evaluate information. In the next paragraphs we discuss experts' knowledge and knowledge representations in greater detail.

Knowledge and Knowledge Representation

Knowledge and adequate knowledge representation are crucial in all phases of software development. Researchers have investigated knowledge and knowledge representation by using recall, problem-sorting, and other tasks. For example, study participants have been asked to recall program lines, either presented in a meaningful or in an arbitrary order (Barfield, 1986; McKeithen, Reitman, Rueter, & Hirtle, 1981). Again, most studies have contrasted inexperienced with more-experienced persons (Eteläpelto, 1993; McKeithen et al., 1981; Weiser & Shertz, 1983; Ye & Salvendy, 1994).

There is consistent evidence that novices and more-advanced students or professionals differ with respect to knowledge and knowledge representation. In recall tasks, more-experienced persons recalled more program lines, often only when the lines were presented in a meaningful order (Barfield, 1986, 1997; McKeithen et al., 1981), but sometimes also when presented in an arbitrary order (Bateson et al., 1987; Guerin & Matthews, 1990). Thus, it seems that experienced persons have organized their knowledge in greater and more meaningful "chunks" (Adelson, 1984; Ye & Salvendy, 1994). However, differences in chunking do not seem to be sufficient to explain experienced persons' better performance when recalling program lines presented in an arbitrary order.

Research has shown that professionals also possess more meta-cognitive knowledge than do students (Eteläpelto, 1993). Eteläpelto assessed students' and professional programmers' meta-cognitive knowledge during an interview that focused on a specific computer program and on the

respondents' more general working strategies. Professional programmers' metacognitive knowledge was much more detailed and more comprehensive than students' knowledge. For example, a professional programmer provided a comment such as "The main program is not very well divided; the logic has been scattered over different parts of the program; the updating paragraph has too much in it. The control function of the program should be implemented in one paragraph; now the program has much depth in itself and scanning is difficult" (p. 248f.), whereas the students' comments were more general and diffuse. In addition, professional programmers expressed a clear idea about a good strategy of reading and comprehending computer programs. Thus, experienced persons know more about how to proceed when solving tasks in their domain.

When it comes to knowledge representation in highly versus moderately performing software professionals, high performers' superior knowledge seems not restricted to knowledge about how to solve rather narrowly defined software-design tasks (Sonnentag, 1998). Highly performing software professionals also possess more detailed knowledge about how to approach cooperation situations (Sonnentag & Lange, 2002). Thus, it seems that as work tasks become more comprehensive in real-world settings, high performers develop a more comprehensive representation of the entire work task, including necessary features of cooperation with coworkers.

In the next paragraphs we describe the role of communication and cooperation in expert performance in more detail.

Communication and Cooperation

Most research on experts has focused on specific activities in software design, such as design, programming, or testing in individual task settings. Nevertheless, some studies have addressed the question whether there are differences between high- and average-performing software professionals in cooperative work settings. Modern professional software development takes place mainly in project teams, suggesting that expertise in software design is not only a matter of knowledge and task strategies but also requires social and communicative skills.

Field studies suggest that high performers show better communication and cooperation competencies than moderate performers (e.g. Curtis, Krasner, & Iscoe, 1988; Kelley & Caplan, 1993; Sonnentag, 1995). In a field study with 17 software-development projects, Curtis et al. (1988) found that exceptional software designers showed superior communication skills. In fact, much of the exceptional designers' design work was accomplished while interacting with other team members. High performers spent a lot of time educating other team members about the application domain and its mapping into computational structures. Similarly, Sonnentag (1995), using a peer-nomination method in a study with 200 software professionals, showed that experts did not spend more time on typical software-development activities such as design, coding, or testing, but were more often engaged in review meetings and spent more time in consultations than did other team members. In a study at the Bell Laboratories, Kelley and Caplan (1993) compared top performers to average performers and found that differences could not be attributed to cognitive ability and that top performers considered interpersonal network abilities as highly important. Furthermore, top performers possessed better functioning interpersonal networks compared to average performers. Riedl et al. (1991) also observed highly developed interpersonal skills in expert software engineers. In a study on behavior in team meetings, Sonnentag (2001a) found that highly performing software professionals' involvement in cooperation situations differed, depending on situational demands. In highly structured meetings, no differences between highly and average performing software professionals were found, whereas differences were observed in poorly structured meetings. Thus, experts showed high adaptation to specific situational constraints and displayed cooperation competencies, particularly in difficult situations. Interestingly,

high performers' communication behavior in unstructured cooperation situations mirrored their behavior when working on design problems in individual settings. More specifically, high performers showed process-regulating behavior such as planning or feedback seeking when the situation demands it, that is, when the meeting was unstructured, but not when it was highly structured.

What work strategies do exceptional software professionals recommend to an inexperienced colleague? This question was approached in a study with 40 software professionals (Sonnentag, 1998). In this study, high performers, more often than moderate performers, mentioned that the imagined inexperienced coworker should cooperate with others. Campbell and Gingrich (1986) demonstrated that this can be a useful strategy. They found that individual performance of software developers is facilitated significantly by a 15-minute talk with a senior designer about their task. These results hold for complex tasks, not just for simple ones, indicating that high task complexity makes it more necessary and more valuable for the programmer to communicate with a more senior person. In a more recent study, Ferris, Witt, and Hochwarter (2001) showed that social skill was more strongly related to job performance among programmers high in general mental ability (GMA) than for those with average or low levels of GMA. Thus, social skills in combination with high mental abilities – not high mental abilities by themselves – are related to the highest level of performance.

Several questions remain open since communication and cooperation processes in software design are rarely studied. It is not clear whether communication and cooperation competencies are a prerequisite of excellent software performance or a positive by-product of being an excellent software designer. Furthermore, third variables might cause the relationship between communication and cooperation competencies and excellent software performance (e.g., generalized problem-solving skills). Additional consideration of possible moderators, such as task or situational characteristics, might also illuminate this relationship, especially in professional software design with changing work requirements. In sum, more studies are needed to examine the causal relationships between expertise and communication skills and how these differences matter in complex real-world settings.

Directions for Future Research

Despite the growing research evidence on how experts differ from non-experts, several questions remain open that should be addressed in future research. In our view, the most important ones refer to: (1) the conceptualization of expertise, (2) the issue of causality, (3) the role of task and situational characteristics, (4) the role of motivation and self-regulation, and (5) the question of how expertise develops.

(1) THE CONCEPTUALIZATION OF EXPERTISE

Most studies on expertise in software design and programming refer to differences between inexperienced and more-experienced persons and compare beginning students with more-advanced students or professionals. Rather few studies examined how highly performing software professionals differ from software professionals performing less well (e.g., Koubek & Salvendy, 1991; Pennington, 1987; Sonnentag, 1998, 2001a; Vitalari & Dickson, 1983).

Of course, studies that contrast inexperienced with more-experienced persons provide useful insights about how skills may develop in a domain. However, it would be premature to assume that findings from comparisons between inexperienced and more-experienced persons can be easily generalized to differences within the group of professionals. Particularly in the domain of software design and programming, where specific knowledge becomes obsolete very quickly, long years of experience do not ensure high performance. Therefore, future studies should examine *professional* software designers and programmers and should operationalize expertise as high performance.

(2) THE ISSUE OF CAUSALITY

Most studies on expertise do not answer the question of causality because most empirical studies use a quasi-experimental or a correlational design. True experiments in which the independent variables are manipulated are very rare. Strictly speaking, without experimental manipulation it is impossible to falsify causal hypotheses. For example, although it is plausible that experts' pursuit of more abstract goals (Koubek & Salvendy, 1991) *leads* to their superior performance, it can not be ruled out that the focus on more abstract goals is a *consequence* of expertise. Similarly, experts' specific approach to communication and cooperation (Curtis et al., 1988; Sonnentag, 1995) may help them to arrive at their high performance level. However, experts' communication and cooperation processes might also be a consequence – or even only a by-product – of their superior performance. Because of their superior ability to master the intellectual demands of their tasks, experts might have more cognitive resources available to spend on communication. Thus, to arrive at a deeper understanding of the causal processes involved in expert performance, experiments are needed.

(3) THE ROLE OF TASK AND SITUATIONAL CHARACTERISTICS

Most studies have examined differences between experts and non-experts by using rather simple tasks with questionable external validity. Only a few studies looked at tasks that took two hours or longer to accomplish (exceptions are Sonnentag, 1998; Vitalari & Dickson, 1983). In real-world settings, software tasks can be laborious and time consuming, and therefore may involve other and more complex processes than those that have been examined. For example, future studies should use more-complex real-world tasks that require the coordination and prioritization of various subtasks. Expertise research should also pay more attention to the specific task demands placed on professional software developers. Particularly in areas such as web design and user-interface design, software designers have to deal with multiple constraints and must take economic, ergonomic, and domain-specific demands into account. Until now, there is not much insight into the strategies expert performers use in order to integrate multiple and sometimes conflicting constrains into their design. Future studies should address this gap.

Moreover, the role of specific task features and other situational characteristics that might enable or hinder expert performance has not been addressed systematically. There is some evidence that the differences observed between experts and non-experts are not obvious for all tasks and in all situations (Sonnentag, 2001a). For example, it might be that experts excel over non-experts only when working on complex tasks, but not when working on simpler tasks.

In addition, when completing tasks in real-world work settings, individuals are often faced with adverse and stressful conditions, such as work overload, time pressure, and role ambiguity (Fujigaki & Mori, 1997; Glass, 1997). It would be an interesting question for future research to study how experts and non-experts differ when confronted with stressful conditions. One may speculate that experts and non-experts differ in their perception on what constitutes stress and in how they deal with stressful conditions. Experts' und non-experts' differential reactions to stressful situations would imply that differences between the two groups become more evident under stressful conditions than under more relaxed conditions. Furthermore, Ericsson and Lehmann (1996) suggested that experts show a better adaptation to task constraints. This would imply that experts are not only less bothered by unfavorable situations but develop better ways to cope with them.

(4) THE ROLE OF MOTIVATION AND SELF-REGULATION

Expertise research has largely concentrated on cognitive issues associated with expert performance and has recently extended its focus to communication and cooperation

skills. Until now, motivational and self-regulatory issues that might be highly relevant for high performance were beyond the research interest of most scholars in the field (But see Ericsson, Chapter 38). Empirical studies from other areas have shown that self-efficacy and the pursuit of difficult and specific goals are closely related to high performance levels (Latham, Locke, & Fassina, 2002; Locke & Latham, 1990; Stajkovic & Luthans, 1998). Moreover, it has been found that high performers verbalize less task-irrelevant cognitions during software design and focus more on the task at hand (Sonnentag, 1998). One interpretation is that high performers have superior self-regulatory skills that inhibit task-irrelevant thoughts. It would be a highly interesting avenue for future research to examine how motivation and self-regulation affect expert performance.

(5) THE QUESTION OF HOW EXPERTISE DEVELOPS

The question of how expertise develops is both highly interesting and practically relevant. However, our knowledge about how expertise in the software domain develops is very limited. Of course, as students progress from a novice level to a graduate student or even professional level, performance normally increases. However, within the group of software professionals, results on the empirical relationship between years of experience and performance are inconclusive. Although some studies have found positive relationships between years of experience and performance (Koubek & Salvendy, 1991; Turley & Bieman, 1995), others have not (Sonnentag, 1995, 1998; Vessey, 1986). There is some evidence that specific aspects of experience, such as its breadth and variety, are related to expert performance (Sonnentag, 1995). Ericsson et al. (1993) have argued that deliberate practice (i.e., regularly pursued purposeful and effortful learning and practice activities) is crucial for the achievement and maintenance of expert performance. Empirical studies have shown that deliberate practice indeed is related to high performance in domains such as music or sports (Davids, 2000; Ericsson et al., 1993; Krampe & Ericsson, 1996) and also in more "classical" work settings such as insurance companies (Sonnentag & Kleine, 2000). We assume that deliberate practice might play also a core rule in the development of expert performance in software design and programming.

An issue closely related to the development of expertise refers to the distinction between routine and adaptive expertise (Hatano & Inagaki, 1986). In areas such as the software domain, where new tools and methodologies continuously emerge and where existing knowledge can become obsolete very quickly, more research on the development of adaptive expertise is particularly needed.

To adequately investigate the development of expertise in professional domains, longitudinal field studies are needed. Although retrospective studies can offer some interesting insights, only longitudinal studies that cover several years will allow conclusive answers about how expertise develops and unfolds over time.

Practical Implications

When it comes to practical approaches to promote expert performance in work settings, two major approaches have to be considered: (1) personnel selection and (2) training. Both approaches are based on the assumption that the characteristics typical for experts causally contribute to their superior performance. Most generally, personnel selection refers to organizational procedures and specific methods in order to select those individuals who may be well suited for a specific job. Based on the literature review provided in this chapter, personnel selection procedures – at least after some domain-specific training is accomplished – should include measures of knowledge and knowledge organization and of domain-specific problem-solving strategies that focus on abstract concepts and goals. In addition, assessments of communication and cooperation skills should be part of the

personnel-selection process. In addition to more standardized tests (Stevens & Campion, 1999), group discussions or other teamwork assignments within an assessment center might be used. With respect to experience as a predictor in personnel selection, the situation is more complicated. Of course, for most jobs one would prefer to select individuals with professional experience over beginning students. However, when having to select individuals from a group of professionals – which would be more often the case – length of experience does not seem to be a very reliable predictor of expert performance. Other aspects of experience, such as experience variety, might be more helpful (Sonnentag, 1995).

In addition to personnel selection, training is a promising approach to the management of expert performance (Hesketh & Ivancic, 2002). Training should help individuals to develop adequate mental models of typical problems in the domain and to choose the most appropriate working strategy. Our literature review suggests that training should focus on domain-specific and meta-cognitive knowledge as well as on abstract planning and evaluation strategies. Effective communication and cooperation skills should be taught and deepened through practical exercises. In our view, it is important that training in communication and cooperation begins at the university because at present training in computer science and programming often relies exclusively on technical knowledge and skills and neglects communication and cooperation skills that seem to be particularly important for expert performance in professional settings.

References

Adelson, B. (1984). When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 10*, 483–495.

Adelson, B., & Soloway, E. (1985). The role of domain experience in software design. *IEEE Transactions on software engineering, 11*, 1351–1360.

Agarwal, R., Sinha, A. P., & Tanniru, M. (1996). The role of prior experience and task characteristics in object-oriented modeling: An empirical study. *International Journal of Human-Computer Studies, 45*, 639–667.

Barfield, W. (1986). Expert-novice differences for software: Implications for problem-solving and knowledge acquisition. *Behaviour & Information Technology, 5*, 15–29.

Barfield, W. (1997). Skilled performance on software as a function of domain expertise and program organization. *Perceptual and Motor Skills, 85*, 1471–1480.

Bateson, A. G., Alexander, R. A., & Murphy, M. D. (1987). Cognitive processing differences between novice and expert computer programmers. *International Journal of Man-Machine Studies, 26*, 649–660.

Batra, D., & Davis, J. G. (1992). Conceptual data modelling in database design: Similarities and differences between expert and novice designers. *International Journal of Man-Machine Studies, 37*, 83–101.

Campbell, D. J., & Gingrich, K. F. (1986). The interactive effects of task complexitiy and participation on task performance: A field experiment. *Organizational Behavior and Human Decision Processes, 38*, 162–180.

Campbell, R. L., Brown, N. R., & DiBello, L. A. (1992). The programmer's burden: Developing expertise in programming. In R. R. Hoffman (Ed.), *The psychology of expertise* (pp. 269–362). New York: Springer.

Chevalier, A., & Ivory, M. Y. (2003). Web site designs: Influences of designer's expertise and design constraints. *International Journal of Human-Computer Studies, 58*, 57–87.

Chi, M. T. H., Feltovich, P. J., & Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. *Cognitive Science, 5*, 121–152.

Collani, G. V., & Schömann, M. (1995). The process of acquisition of a new programming language (LISP): Evidence for transfer of experience and knowledge in programming. In K. F. Wender, F. Schmalhofer & H.-D. Boecker (Eds.), *Cognition and computer programming* (pp. 169–191). Norwood, NJ: Ablex.

Curtis, B. (1986). By the way, did anyone study any real programmers? In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 256–262). Norwood, JN: Ablex.

Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM*, 31, 1268–1287.

Davids, K. (2000). Skill acquisitaion and the theory of deliberate practice: It ain't what you do it's the way you do it *International Journal of Sport Psychology*, 31, 461–466.

Davies, S. P. (1990). Plans, goals and selection rules in the comprehension of computer programs. *Behaviour & Information Technology*, 9, 201–214.

Davies, S. P. (1991). The role of notation and knowledge representation in the determination of programming strategy: A framework for integrating models of programming behavior. *Cognitive Science*, 15, 547–572.

Détienne, F. (1995). Design strategies and knowledge in object-oriented programming: Effects of experience. *Human-Computer Interaction*, 10, 129–169.

Détienne, F. (2002). *Software design. Cognitive aspects*. London, UK: Springer.

Ericsson, K. A., Krampe, R. T., & Tesch-Römer, C. (1993). The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, 100, 363–406.

Ericsson, K. A., & Lehmann, A. C. (1996). Expert and exceptional performance: Evidence of maximal adaptation to task constraints. *Annual Review of Psychology*, 47, 273–305.

Ericsson, K. A., & Smith, J. (1991). Prospects and limits of the empirical study of expertise: An introduction. In K. A. Ericsson & J. Smith (Eds.), *Toward a general theory of expertise: Prospects and limits* (pp. 1–38). Cambridge: Cambridge University Press.

Eteläpelto, A. (1993). Metacognition and the expertise of computer program comprehension. *Scandinavian Journal of Educational Research*, 37, 243–254.

Feltovich, P. J., Spiro, R. J., & Coulson, R. L. (1997). Issues of expert flexibility in contexts characterized by complexity and change. In P. J. Feltovich, K. M. Ford, & R. R. Hoffman (Eds.), *Expertise in Context: Human and Machine* (pp. 125–146). Menlo Park, CA: MIT Press.

Ferris, G. R., Witt, L. A., & Hochwarter, W. A. (2001). Interaction of social skill and general mental ability on job performance and salary. *Journal of Applied Psychology*, 86, 1075–1082.

Fujigaki, Y., & Mori, K. (1997). Longitudinal study of work stress among information system professionals. *International Journal of Human-Computer Interaction*, 9, 369–381.

Glass, R. L. (1997). The ups and downs of programmer stress. *Communications of the ACM*, 40(4), 17–19.

Guerin, B., & Matthews, A. (1990). The effects of semantive complexity on expert and novice computer program recall and comprehension. *The Journal of General Psychology*, 117, 379–389.

Hatano, G., & Inagaki, K. (1986). Two courses of expertise. In H. Stevenson, A. Azuma, & K. Hakuta (Eds.), *Child development and education in Japan* (pp. 262–272). San Francisco: Freeman.

Hesketh, B., & Ivancic, K. (2002). Enhancing performance through training. In S. Sonnentag (Ed.), *Psychological management of individual performance* (pp. 247–265). Chichester: Wiley.

Hoffman, R. R. (1992). Bibliography: Expertise in programming. In R. R. Hoffman (Ed.), *The psychology of expertise: Cognitive research and empirical AI* (pp. 359–362). Hillsdale, NJ: Erlbaum.

Jeffries, R., Turner, A. A., Polson, P. G., & Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 255–283). Hillsdale, NJ: Erlbaum.

Kelley, R., & Caplan, J. (1993). How Bell Labs creates star performers. *Harvard Business Review*, 71, 128–139.

Kimball, D. R., & Holyoak, K. J. (2000). Transfer and expertise. In E. Tulving & F. I. M. Craik (Eds.), *The Oxford handbook of memory* (pp. 109–122). Oxford: Oxford University Press.

Koubek, R. J., & Salvendy, G. (1991). Cognitive performance of super-experts on computer program modification tasks. *Ergonomics*, 34, 1095–1112.

Koubek, R. J., Salvendy, G., Dunsmore, H. E., & LeBold, W. K. (1989). Cognitive issues in the process of software development: Review and rappraisal. *International Journal of Man-Machine Studies*, 30, 171–191.

Krampe, R. T., & Ericsson, K. A. (1996). Maintaining excellence: Deliberate practice and elite performance in young and older pianists. *Journal of Experimental Psychology: General, 125*, 331–359.

Krems, J. F. (1995). Expert strategies in debugging: Experimental results and a computational model. In K. F. Wender, F. Schmalhofer, & H.-D. Boecker (Eds.), *Cognition and computer programming* (pp. 241–254). Norwood, NJ: Ablex.

Latham, G. P., Locke, E. A., & Fassina, N. E. (2002). The high performance cycle: Standing the test of time. In S. Sonnentag (Ed.), *Psychological management of individual performance* (pp. 201–228). Chichester: Wiley.

Law, L.-C. (1998). A situated cognition view about the effects of planning and authourship on computer program debugging. *Behaviour & Information Technology, 17*, 325–337.

Locke, E. A., & Latham, G. O. (1990). *A theory of goal setting and task performance*. Englewood Cliffs, NJ: Prentice Hall.

Malhotra, A., Thomas, J. C., Carroll, J. M., & Miller, L. A. (1980). Cognitive processes in design. *International Journal of Man-Machine Studies, 12*, 119–140.

McKeithen, K. B., Reitman, J. S., Rueter, H. H., & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology, 13*, 307–325.

Nanja, M., & Cook, C. R. (1987). An analysis of the on-line debugging process. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical studies of programmers. Second workshop* (pp. 172–184). Norwood, NJ: Ablex.

Olson, G. M., Sheppard, S., & Soloway, E. (Eds.) (1987). *Empirical studies of programmers: Second workshop*. Norwood, NJ: Ablex.

Pennington, N. (1987). Comprehension strategies in programming. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical studies of programmers: Second workshop* (pp. 100–113). Norwood, NJ: Ablex.

Riedl, T. R., Weitzenfeld, J. S., Freeman, J. T., Klein, G. A., & Musa, J. (1991). What we have learned about software engineering expertise. *Proceedings of the Fifth Software Engineering Institute Conference on Software Engineering Education* (pp. 261–270). New York: Springer.

Rist, R. S. (1986). Plans in programming: Definition, demonstration and development. In E. Soloway & R. Iyengar (Eds.), *Empirical studies of programmers*. Norwood, NJ: Abley.

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

Rist, R. S. (1996). System structure and design. In W. D. Gray & D. A. Boehm-Davis (Eds.), *Empirical studies on programmers: Sixth workshop*. Norwood, NJ: Ablex.

Scholtz, J., & Wiedenbeck, S. (1992). Learning new programming languages: An analysis of the process and problems encountered. *Behaviour & Information Technology, 11*(4), 199–215.

Simon, H. A. (1973). The structure of ill-structured problems. *Artificial Intelligence, 4*, 181–204.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering, 10*, 595–609.

Soloway, E., & Iyengar, S. (Eds.) (1986). *Empirical studies of programmers*. Norwood, NJ: Ablex.

Sommerville, I. (2001). *Software engineering* (6th edition). Harlow, UK: Addison-Wesley.

Sonnentag, S. (1995). Excellent software professionals: Experience, work activities, and perceptions by peers. *Behaviour & Information Technology, 14*, 289–299.

Sonnentag, S. (1998). Expertise in professional software design: A process study. *Journal of Applied Psychology, 83*, 703–715.

Sonnentag, S. (2000). Expertise at work: Experience and excellent performance. In C. L. Cooper & I. T. Robertson (Eds.), *International Review of Industrial and Organizational Psychology* (pp. 223–264). Chichester: Wiley.

Sonnentag, S. (2001a). High performance and meeting participation. An observational study in software design teams. *Group Dynamics: Theory, Research and Practice, 5*, 3–18.

Sonnentag, S. (2001b). Using and gaining experience in professional software development. In E. Salas & G. Klein (Eds.), *Linking expertise and naturalistic decision making* (pp. 275–286). Mahwah, NJ: Erlbaum.

Sonnentag, S., & Kleine, B. M. (2000). Deliberate practice at work: A study with insurance agents. *Journal of Occupational and Organizational Psychology, 73*, 87–102.

Sonnentag, S., & Lange, I. (2002). The relationship between high performance and knowledge

about how to master cooperation situations. *Applied Cognitive Psychology, 16*, 491–508.

Stajkovic, A. D., & Luthans, F. (1998). Self-efficacy and work-related performance: A meta-analysis. *Psychological Bulletin, 124*, 240–261.

Sternberg, R. J. (1996). Costs of expertise. In K. A. Ericsson (Ed.), *The road to excellence: The acquisition of expert performance in the arts and sciences, sports and games* (pp. 347–354). Mahwah, NJ: Erlbaum.

Stevens, M. J., & Campion, M. A. (1999). Staffing work teams: Development and validation of a selection test for teamwork settings. *Journal of Management, 25*, 207–228.

Teasley, B. E., Leventhal, L. M., Mynatt, C. R., & Rohlman, D. S. (1994). Why software testing is sometimes ineffective: Two applied studies of positive test strategy. *Journal of Applied Psychology, 79*, 142–155.

Turley, R. T., & Bieman, J. M. (1995). Competencies of exceptional and nonexeptional software engineers. *Journal of Systems and Software, 28*, 19–38.

Vessey, I. (1986). Expertise in debugging computer programs: An analysis of the content of verbal protocols. *IEEE Transactions on Systems, Man, and Cybernetics, 16*, 621–637.

Vitalari, N. P., & Dickson, G. W. (1983). Problem solving for effective systems analysis: An experimental exploration. *Communications of the ACM, 26*, 948–956.

Voss, J. F., Fincher-Kiefer, R. H., Greene, T. R., & Post, T. A. (1986). Individual differences in performance: The contrastive approach to knowledge. In R. J. Sternberg (Ed.), *Advances in the psychology of human intelligence* (Vol. 3, pp. 297–334). Hillsdale, NJ: Erlbaum.

Weinberg, G. M., & Schulman, E. L. (1974). Goals and performance in computer programming. *Human Factors, 16*, 70–77.

Weiser, M., & Shertz, J. (1983). Programming problem representation in novice and expert programmers. *International Journal of Man-Machine Studies, 19*, 391–398.

Widowski, D., & Eyferth, K. (1985). Comprehending and recalling computer programs of different structural and semantic complexity by experts and novices. In H.-P. Willumeit (Ed.), *Human decision making and manual control* (pp. 267–275). Amsterdam: Elsevier.

Wiedenbeck, S., Fix, V., & Scholtz, J. (1993). Characteristics of the mental representations of novice and expert programmers: An empirical study. *International Journal of Human-Computer Studies, 39*, 793–812.

Ye, N., & Salvendy, G. (1994). Quantitative and qualitative differences between experts and novices in chunking computer software knowledge. *International Journal of Human-Computer Interaction, 6*, 105–118.