ISR Institute for Software Research
University of California, Irvine

An Initial Study to Develop an Empirical Test for Software Engineering Expertise

Susan Elliott Sim
University of California, Irvine
ses@ics.uci.edu

Erin Morris
University of California, Irvine
emorris@uci.edu

Sukanya Ratanotayanon
University of California, Irvine
sratanot@ics.uci.edu

Oluwatosin Aiyelokun
University of California, Irvine
oaiyelok@ics.uci.edu

May 2006

ISR Technical Report # UCI-ISR-06-6

Institute for Software Research
ICS2 110
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

ISR Institute for Software Research
University of California, Irvine

Tool Support for Incorporating Trust Models into Decentralized Applications

Susan Elliott Sim
University of California, Irvine
ses@ics.uci.edu

Erin Morris
University of California, Irvine
emorris@uci.edu

Sukanya Ratanotayanon
University of California, Irvine
sratanot@ics.uci.edu

Oluwatosin Aiyelokun
University of California, Irvine
oaiyelok@ics.uci.edu

May 2006

ISR Technical Report # UCI-ISR-06-6

Institute for Software Research
ICS2 110
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

An Initial Study to Develop an Empirical Test for Software Engineering Expertise

Susan Elliott Sim, Sukanya Ratanotayanon, Oluwatosin Aiyelokun
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425 USA
{ses, sratanot, oaiyelok} @ics.uci.edu

Erin Morris
Dept. of Social Ecology
University of California, Irvine
Irvine, CA 92697-3425 USA
emorris@uci.edu

ISR Technical Report #UCI-ISR-06-6

May 2006

ABSTRACT

Expertise is the consistently superior performance on a set of tasks in some area of human activity. Software engineering expertise is difficult to define, and characterize empirically. In this paper, we present and evaluate three candidate criteria for assessing software engineering expertise. These three criteria are: experience; characteristics common to experts across fields; and software-specific proficiencies. We conducted an initial empirical study to evaluate these three definitions. In a laboratory experiment, we asked novice and expert subjects to complete a number of software engineering tasks on a web application. We found that all three criteria should be used to provide a full categorization of an individual's level of expertise. Experience is a useful first filter, but cannot be used as the only criterion. Domain-independent characteristics are most useful for assessing the quality of the criteria for labeling an individual as novice or expert. Software-specific proficiencies appear to be the most promising, but assessments for various skills, technologies, and problem domains will need to be developed separately.

General Terms

Measurement, Design, Economics, Experimentation, Human Factors.

Keywords

Expertise, empirical study, problem solving, novice, expert

An Initial Study to Develop an Empirical Test for Software Engineering Expertise

Susan Elliott Sim, Sukanya Ratanotayanon, Oluwatosin Aiyelokun
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425 USA
{ses, sratanot, oaiyelok} @ics.uci.edu

Erin Morris
Dept. of Social Ecology
University of California, Irvine
Irvine, CA 92697-3425 USA
emorris@uci.edu

ISR Technical Report #UCI-ISR-06-6

May 2006

ABSTRACT

Expertise is the consistently superior performance on a set of tasks in some area of human activity. Software engineering expertise is difficult to define, and characterize empirically. In this paper, we present and evaluate three candidate criteria for assessing software engineering expertise. These three criteria are: experience; characteristics common to experts across fields; and software-specific proficiencies. We conducted an initial empirical study to evaluate these three definitions. In a laboratory experiment, we asked novice and expert subjects to complete a number of software engineering tasks on a web application. We found that all three criteria should be used to provide a full categorization of an individual's level of expertise. Experience is a useful first filter, but cannot be used as the only criterion. Domain-independent characteristics are most useful for assessing the quality of the criteria for labeling an individual as novice or expert. Software-specific proficiencies appear to be the most promising, but assessments for various skills, technologies, and problem domains will need to be developed separately.

General Terms

Measurement, Design, Economics, Experimentation, Human Factors.

Keywords

Expertise, empirical study, problem solving, novice, expert

1. INTRODUCTION

> "*Candidates must have 3+ years software development experience on both UNIX and Windows. Strong technical skills covering the design and implementation of cross platform distributed systems are a must. Strong Java/ J2EE/ JSP development experience is required. Strong verbal and written communication skills are required. Knowledge of security models, JSP, JSF, JBoss, Weblogic, WebSphere and relational databases is desired BS in Computer Science or Computer Engineering or the equivalent experience is desired*". – Excerpt from job advertisement 24639 for a senior software engineer on monster.com

In industry, companies are constantly searching for highly qualified personnel because they are critical to the success of a software project. It is a challenge to find someone with the right experience, skills, and background who is also compatible with the existing team. On routine projects, a proficient software engineer helps things go smoothly. On innovative projects, an expert can be the difference between success and failure. As F. P. Brooks pointed out, "The differences are not minor…the very best designers produce structures that are faster, smaller, simpler, cleaner, and produced with less effort" [6].

We can learn a great deal by studying expert software engineers. Through them, we can access many years of experience, thereby improving our understanding of software engineering itself. Being able to characterize, and even quantify, expertise is beneficial for human resource management, software engineering education, and even empirical evaluation of software tools and methods. This knowledge can be used in human resources to more effectively recruit and select team members. By identifying differences between novices and experts, we can teach students strategies and approaches so they can solve problems in the same manner as experts. Finally, it can be used empirical evaluations of tools and methods as a base for developing metrics to measure performance of software engineers. In other words, a software technology can be considered helpful if it assists a user to work more proficiently, that is, to behave more like an expert software engineer.

In research, expert software engineers are equally hard to find. Setting aside the practical problem of enticing them to participate in an empirical study, a fundamental problem is defining software engineering expertise. While we can consult software engineers for evaluations of their peers, we also need independent confirmation through an empirical test.

In this paper, we report on initial work to establish a definition and empirical test for software engineering expertise. Beginning with a review of the literature on expertise in program comprehension and psychology, we identified three candidate definitions of expertise: years of work experience; cognitive characteristics of experts; and software-specific proficiencies. Subsequently, we conducted a pilot study to assess the three definitions.

We found that all three criteria are necessary to provide a full characterization of an individual's expertise. The number of years of experience provides an easy first test, but there is a great deal of variability among individuals who have been working the same amount of time. Cognitive criteria are most useful for evaluating the a definition of expertise, because they provided us with a list of behaviors to expect from experts. The software-specific proficiencies provide the clearest characterization of a software engineer's expertise. However, the drawback of this

criterion is that specific tests must be developed for each skill, technology, or problem domain of interest.

This paper is organized as follows. Section 2 reviews different definitions of expertise used in the literature. Our three candidate criteria for expertise and their associated empirical tests are presented in Section 3. In Section 4, we describe the pilot study that we conducted to evaluate the candidate criteria. We present and discuss our findings in Section 5. We conclude the paper by with a preview of Future Work in Section 6 and a Summary in Section 7.

2. PRIOR RESEARCH

There have been many studies of novice or expert programmers, but relatively few of these examine the differences between them on software development tasks. The most well-known of these focus on only programming and have produced results on mental model of programs and program comprehension strategies. Findings from these studies pertain to how programs are remembered, characteristics of mental models, or how schemas are acquired. For instance, experts tend to remember the semantics (or abstract representation) of a program, while novices tend to remember functions or syntax (or concrete representation) [2, 31].

Studies of programming expertise share a number of methodological characteristics, such as the size of programs used, the tasks used in the experiment, and selection of subjects [2, 11, 31-33]. The programs used in the experiments had about 100 lines of code (LOC), and sometimes as few as 10-12 LOC [33]. Subjects were asked to study the programs for a few minutes and then asked about what they remembered. Finally, the novices and experts in the experiments were specifically focused on programming. Novices were undergraduate students who had recently completed an introductory programming course. Experts were more senior students (sophomores or graduates) or teachers of an introductory course. The criteria for expertise used in these studies are given in Table 1.

In contrast, our research seeks to study software engineering expertise as an activity encompassing more than programming and requiring many years to become an expert. More specifically, we are interested in problem solving on a large scale, like the kind required on large

Table 1: Criteria Used in Studies of Programming Expertise

Study	Novices	Experts
Adelson, 1984 [2]	Undergraduates with an introductory course	Teaching fellows from same introductory course
Davies, 1994 [11]	Undergraduates with one course in Pascal	Teachers of Pascal or professional programmers
Widenbeck, 1991 [31]	Sophomores with an average of 3.4 programming courses	Graduate students and working programmers with average of 10.5 courses in programming
Wiedenbeck, Fix, and Scholtz, 1993 [32]	Undergraduates with an introductory course in Pascal	Professional programmers with 2-13 years experience (avg. 7)
Wong, Cheung, and Chen, 1998 [33]	Undergraduates with one course in BASIC	Students with two years experience and one course in BASIC

software projects. Our goal is to acquire a better understanding of software engineering by studying expert practitioners and their work.

There is a smaller body of work that approaches software engineering expertise in the manner that we propose. These involve a wider range of software engineering activities, involve subjects with more experience, and include more qualitative methods. The activities studied include specification [29], design [3, 4], documentation [26], and maintenance [5, 19]. The experts in these studies have 5, 10, and even 30 years of industry experience, often on large projects. While some of these studies were laboratory experiments [3, 19, 26, 29], others were field studies that followed software engineers over time [4, 5, 8], using interviews or recordings of their

Table 2: Criteria Used in Studies of Software Engineering Expertise

Study	Novices	N=	Experts	N=
Adelson and Soloway, 1988 [3]			At least 8 years commercial experience	3
Ahmed and Wallace, 2003 [4]	Trainees with a few months of experience (most less than one year)	6	Authorities with decades of experience (at least 10 years and some with more than 30 years)	11
Berlin, 1993 [5]	Graduate students or professional researchers	3	Researchers with approximately 10 years of programming experience	3
Campbell, Brown, and DiBello, 1992 [8]			At least 5 years of professional experience and considered to be experts by peers	7
Guindon, 1990 [16]			Had advanced degrees and many years of experience Considered by peers and managers to be very competent Exhibited one of three styles observed in initial set of eight designers	3
Koenemann and Robertson, 1991 [19]			>= 3 programming languages 4-15 years of programming experience worked with large programs regularly formal education in CS or EE	12
Letovsky, 1986 [21]	<3 years of professional experience	2	3-20 years of professional experience	4
Mastaglio and Rieman, 1993 [24]			Professional research associates and doctoral students	8
Riecken, Koenemann-Belliveau, and Robertson, 1991 [26]			≥ 10 years experience programming ≥ 6 years experience with C Active in software engineering, systems research and development	16
Sutcliffe and Maiden, 1992 [29]	<= 6 months experience with structured analysis	13		

interactions. The criteria for expertise used in these studies are given in Table 2.

3. WHO IS AN EXPERT?

An expert is someone who consistently performs at a high level in a specific field of human activity [30]. This skill is often accompanied by a track record of accomplishments and consequently the achievements cannot be attributed to luck.

While the definition is easy enough to understand, it's more difficult to operationalize, that is, to devise a test or measure to separate novices from experts. We found three different bases for identifying experts: 1) amount of work experience; 2) cognitive characteristics of experts; and 3) software-specific proficiency.

3.1 Expertise = Time?

Job advertisements, such as the one at the start of this paper, typically use number of years of experience as an indicator of expertise. Using time as an indicator is also done in cognitive psychology, which tends to view expertise as a dichotomy and a frequently-used test is the amount of time spent perfecting the craft. Figures of merit include 10 000-20 000 hours of practice and preparation (chess and physics) [20, 28] and 10+ years of training or education (sports, science, medicine) [12]. The technical term for this preparation time is deliberate practice.

The strongest proponents of using deliberate practice to define expertise are Ericsson and Charness [12]. They argue that practice is the sole factor in achieving expertise and that talent, or innate ability, does not play a role in the level of performance achieved. Innate characteristics affect only an individual's ability to organize their time to put in the requisite hours of practice.

Translating deliberate practice to software engineering is less than straightforward. Practice is clear in endeavors where there is an obvious performance. In music, the performance is the concert and practice is scales, rehearsals, lessons, etc. In sports, the track event is the performance, and practice is training, conditioning, drills, etc. In software engineering, we don't have practice sessions where developers refine basic skills, receive guidance from a teacher, or rehearse part of their job. In a sense, software engineers are performing all the time. They don't really have the opportunity to receive feedback, nor do they undertake activities specifically to hone their skill.

If we set aside this subtlety and permit the substitution of performance for practice, how many years of work experience is required to attain 10 000 hours of deliberate practice? A lower limit is five years (40 hours per week x 50 weeks per year). However, this calculation excludes the overhead activities involved in working. Consequently, it may 10 or more years to become an expert software engineer. Still, the five-year breakpoint is useful because job advertisements for senior software engineers typically request a minimum of five years of work experience.

3.2 Cognitive Characteristics of Experts

Another way of defining an expert is by identifying characteristics that they have in common when performing, working, or problem solving. Glaser and Chi [14] and Tan [30] surveyed the psychology literature and identified seven domain-independent characteristics of experts. These

characteristics can serve as a prescriptive behavioral definition, that is, individuals exhibiting these characteristics should be classified as experts.

i) Experts have an extensive knowledge base associated with a specific domain.
Estimates of the size of this knowledge base range from 50 000 chess patterns to 20 000 "things" (concepts, equations, and techniques) in kinematics [20]. This knowledge base is acquired over the years of deliberate practice. This knowledge base is strongly tied to a particular domain, so it is difficult to transfer the knowledge from one domain to another [8, 25]. Indeed, Curtis, Krasner, and Iscoe [10] observed that this was one of the causes of breakdowns when designing large software systems.

ii) Experts are fast, much faster than novices, and they quickly solve problems with little error.
During hours of practice, experts have had many opportunities to rehearse and repeat behavior until they are subconscious, automatic routines. Examples of this automaticity are an instinctive knowledge of how to use software tools [5] and translation of telegraphic identifiers in source code to meaningful keywords [19]. In addition to this speed with basic skills, experts explore fewer options overall, and are adept at avoiding dead ends [5, 15].

iii) Experts perceive large meaningful patterns in their domain.
Details that are missed by novices are frequently noticed by experts, and this information often serves as a trigger to plans of action or large networks of knowledge [15, 19, 31]. For instance, seeing a SELECT statement inside C++ code immediately tells the expert that ESQL (Embedded Structured Query Language) is being used to interface with a database. This observation leads to an activation of the plans associated with using ESQL, expectations regarding beacons or landmarks in the source code, strategies for debugging, etc.

iv) Experts have superior recall because they organize their short-term and long-term memory more efficiently.
While an expert may appear to have a larger memory, this superior performance is more accurately attributed to chunking and hierarchical organization. Chunking is the process of grouping together meaning information into a single unit, so that it takes up less "space" in memory. These chunks are organized hierarchically that the essential features of a problem can be used to recall the information precisely and efficiently. This aspect of expert performance has been studied more extensively than the others, and programming is no exception [11, 22, 31, 32].

v) Experts see and represent a problem at a deeper (more principled or abstract) level than novices; novices tend to represent a problem at a superficial level.
In other words, novices tend to get bogged down in the surface details of a problem (e.g. programming language syntax), whereas an expert understands the underlying structure of the problem (e.g. semantics or design pattern). Experts' representations of problems rely more on principles and metaphors rather than literal features [3, 27].

vi) Experts spend a great deal of time analyzing a problem qualitatively.
When faced with a problem, an expert is initially concerned with defining the problem. In both physics and engineering design, the first step is create a sketch of the problem which is used to create a conceptual model [9, 20]. Only when the problem has been scoped, does the expert begin searching for solutions. From there, understanding of the problem and creation of the solution proceed together [3, 17, 29].

vii) Experts have strong self-monitoring skills.

They are more aware of when they make errors, why these errors occur, and when a solution needs to be verified. They can also more accurately predict which problems will be difficult. Results of studies suggest that this self-awareness comes from a deep understanding of the problem to be solved and experience. Ericsson and Charness, on the other hand, argue that a degree of self-awareness is necessary to create experts, so that time spent in deliberate practice can be used effectively [12]. In programming, this comes into play when experts are giving explanations [5] or adding comments to code [26].

These seven characteristics set experts apart from novices in domains with close-ended problems. However, there are a few occasions or problem types where novices surpass experts. One example is problems that are ill-formed or involve uncertainty. These problems do not necessarily have a single correct answer and there is either insufficient information or too much information to constrain the solution. Expert judgment has been studied in domains such as selecting medical students and predicting securities prices, and it was found that novices made better choices than experts and used more information in their decisions [18]. In engineering design, it was found that experts generated fewer options or hypotheses and were more reluctant to abandon them in face of difficulty than novices [9].

These cognitive characteristics can be used as a behavioral definition of expertise. Anyone displaying them can be classified as an expert. This is particularly useful in domains, such as ours, where other objective criteria are yet to be established and ground truth is not yet known. In our study, we will be using these cognitive characteristics to test our other definitions.

3.3 Software-Specific Proficiencies

A third way of defining a software engineering expert is in terms of what such a person ought to be able to do. In other words, what kind of skills should he or she have and to what degree? What kinds of problems should he or she be able to solve? One implication of this type of definition is that it treats expertise as a continuum, that is, as a series of milestones that mark different levels of expertise. This approach is typical of developmental psychology, where children and adults are studied to characterize changes over time [8]. In developmental sequences, it is not possible to skip stages, although it is possible to enter the next stage prior to completing the previous one. There are two types of developmental sequences: issue-based and proficiency-based. Issue-based developmental sequences characterize expertise in terms of problems that learners have at a particular stage. For example, understanding the Model-View-Controller architectural style and how to use the style is a milestone in learning object-oriented programming. A proficiency-based developmental sequence is defined in terms of skill or understanding. An example of a skill is building a graphical user interface using a toolkit. A novice would follow examples or recipes for building windows, while an expert would be able to build novel interfaces quickly and easily.

This developmental definition of expertise is much more nuanced than the first two. Expertise is defined in terms of a domain, and a person can be an expert in one domain, but a novice in another. The definition is attractive here, because software engineering is such a complex activity. There are a number of ways of partitioning the problem space into domains. The Software Engineering Body of Knowledge (SWEBOK) lists ten knowledge areas ranging from requirements to testing to tools and methods [1]. Each of these knowledge areas can serve as a

sub-domain. Campbell, Brown, and DiBello suggest a slightly different partitioning: programming knowledge (e.g. languages, data structures), technology (e.g. tools, frameworks, libraries), and application domain (e.g. banking, aerospace) [8]. While this characterization identifies three orthogonal areas, it seems incomplete. For instance, there is not an obvious place for software engineering conceptual knowledge nor skills for team work.

This definition of expertise can be operationalized as the ability to solve a task. This can be a single complex task designed that requires a range of skills. Or it can be a sequence of small tasks at varying levels of difficulty or requiring different amounts of domain knowledge. In our experiment, we selected a task that is solvable by experts, but too difficult for novices. It requires knowledge of a number of different software technologies, the understanding of web application architecture, the ability to use software tools, and communication and documentation skills. This is not a small list of requirements, but it is typical of modest software engineering problems. Consequently, it will serve adequately as an empirical test for a proficiency-based definition of expertise.

4. EMPIRICAL STUDY

We conducted a pilot study to perform an initial evaluation of the three criteria for expertise. We used number of years of experience to initially categorize a subject as novice or expert. We subsequently analyzed their behavior using the domain-independent characteristics. Finally, their performance was evaluated in terms of software-specific knowledge and skills.

In our experiment, we used a software engineering task with three components: information seeking when comprehending software, communication skills and strategies, and code change. We asked subjects to add a new feature to an existing web-based survey management application. Two subjects, A and B, are required for each session. The first subject studied a web application and completed a form describing how to make a prescribed change to the application. Subject A then passed on this information to the second subject. Finally, Subject B implemented the changes. In total, there were six sessions, consisting of the different pairings of experts and novices. Two out of the six pairs were asked to think aloud to provide us with additional insight into their behavior [13].

This division of tasks not only allows us to examine a number of software engineering skills, it also mimics situations where research is separated from detailed work. Examples of such situations are outsourcing a set of maintenance tasks to another organization, a senior developer assigning a task to a software immigrant, and preparing a Change Request Proposal (CRP) for an architectural Change Control Board (CCB). The CRP provides the CCB a better understanding of the proposed change, its importance, and its impact.

4.1 Procedure

Each session comprised three tasks: Task A, Handover and Task B and took a total of 210 minutes. The time line of a session is depicted in Figure 1. The first subject was required to complete Task A and the Handover. The second subject was required to complete the Handover and Task B. During each session, a subject's activities were recorded by a web camera, a microphone, and screen capture software. Scratch paper was provided to the participants and was

collected at the end of the study. Eclipse IDE containing a project set up for the task and TextPad software were also prepared for subjects.

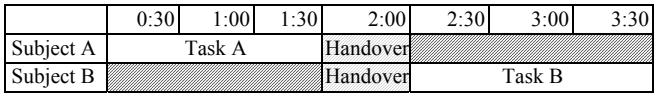

Figure 1: Time Line of Study Procedure

Task A. This task was performed by Subject A individually. The subject was given a scenario where a customer requested a feature in the company's survey management application. He or she was asked to complete a CRP describing how to make the change. This CRP will be described further in the next subsection. The subject was also asked to not make any modification to the application. The subject was given up to 90 minutes to complete this task.

Handover. In this task, Subject A verbally and physically handed over to Subject B the information gathered in the first task. Once the Task A finished, Subject B was introduced to Subject A for an explanation of the application to be changed and how to make the changes. The Handover task began with Subject A giving an explanation without any interruptions. Once the explanation was completed, Subject B was allowed to ask questions. The subjects were given 30 minutes to perform this task.

Task B. Following the Handover, Subject B was left with the CRP and Subject A's notes. Subject B had to work individually and make the modification in 90 minutes.

For all three tasks, subjects were allowed to use any information and resources available on the Internet. They were given documentation about the application and task description, but there were no application developer notes.

Some accommodations were provided to subjects in case they were unfamiliar with web application development and the available tools. Subjects were also given a short description of typical architecture of web applications, and instructions on running and compiling the application. In addition, each subject was asked every 30 minutes whether he or she had any problems performing the task.

4.2 Subjects

A total of twelve subjects (six novices and six experts) participated in the study. In this study, we defined novices as senior undergraduates or recent graduates who had been working for less than one year as software engineers. Experts were expected to have at least five years of work experience as software engineers.

The novices had an average age of 24.3 (s.d.=4.6) and the experts 34.2 (s.d.=3.6). We had 3 female subjects and all of them were experts. All subjects considered themselves fluent in English.

All the novices had majored or were majoring in computer science at the time of the study. Two of the experts had undergraduate degrees in computer science and the remainder had Master's degrees (two in computer science, one in electrical engineering, and one in economics). Both novices and experts knew about the same number of programming languages (5.5 and 5.8 respectively), although the experts had more difficulty enumerating them, suggesting a ceiling effect in terms of recall. Subjects were asked using multiple choice questions about the amount of experience they had in six different domains. All of the experts, except for the one with a

background in electrical engineering, indicated at least three to five years of experience in at least one domain. Some reported similar amounts of experience in as many as three domains and up to 10 years of experience was reported by others. In contrast, none of the novices reported more than 1-3 years of experience in any domain.

Six sessions were conducted with different pairings: expert-novice, expert-expert, novice-expert, and novice-novice, pairs. The novice-expert and expert-novice pairs were conducted twice. The diversity in the pairings afforded a means of investigating the interactions between subjects with different, or same, levels of expertise.

		Task A	
		Novice	Expert
Task B	Novice	N6-N5	E1-N1 E2-N2*
	Expert	N3-E3 N4-E4*	E7-E8

* indicates think-aloud was used

Figure 2: Novice-Expert Subject Pairings by Task

4.3 Software and Change Request

The application used in this study is an open-source web-based survey management tool called VTSurvey, developed at Virginia Tech. It is a typical three-tier web application, created with JSP, Java and XML technologies, and running on a Tomcat application server. It enables an application administrator to create, maintain and run online surveys.

VTSurvey consists of 38 Java™ files and 74 JSP (Java Server Page) files. Total line of code of the application is 10 342 lines. In addition, there are 4 DTD (Document Type Definition) files. The application is well-structured and the source code is well-formatted, but sparsely commented.

Originally, VTSurvey did not maintain each user's email address. Subjects were asked to modify the system to maintain this information. Our requirements were: i) the system shall allow the administrator to provide an email address upon adding a new user; ii) the system shall store the email address; and iii) the system shall present the user's email address in a column next to the user account on the user account listing. The change involves only one Java file, two JSP files, and one DTD file. In total, there are 393 lines in these four files.

In Task A, the subject was asked to fill the CRP form. Completing this document allowed the subject to demonstrate his or her understanding of the system. The information required by the CRP form are: i) change type and description, ii) system description, iii) modules and files affected by the change, iv) risks, v) effort estimation, and vi) test cases.

4.4 Questionnaires and Debriefing

In order to gain a better understanding of the subject's personal characteristics, we administered four questionnaires: i) background and work experience, ii) computer science background knowledge, iii) personality, and iv) general problem solving ability.

The background questionnaire asked subjects about their educational background, employment history and level of experience with some software technologies. The second questionnaire tested a subject's knowledge of computer science concepts and included topics such as process models, programming languages, and algorithms.

A debriefing session was conducted with each subject in order to elicit his or her reflections on the work done. They were asked about accuracy and completeness, possible mistakes in the CRP form or implementation, and their level of confidence in their work. Subjects B also had the opportunity to evaluate the correctness of the information received during the Handover task and in the CRP form.

5. RESULTS AND DISCUSSION

5.1 Years of Experience

As an initial categorization, we defined an expert as someone who had five years or more of industry experience as a software engineer. A novice was a senior undergraduate student or recent graduate with less than one year of work experience. Using this criteria, we did not see consistent differences between the two groups in terms of performance on tasks or behavior.

Table 3 shows how novices and experts scored on the two tasks. These scores were generated by having two independent raters score the completed CRP form and the implementation. In almost all cases, the raters agreed with each other and when they did not they were within one point. While the top scores for both tasks were earned by experts, there were not consistent differences between the two groups.

Table 3: Novice vs. Expert Performance on Tasks

	Novice	Expert
Task A- CRP Form (out of 33)	19 24 28	19.5 29 33
Task B- Code Change (out of 55)	12 35 50	35 44 55

The data indicates that the two groups have overlapping performance profiles, meaning that our thresholds did not produce two distinct groups. Five years of experience may not be sufficient to become an expert. In comparison, we had two subjects who had 10 years of experience with web applications (E4 and a test subject) and both their qualitative and quantitative results were more in line with our expectations. These subjects were prominent in our discussions and analysis as we worked with the data. We will take these findings into consideration when we define cut-offs in future studies.

5.2 Cognitive Characteristics

We conducted further qualitative analyses to determine whether we could identify any consistent behavioral differences between experts and novices. Using the characteristics described in Section 3.2, we looked for examples of expert behavior.

5.2.1 Information Seeking

While subjects perform information seeking in both Task A and B, we focused on Task A in this analysis, because the strategies used are clearer. For VTSurvey, subjects needed to seek the following information to plan the modification and complete the CRP form.

Relevant Components. It is not possible for a subject to study the whole system in the limited time available. Subjects had to selectively identify the relevant module, in this case the user management module, and focus their efforts here.

Component Interaction. Subjects had to understand the interaction among components because functionality, such as adding or listing a user, is achieved by interaction of various components in different architectural layers in web application.

Data Structure and Data Storage. VTSurvey stores data as XML files on the server's file system in a specific pattern and uses a SAX (Simple API for XML) library to manage the data instead of using a relational database management system. Subjects gather this knowledge while studying the system.

Patterns in User Information Management. Our change requirement asked that an email address be maintained with user data. If the subject understood how this information is stored and retrieved, they can use this knowledge to make the required modification.

Since experts have more background knowledge and see problems in a more principled way than novices, we would expect them to use a more orderly strategy to gather information needed to complete the task, thus leading to faster performance and fewer errors (by characteristics i, ii, and v in Section 3.2).

5.2.1.1 Strategies Used by Experts

We found that two out of our three experts employed approaches that resembled the top-down and hypothesis-driven approach proposed by R. Brooks [7]. In addition, the strategy they used to study a component and to transition from one to another was systematic and orderly. The performance of these two subjects was clearly superior to other subjects. This finding is consistent with the literature [23].

Experts normally have a larger knowledge base about related technology and software engineering practices, e.g. XML technology, web application architecture, modularization of large systems. This knowledge can aid them in forming and validating their hypotheses about the VTSurvey application when using top-down and systematic strategy. For an example, knowledge about general web application architecture helps experts to perceive the interaction among components in different layers which helps them in understanding delocalized plans in the design.

In order to illustrate the top-down and systematic approach employed by experts, we present a detailed analysis of the information seeking behavior by a representative expert, E1. The diagram shows the files viewed in her session is presented in Figure 3. The vertical axis shows the different levels of the architecture and the horizontal axis shows the amount of time spent on a file or activity.

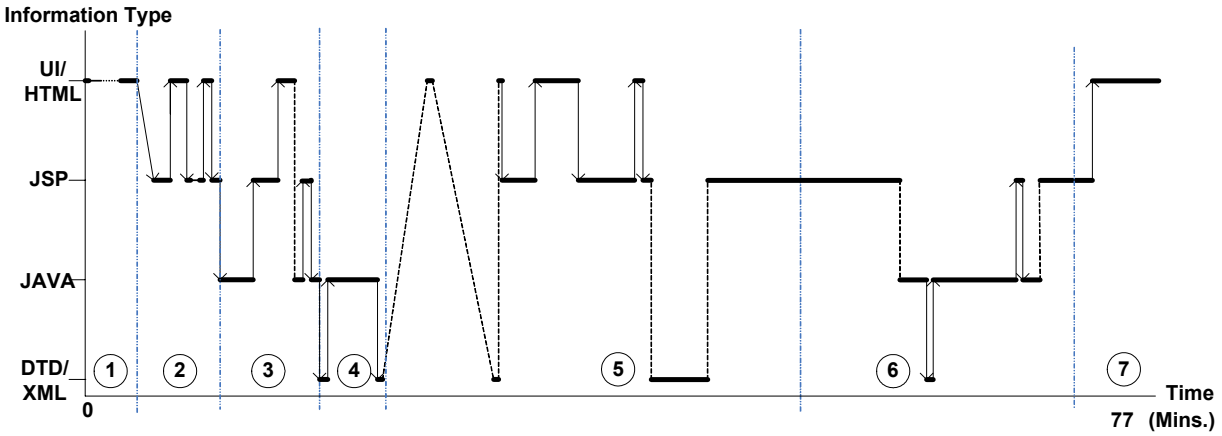

Figure 3: Sequence of Files Viewed by Expert 1

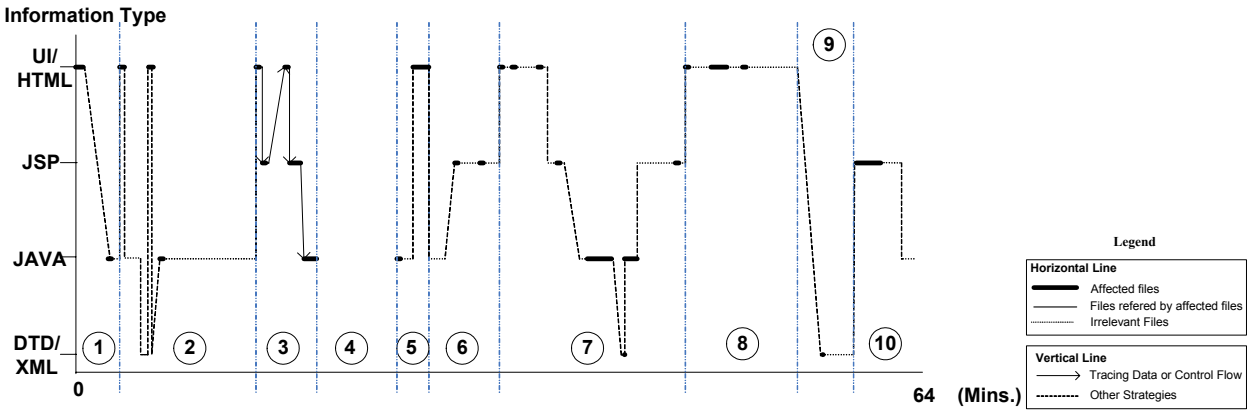

Figure 4: Sequence of Files Viewed by Novice 3

The goals for E1's activities during various phases were clear and easy to identify. In Phase 1 to Phase 4, as labeled in Figure 3, E1 studied the application to plan the change. In Phase 6, she was planning the part of the modification relating to retrieving and displaying the email information. Phases 5 and 7 were spent filling out the CRP form, and the files viewed in those phases were used to refresh her knowledge obtained from previous phases. From Figure 3, we can see that E1's approach shows the following characteristics:

Top-Down. E1 studied the system starting from information in a higher level of abstraction to information in lower level of abstraction as seen in Phase 1 to Phase 4. In Phase 1, she tried to obtain understanding about the behavior of the system by interacting with the running system. She formed hypotheses about the behavior that she saw and validated these by studying the JSP, Java, DTD, and XML files.

Incremental. We found that E1 didn't study all system behaviors in Phase 1. She only looked at simple behaviors and made simple hypotheses, for example, about the UI components. Once simpler hypotheses were validated, more complicated system behaviors were studied and hypotheses about those behaviors were made. For examples, in Phase 2, E1 went from JSP level to UI/HTML level to explore the case in which incomplete user information was supplied when adding a new user, and in Phase 3, the case in which a user is successfully added to the system was explored.

Systematic. E1 employed an orderly systematic strategy to comprehend and validate her assumptions about the system. The transitions from one file to another were the result of following the control flow and data flow. For example, the transition from the UI layer to the JSP layer was made by tracing how the system passed data; and transition from JSP to Java class was made by following a method call from a JSP file. She was also able to map source code behavior identified by symbolic execution to system behavior. This was reflected in her Handover session when she described knowledge obtained in Phase 6, saying "This JSP basically is looking into the directory where XML files are stored and just display the name of all the files which is also the user ID."

The other expert, E2, employed an approach based on keyword searching of the filenames and made assumptions about their behavior based on their names. However, this subject spent less than ten minutes comprehending the system and demonstrated little understanding of how the system worked. Both of these actions led us to question the categorization of this subject as an expert and supported the use of cognitive characteristics to classify subjects.

5.2.1.2 Strategies Used by Novices

Unlike the experts, the novices used a strategy based primarily on keyword searching and didn't show the characteristic of top-down approach. Even the novice with the best performance scores, N6, did not employ the same strategies as experts. However, he was able to focus on relevant files more than other novices and showed a systematic approach to studying files at a given architectural layer, such as JSP, or in Java object, but not when moving from one layer to another. Since he based his searches of the source code on keywords and identifiers, he incorrectly selected a file because its name, `showUser.jsp`, suggested that it contained the functionality that needed to be modified.

Detailed analysis of subject N3, is presented in Figure 4 as representative of novice's approaches. Sequences of files viewed in his session is grouped into phases and labeled in the same manner as Figure 3. We found that N3's approach has the following characteristics.

Based on Keyword Searching. N3 relied heavily on keyword searching to seek out information about the system. For example, N3 looked for files that have names containing a keyword and restricted his investigation to those files. Examples of keywords used are "user," "email," or the name of User Interface controls like "button" or "text area." This behavior was displayed in Phases 1, 5, and 6, among others. N3 also displayed usage of keyword searching to help in validating his hypothesis. He assumed that one of the Java objects was responsible for creating the UI needed to be modified and validated this assumption by trying to search for a particular caption from the UI in every Java class.

Not Orderly. Unlike E1 who incrementally formed and validated her hypotheses and made a transition from one component to another in a systematic way, N3 used a less orderly process. This might be due to the fact that N3 based his approach on keyword searching, but lacked a clear goal or criterion for selecting a keyword. N3 tended to jump from one assumption to another as he studied opened files based on keyword searching. In addition, he didn't seem to realize that most of the components he studied were not relevant to the task. This resulted in little time for focusing on understanding the relevant components, and made it more difficult to grasp the interaction among components.

Easily Derailed. N3 was less skillful in performing symbolic execution than E1. This difference may be due to differences in their knowledge base and their ability to perceive interactions among components. N3's approach was easily derailed by the unexpected. Examples of this can be seen in Phases 3, 5, and 6. In Phase 3, N3 displayed top-down and systematic strategy by following control flow from `adminAddNewUser.jsp` to a Java User object. In addition, he also closed irrelevant files opened in previous phases which showed that he identified that they are not related to the task. However, in Phase 5, he found the usage of an unfamiliar SAX object in the User class and was derailed. Subsequently, N3 spent his time performing more keyword searching and studying irrelevant files.

It is interesting to note that in his debriefing session, N3's description of his approach resembled a top-down approach. It is possible that the subject was trying to perform top-down and systematic approach as was the case in his Phase 3, but lacked the skill to follow through with this intention.

5.2.2 Communication

Experts see and represent a problem in a more principled manner in comparison to the novices, because of their ability to perceive large meaningful patterns in their domain as well as their ability to represent a problem at a more abstract level (characteristics iii and v in Section 3.2). In addition, they spend more time understanding the problem qualitatively (characteristics vi). Consequently, we expect explanations by experts to be better organized and to be broader in scope, i.e. include information about the boundaries of the problem.

To gain insight into differences in how novices and experts communicate about modifying a system, we studied the Handover session. This task consisted of the uninterrupted explanation given by Subject A and the questions posed by Subject B. The information presented by Subject A reflects his or her judgment of what constitutes useful information for task completion. The questions posed by Subject B also reflected his or her evaluation of the explanation.

Analysis of the handover session was based on audio and video records of the interaction and transcripts of the conversation. The transcripts were analyzed inductively and a coding scheme was developed. Utterances by both subjects were placed into one of seven categories, see Table 4. All the transcripts were categorized by two coders; an inter-rater reliability of score 0.88 was calculated on sample transcripts. All the disagreements were between the categories of Setup and Change Instructions, and these were easily resolved by consensus.

Our coding scheme maps onto three levels of abstraction: problem domain, intermediate level, and a program or implementation domain. The problem domain level includes four categories, Restatement of Change Request, Setup, Usage, and Scoping. The program or implementation domain corresponds to the Change Instructions and Testing categories. The intermediate layer maps the high-level specifications in the problem domain to low-level implementation details in the program domain. The Mechanics category falls into this layer of abstraction.

Table 4: Mapping Coding Categories to Level of Abstraction

		Coding Category
Level of Abstraction	**Problem**	**Setup.** Information about file locations, compiling and running the program. For example, "…the DTD files are in this folder." **Restatement of Change Request**. Information that was given in initial change request fell into this category. For example, "…they will also like to add the user's email address." **Usage**. This category was for information related to the running system, as seen by users. For example, "…this is where the administrator does his work or her work." **Scoping.** Some subjects had concerns that were not explicitly addressed in the change request, such rationale for the change error checking. These concerns were raised when attempting to understand the bounds of the problem. For example, "Do I have to verify that it's [email address field] blank?"
	Intermediate	**Mechanics**. General information about how the underlying mechanisms of specific technologies or system subcomponents work. For example, "The DTD files are kind of like a descriptor of how an XML should look like."
	Program	**Change Instruction.** Specification information on how to implement the requested change. For example, *"*Then all you need is a get/set method inside of `user.java` to talk between the `user.dtd` and the two JSP files which are user interface." **Testing.** This category included information relating to test cases and how to validate the changes made. For example, "…log-in with this fake user to see whether you're able to send in the survey, to make sure we didn't break anything."

A principled presentation should provide a good mapping between the problem and program domains without being bogged down with the program details. At the same time, it should span and map between these levels of abstraction. Such an explanation is both an indication of the subject's understanding of the program and a requirement for a successful implementation of the change.

Figure 5 shows the percentage of the explanation spent at each of the abstraction levels by each of the subjects. The novices were subjects N3, N4, and N6 and the experts were E2, E1, and E7. (Subject N6 did not spend any time on the program level.) Only three of the explainers spent more than 20% of the time in their explanations on Mechanics, the intermediate level that bridges the problem and program domains. In the graph, the intermediate level is depicted by a solid black bar. The remaining subjects, mostly novices, spent less than 10% of their explanation mapping between these two layers. Subjects E1 and E6 spent 132s (28.5%) and 152s (21.2%) respectively on the technologies and subsystems involved in the project and grounding the specifications in program details. Although N3 spent more than 20% of his explanations on Mechanics, the absolute length was brief, only 15s, indicating it was a small utterance in a small explanation.

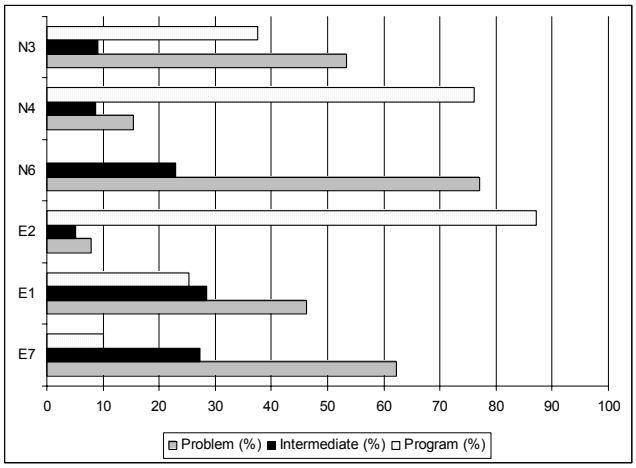

Figure 5: Percentage Time Spent at Abstraction Levels

Much of the information at the most abstract level, the problem domain, was presented to the subject in initial the documents. Specifically, they were given a document that described the program and another that contained notes on file locations, running the program, and compilation. Consequently, generally less time was spent conveying information at this level of abstraction. Scoping was the only category at this level of abstraction that required careful thought or analysis.

We found that experts provided explanations that had more scoping information, that is, gave more details about the context of the problem. Table 5 shows the amount of time spent on scoping by the subjects. Only one of the three novices spent any time explaining the context of the problem. In contrast, all of the experts spent some time on scoping; one spent nearly four minutes on the topic.

Table 5: Time (seconds) Spent on Scoping

Novice	Expert
0	9
0	9
11	227

Explanations provided by experts also tended to be grounded in the code base, that is, contained fewer errors. This knowledge of the low-level program details enabled subjects to advise the implementers about the relevant files and methods that needed to be changed. Subjects E1 and E2 spent less time on the program details compared to the problem domain and mechanics. In addition they are the only subjects that did not provide any incorrect information to their respective implementers (see Table 6).

Table 6: Incorrect Statements by Subjects

Novice Subjects	Number of Incorrect Statements	Expert Subjects	Number of Incorrect Statements
N3	1	E1	0
N4	2	E7	0
N6	2	E2	2

Although novices allocated as much time to the program details, they sometimes passed on incorrect information. This can be attributed to their inability to obtain a coherent view of the code base. Since the primary information seeking strategy used by the novices was keyword searching, they were misled at times by the identifiers. In addition, they rarely looked at the code to verify their assumptions once a candidate file was found.

Similarly to information seeking task, Subject E2's performance deviates from that of other experts as she provided incorrect information during the Handover session. For example, she made reference to Java Database Connection (JDBC), despite the fact that JDBC nor a relational database were used in VTSurvey. Some factors that may be responsible for this include: 1) Unlike all the other subjects, Subject E2 worked on the CRP form before looking into the system to justify her hypotheses; and 2) Like the novices, she used the keyword searching to seek information and never opened some files to verify her hypotheses. This increased our doubt of categorizing this subject as an expert and supported the use of cognitive characteristics to classify subjects.

5.3 Software-Specific Proficiencies

This third criterion for defining expertise considers an individual's level of proficiency with a software-specific area. While we knew that expertise was domain-specific, we underestimated the degree of this specificity. A number of the novices had experience working with web applications and this contributed to their performance on tasks. We had expected that a software engineer with many years of experience would have acquired a set of skills that were broadly applicable and would allow them to perform as experts even outside of their usual application domain. This expectation is analogous to the idea that learning a new programming language is easy after one has done it a couple times already. However, this was not the case.

Table 7: Experience by Performance on Task A

Years of Experience	Web Application Experience	Task A Score
Expert	3-5 years	100%
Novice	1-3 years	84%
Novice	1-3 years	72%
Expert	0-1 years	87%
Novice	0-1 years	57%
Expert	*	59%

Table 7 and Table 8 show the scores on the Change Request Proposal and Implementation tasks sorted by the amount of experience with web applications. We found that novices with some web application experience sometimes outperformed experts without any web application experience. For example, one expert was stymied on the implementation task (Task B), because he wasn't able to run the modified web application. While he had no trouble compiling the application, he didn't realize that the Tomcat server needed to be restarted in order to load the newly compiled program.

Table 8: Experience by Performance on Task B

Years of Experience	Web Application Experience	Task B Score
Expert	5-10 years	100%
Expert	3-5 years	64%
Novice	1-3 years	90%
Expert	1-3 years	80%
Novice	1-3 years	63%
Novice	0-1 years	22%

This criteria for assessing expertise is more complex than the other two, taking into account both experience and domain knowledge. However, it shows the most potential for accurately characterizing software engineering expertise. This complexity means that an empirical test will need to be defined separately for each proficiency of interest. The challenge will be to find a general, but useful, way of measuring expertise in this manner. We believe that this definition is the most promising of the three and further experiments are under way.

6. FUTURE WORK

The research reported in this paper represents a first step in studying software engineering expertise. An empirical test for identifying expertise is necessary so that we can correctly categorize subjects in future studies. We plan to build on this work in a number of ways.

One of the next steps is to construct an inventory of software engineering skill areas. What is the range of skills and knowledge that a software engineer needs to master en route to becoming an expert? Some of these skills are technical, such as programming, testing, and best practices.

Other skills are more social, such as project management, and team work. This inventory will be beneficial for our next task.

We intend to conduct further experiments to refine a proficiency-based definition of expertise. The task in these experiments will be focused on web applications and involve a larger number of subjects. We will be constructing a ladder of skills and knowledge areas, where the rungs on the ladder represent different levels of expertise.

Once we have established a good definition and empirical test, we can then move on to study the work practices and work products of experts. It would not be difficult to incorporate the strategies used by experts into software engineering curriculum, for instance, showing students how to read code in a top-down fashion and what kind of knowledge is needed to plan a code change. In the short term, such changes would ensure that new graduates are better equipped to handle problems in the workplace. In the longer term, educational innovations would also benefit practicing software engineers, resulting in skill improvements over their working lives and the software that they develop.

Last, but not least, the knowledge accumulated by experts over many years can provide insight into software engineering itself. While this knowledge may have been acquired informally, we can certainly study and formalize it. Furthermore, it represents years of experience that we, as researchers, could not experience ourselves and must study empirically.

7. SUMMARY

In this paper, we consider three candidate definitions for expertise and their corresponding test criteria. These three criteria were: number of years of work experience in software engineering; cognitive characteristics of experts; and software-specific proficiencies. We found that each definition has its place in characterizing software engineering expertise.

The first definition, number of years of work experience, is easy to test. Consequently, it is a good first filter, provided that the threshold for an expert is set appropriately. In our pilot study, we used a threshold of 5 years and found that this was not high enough. Ten years may be required to attain a level of expertise that is observably and consistently different from novices.

The second definition was a set of cognitive characteristics found in experts from a number of different domains. This criteria was useful because they provide a theoretical basis for characterizing expertise. As a result, the presence or absence of these characteristics could be used to evaluate the other definitions. In our study, we found patterns of behavior that were consistent with these characteristics. Moreover, putative novices and experts exhibited them to different degrees, leading us to our final definition.

The third definition was software-specific proficiencies. This is a fine-grained definition that considers an individual's skill and knowledge in a particular area. This definition can potentially provide a detailed characterization of expertise, but it is time-consuming to develop and use an empirical test associated with this definition.

Our findings regarding these three definitions mirror the strategies that companies use to select new employees. They publish a job advertisement like the one at the beginning of this paper. It includes the number of years of experience desired, characteristics of a successful candidate, and a long list of required skills and education. Applicants submit a resume containing information

about themselves in the aforementioned areas. A few people are chosen to be interviewed by a manager and some of these are subsequently given technical interviews.

The work reported in this paper is the start of a sequence of work on software engineering expertise. A definition and empirical test is necessary to help identify experts. Lessons learned from studying them can be used to improve curriculum, support future software tool and method development, and better understand software engineering itself.

8. ACKNOWLEDGMENTS

Our thanks to Justin Beltran, Matt McMahand, Teerawat Meevasin, John Situ, Derrick Tseng, Jonathan Zargarian for providing invaluable assistance in preparing and conducting the experiment. Thanks also to Jeff Elliott and Yuzo Kanomata for helping us with initial pilot testing.

9. REFERENCES

[1] Abran, A., Moore, J. W., Bourque, P., Dupuis, R., and Tripp, L. J., "Guide to the Software Engineering Body of Knowledge,": IEEE Computer Society, 2004.

[2] Adelson, B., "When Novices Surpass Experts: The Difficulty of a Task May Increase with Expertise," *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *10*, (1984), pp. 483-495.

[3] Adelson, B. and Soloway, E., "A Model of Software Design," in *The Nature of Expertise*, Michelene T. H. Chi, Robert Glaser, and M. J. Farr, Eds. Hillsdale, NJ: Lawrence Erlbaum Associates, Publishers, pp. 185-208, 1988.

[4] Ahmed, S. and Wallace, K. M., "Understanding the Knowledge Needs of Novice Designers in the Aerospace Industry," *Design Studies*, *25*, (2004), pp. 155-173.

[5] Berlin, L. M., "Beyond Program Understanding: A Look at Programming Expertise in Industry," *Proc. Empirical Studies of Programmers: Fifth Workshop*, Palo Alto, CA, pp. 6-25.

[6] Brooks, F. P., "No Silver Bullet: Essence and Accident in Software Engineering," *IEEE Computer*, *20*, 4, April, (1987), pp. 10-19.

[7] Brooks, R., "Towards a Theory of the Comprehension of Computer Programs," *International Journal of Man-Machine Studies*, *18*, (1983), pp. 543-554.

[8] Campbell, R. L., Brown, N. R., and DiBello, L., "The Programmer's Burden: Developing Expertise in Programming," in *The Psychology of Expertise: Cognitive Research and Empirical AI*, Robert R. Hoffman, Ed. New York: Springer-Verlag, pp. 269-294, 1992.

[9] Cross, N., "Expertise in Design: An Overview," *Design Studies*, *25*, (2004), pp. 427-441.

[10] Curtis, B., Krasner, H., and Iscoe, N., "A Field Study in the Software Design Process for Large Systems," *Communications of the ACM*, *31*, 11, (1988), pp. 1268-1287.

[11] Davies, S. P., "Knowledge Restructuring and the Acquisition of Programming Expertise," *International Journal of Man-Machine Studies*, *40*, (1994), pp. 703-726.

[12] Ericsson, K. A. and Charness, N., "Expert Performance: Its Structure and Acquisition," *American Psychologist*, *49*, 8, August, (1994), pp. 725-747.

[13] Ericsson, K. A. and Simon, H. A., *Protocol Analysis- Revised Edition: Verbal Reports as Data*. Cambridge, MA: The MIT Press, 1993.

[14] Glaser, R. and Chi, M. T. H., "Overview," in *The Nature of Expertise*, M. J. Farr, Ed. Hillsdale, NJ: Lawrence Erlbaum Associates, Publishers, pp. xv-xxviii, 1988.

[15] Gugerty, L. and Olson, G. M., "Comprehension Differences in Debugging by Skilled and Novice Programmers," *Proc. Empirical Studies of Programmers (ESP1)*, Washington, DC, pp. 13-27, 5-6 June 1986.

[16] Guindon, R., "Designing the Design Process: Exploiting Opportunistic Thoughts," *Human-Computer Interaction*, *5*, (1990), pp. 305-344.

[17] Holt, R. W., Boehm-Davis, D. A., and Shultz, A. C., "Mental Representations of Programs for Student and Professional Programmers," *Proc. Empirical Studies of Programmers, Second Workshop*, Washington, DC, 7-8 December 1987.

[18] Johnson, E. J., "Expertise and Decision under Uncertainty: Performance and Process," in *The Nature of Expertise*, Michelene T. H. Chi, Robert Glaser, and M. J. Farr, Eds. Hillsdale, NJ: Lawrence Erlbaum Associates, Publishers, pp. 209-228, 1988.

[19] Koenemann, J. and Robertson, S. P., "Expert Problem Solving Strategies for Program Comprehension," *Proc. SIGCHI Conference on Human Factors in Computing Systems: Reaching through Technology*, New Orleans, LA, USA, pp. 125-130, 27 April - 2 May 1991.

[20] Larkin, J., McDermott, J., Simon, D. P., and Simon, H. A., "Expert and Novice Performance in Solving Physics Problems," *Science*, *208*, 4450, June 20, 1980, (1980), pp. 133-1342.

[21] Letovsky, S., "Cognitive Processes in Program Comprehension," *Proc. Empirical Studies of Programmers, First Workshop*, Washington, DC, pp. 58-79, 5-6 June 1986.

[22] Linn, M. C. and Clancy, M. J., "The Case for Case Studies of Programming Problems," *Communications of the ACM*, *35*, 3, March, (1992), pp. 121-132.

[23] Littman, D. C., Pinto, J., Letovsky, S., and Soloway, E., "Mental Models and Software Maintenance," *Proc. Empirical Studies of Programmers, First Workshop*, Washington, DC, pp. 80-98, 5-6 June 1986.

[24] Mastaglio, T. and Rieman, J., "How Experts Infer Novice Programmer Expertise: A protocol analysis of LISP code evaluation," *Proc. Empirical Studies of Programmers, Fourth Workshop*, New Brunswick, NJ, pp. 177-195.

[25] Perkins, D. N., Schwartz, S., and Simmons, R., "A View from Programming," in *Toward a Unified Theory of Problem Solving*, Mike U. Smith, Ed. Hillsdale, NJ: Lawrence Erlbaum Associates, Publishers, pp. 45-67, 1991.

[26] Riecken, D. R., Koenemann-Belliveau, J., and Robertson, S. P., "What Do Expert Programmers Communicate By Means of Descriptive Commenting?," *Proc. Empirical Studies of Programmers, Fourth Workshop*, New Brunswick, NJ, pp. 177-195.

[27] Rist, R. R., "Plans in Programming: Definition, Demonstration, and Development," *Proc. Empirical Studies of Programmers, First Workshop*, Washington, DC, pp. 28-47, 5-6 June 1986.

[28] Simon, H. A. and Chase, W. G., "Skill in Chess," *American Scientist*, *61*, (1973), pp. 394-403.

[29] Sutcliffe, A. G. and Maiden, N. A. M., "Analysing the Novice Analyst: Cognitive Models in Software Engineering," *International Journal of Man-Machine Studies*, *36*, (1992), pp. 719-740.

[30] Tan, S. K. S., "The Elements of Expertise," *Journal of Physical Education, Recreation & Dance*, *68*, 2, February, (1997), pp. 30-33.

[31] Wiedenbeck, S., “The Initial Stage of Program Comprehension,” *International Journal of Man-Machine Studies*, *35*, (1991), pp. 517-540.
[32] Wiedenbeck, S., Fix, V., and Scholtz, J., “Characteristics of the Mental Representations of Novice and Expert Programmers: An Empirical Study,” *International Journal of Man-Machine Studies*, *39*, (1993), pp. 793-812.
[33] Wong, S.-Y., Cheung, H., and Chen, H.-C., “The Advanced Programmer's Reliance on Program Semantics: Evidence from some Cognitive Tasks,” *Internaional Journal of Psychology*, *33*, 4, (1998), pp. 259-268.