International Journal of Management Science
Vol. 2, No. 1, December 1996

An Empirical Study on the Cognitive Difference between the Creators and Users of Object-Oriented Methodology*

Jinwoo Kim · Jungpil Hahn**

Abstract

The main objective of this study is to uncover the differences in the programming behavior between methodology creators and methodology users. We conducted an experiment with methodology creators who have invented one of the major object-oriented methodologies and with professional programmers who have used the same methodology for their software-development projects. In order to explain the difference between the two groups, we propose a theoretical framework that views programming as search in four problem spaces: representation, rule, instance and paradigm spaces. The main problem spaces in programming are the representation and rule spaces, while the paradigm and instance spaces are the supporting spaces. The results of the experiment showed that the methodology creators mostly adopted the paradigm space as their supporting space, while the methodology users chose the instance space as their supporting space. This difference in terms of the supporting space leads to different search behaviors in the main problem spaces, which in turn resulted in different final programs and performance.

---

* This paper was supported by NON DIRECTED RESEARCH FUND, KOREA Research Foundation, 04-C-0210.

**Department of Business and Administration, Yonsei University

1. Introduction

One of the most important artifices in the modern era is the software that runs computers. In general, the key characteristics of artifices are their functions and goals. The creator of artificial things is concerned with how things ought to be in order to attain their user's goal. In the case of software, the programmer is the creator, and the customer who utilizes the software to accomplish his / her tasks is the user. For example, in order to write term papers, students (users) run a word processing software that is developed by application developers (creators). The cognitive differences between the creators and users of computer software has been the subject of several studies. For example, Laurel (1986) investigated criteria for formulating a creator model of programs that makes sense to users, Mark (1986) provided guidelines for programmers to ensure that a good creator model is well expressed in the software, and diSessa (1986) analyzed the ways that software affects the formation and evolution of the users' understanding of given programs.

A more abstract, but maybe more important, artifice in the field of software development is the *methodology*. A software methodology provides the basic architecture, procedure and tools for developing computer programs (Jacobson, 1995). Several different methodologies, such as structural programming (Dahl, Dijkstra, and Hoare, 1972; Yourdon, 1989), data-oriented methodologies (Jackson, 1983), and object-oriented methodologies (Rumbaugh, Blaha, Premerlani, Eddy and Lorensen , 1991; Booch, 1994) have been invented by industry gurus, and these methodologies have made substantial impacts on the overall computer industry. However, unlike the numerous studies on the *actual programming*, few study has been conducted on the user-creator issue in *programming methodology*. In this case, the programmer should be the user, and the methodology inventor should be the creator. For example, in order to develop an object-oriented payroll application, programmers (user) employ the OMT methodology developed by the methodology creators (Rumbaugh, et al, 1991).

This study investigates the cognitive process of object-oriented programming focusing on the differences between methodology creators and methodology users. There are two objectives in this study. First, it provides a cog-

nitive theory that can explain the difference between the user and the creator in the object-oriented *methodology*. The theory should be comprehensive as to account for other activities in object-oriented *programming* such as program comprehension and reuse. The theory developed in this paper is based on the concept of search in four different problem spaces (Klahr and Dunbar, 1988; Kim, Lerch and Simon, 1995; Schunn and Klahr, 1995). Second, it provides detailed results of an experiment in which three creators who had invented one of the major object-oriented methodologies participated as subjects. The experiment also included three professional programmers who were users of the object-oriented methodology that our creator subjects had invented. Results from the experiment show that the different cognitive processes between users and creator can be explained in terms of the different search behavior in the four problem spaces. The next section of this paper provides the theory of programming as search in the four problem spaces. Section three explains our experimental design and procedure. Section four provides detailed analyses of the experiment. And the last section concludes this paper with general discussions.

2. Programming as Search in the four problem spaces

2.1. Programming as scientific discovery

Problem solving, especially for simple tasks such as puzzles, has been described as a search in a single problem space (Newell and Simon, 1972). However, as the complexity of the task increases, we need a more comprehensive framework to fully understand the complexity of the cognitive activities involved. Multiple problem spaces have been proposed as a way to deal with these more complex tasks, such as rule induction (Simon and Lea, 1974) and Scientific Discovery (Klahr and Dunbar, 1988; Baker and Dunbar, 1996). The multiplicity of problem spaces adds significantly more explanatory power that we need in the areas of complex tasks (Burns, 1996).

Computer programming is a complex and difficult cognitive task (Simon, 1973; Jeffries, Turner, Polson and Atwood, 1981; Mayer, 1981; Letovsky, 1986; Guindon, 1990). In this paper, we regard programming as problem

solving in four problem spaces: the rule space, the representation space, the instance space, and the paradigm space. Our view of programming is similar to current cognitive theories of scientific discovery (Simon and Lea, 1977; Langley, Simon, Bradshaw, and Zytkow, 1987; Klahr and Dunbar, 1988; Klahr, Fay, and Dunbar, 1993; Qin and Simon, 1990; Schunn and Klahr, 1995). Prior studies in scientific discovery have investigated how people generate hypotheses (i.e., search in the rule space) to develop rule systems. A rule system is a well-defined set of propositions for predicting or explaining a phenomenon (Holland, Holyoak, Nisbett and Thagard, 1986). A computer program can be portrayed as rule system for solving a set of task instances (Hoc and Nguyen-Xuan, 1990), in the same way that rule systems in scientific discovery aim to explain a set of related experimental results (Klahr and Dunbar, 1988). Empirical results indicate that scientific discovery is conducted by interrelated searches in the four problem spaces, which implies that programming may be also conducted by interrelated searches in the four problem spaces.

2.2. The four problem spaces

This section presents in more detail our theoretical framework on the four problem spaces (rule, representation, instance and paradigm spaces) in which programmers and scientists perform search to develop rule systems (theories in the case of scientists, and programs in the case of programmers).

In the rule (hypothesis) space, both scientists and programmers develop hypotheses for solving or understanding the related phenomena. They either transform existing rules into the solution of their current problem (Holland, Holyoak, Nisbett, and Thagard, 1986; Dunbar, 1993), or use solution instances to generate new rules or test existing ones (Langley, Simon, Bradshaw, and Zytkow, 1987). For example, Coulomb derived his inverse-square laws of electrical forces by transforming Newton's laws of gravitational forces into his rules for electrical and magnetic forces (Langley, Simon, Bradshaw, and Zytkow, 1987). On the other hand, Kepler used the instance of the planetary relation between Mercury and the Sun for generating a general rule about the relation between the planets and the

Sun for generating a general rule about the relation between the planets and the sun. Similarly, many empirical studies in programming have also observed similar behavior in both highly skilled professional programmers (Kim and Lerch, 1992; Guindon, Curtis, and Krasner, 1987) and novices (Jeffries, Turner, Polson, and Atwood, 1981).

The representation space is the set of all possible representations for a given problem. A representation is a mental model which encodes the programmer's current understanding of the target problem (Letovsky, 1986). In the representation space, scientists search for suitable representations from the set of possible features by using analogy or brute force search (Schunn and Klahr, 1995; Baker and Dunbar, 1996). In the same way, programmers search for appropriate data representations for their programs in the representation space. Kim, Lerch and Simon (1995) provided an ample amount of empirical data about programmers' search in the representation space. Moreover, Kim and Lerch (in press) found that programmers also use analogy and brute-force search in the representation space.

In the instance (experimental) space, scientists design and conduct experiments to test or falsify their current hypotheses. The results are instances (i.e., experimental results) that may support a hypothesis or cast doubts about the theory (Klahr and Dunbar, 1988). The activities in the instance space are called mental simulations in programming. Guindon et al., (1987) found that programmers use a strategy called "explorative mental simulation" which helps them generate new rules when they have no clue about how to start the programming process. Jeffries, Turner, Polson, and Atwood (1981) proposed that creators use "problem solving by understanding" when they need to generate an initial rule. Mental simulations are also used to refine the content of existing rules by guiding the programmer during incremental changes. Adelson and Soloway (1985) found that programmers use mental simulation for "systematic expansion" — refining programs by using the results of partial solutions. Finally, programmers rely heavily upon mental simulation for evaluating the validity of rules (Kant and Newell, 1984). Detienne and Soloway (1990) found that their subjects used Mental simulations when they wanted to evaluate the external coherence among several specific programming plans that were derived from general program plans.

Finally, in the paradigm space, scientists adopt a new way of thinking to make an important discovery (Kuhn, 1970). The new way of thinking includes a new method for gathering data in the instance space (Schunn and Klahr, 1995), a new way to view existing rules (Burns and Vollmeyer, 1996), or a new way to coordinate search for data representations (Schunn and Klahr, 1995). Therefore, the paradigm space in scientific discovery has significant impact on the entire discovery process, because the selected paradigm provides or modifies the criteria of search strategies in the other three problem spaces. However, compared to the various studies about the paradigm space in scientific discovery (Schunn and Klahr, 1995; Schunn and Klahr, 1996), few empirical studies show that people also use the paradigm space in programming. This study compares the methodology creators and methodology users in programming to investigate the cognitive activities in the paradigm space.

2.3. Methodology creators vs. Methodology users

The key aspect in the architecture of multiple problem spaces is the interaction among activities in the individual spaces (Burns, 1996). We characterize the interaction based on the classification of the four problem spaces into either the main space or the supporting space. The main space is where the actual task is performed, whereas search in the supporting space is mostly done to provide information to constrain search in the main space. Therefore, the distinction between the main and supporting spaces is determined by the characteristics of the task. For example, in programming, the rule and representation spaces are the main spaces, because programs consist of data structure and algorithms: an appropriate data structure is searched in the representation space, and an adequate algorithm is searched in the rule space. On the other hand, the instance and paradigm spaces are the supporting spaces in the programming task, since search in the two spaces provides valuable information to the search in the main spaces, but no direct work is done for the design of the final program. In programming, we expect that search is mostly done in the rule and representation spaces, because in programming these are the two main spaces. On the other hand, search in the paradigm and instance spaces is done mainly to support

search in the rule and representation spaces. Therefore, we expect both the methodology creators and users to spend more time in the rule and representation spaces than the other two supporting spaces.

However, we hypothesize that methodology creators and methodology users to act differently in terms of which of the two supporting spaces are utilized more heavily. The creators will exploit search in the paradigm space more than the users, while the users will depend on the search in the instance space more heavily than the creators. The methodology creators had the experience of comparing various paradigms or methodologies when they were inventing their own methodology. Therefore, they have the knowledge about which method appropriate for a certain type of problem. So, they are going to show a top-down search behavior, in which they first decide the appropriate paradigm for a given problem, then search for the appropriate data representation and rules based on the paradigm selected. Consequently, search in the two main spaces will be much simpler for the creators, and significantly less time will be devoted to search in the instance space. On the other hand, the methodology users who lack the experience in the paradigm space will depend on the most general search strategy, which is the generate-and-test heuristic (Newell and Simon, 1972). The generate-and-test heuristic depends heavily on the search in the instance space, since the generated rules and representations have to be tested in the instance space. Therefore, methodology users will spend more time in the instance space compared to the creators. Finally, their search in the main spaces will be more complex, with more backtracking since they lack the top-down guidance from the paradigm space, and depend on the results of individual instances.

In sum, even though the creators and users have the same main problem spaces, they are expected to have a different major supporting space. The major supporting space for the creators is the paradigm space, whereas the major supporting space for the users is the instance space. The top-down knowledge from the paradigm space results in a simple search in the creators' main spaces, whereas the bottom-up knowledge from the instance space results in a more complex search in the users' main spaces. In order to investigate our theory with empirical data, we conducted an experiment that is explained in detail in the next section.

3. Experiment

The experiment in this study investigates how creators and users of the object-oriented methodology write computer programs. The experiment focuses on the difference between these two groups in terms of search between and within the four problem spaces.

3.1. Subjects

Six subjects took part in the experimental study. The subjects were divided into two groups: the creator group and the user group. The creator group was composed of three methodology developers who had actually invented one of the major object-oriented methodologies (Rumbaugh et. al, 1991). All the subjects in the creator group were project team leaders in the development of the object-oriented methodology. The subjects in the user group were programmers who used the object-oriented methodology that the subjects in the creator group had invented. These subjects were professional systems engineers who had experience with large scale system development projects. None of the six subjects were familiar with the problem given in the experiment.

3.2. Experimental Design

The two groups were asked to write a computer program for a problem described in a single sheet of paper. The problem used in the experiment was the monster problem which is a Tower of Hanoi (TOH) problem isomorph (Hayes and Simon, 1977). The monster problem has monsters and globes of three different sizes. The monsters change their globe sizes so that each would have a globe proportionate to its size. However, the changes were subject to three constraints. The constraints were: 1) only one globe may be changed at a time, 2) if two globes have the same size only the globe held by the larger monster may be changed, and 3) a globe may not be changed to the same size as the globe of a larger monster. The monster problem seems like a simple puzzle, yet it is very complex to write a general rule for all the possible initial states because of the three constraints built

into it.

The subjects were asked to understand the given problem thoroughly and then to write a computer program in detailed pseudo-code that shows by what sequence of changes the monster can solve the problem given any random initial arrangement.

3.3. Coding Schema

We used protocol analysis (Ericsson and Simon, 1993; Van Someren, Barnard and Sandberg, 1994) to compare the cognitive search activities in the four problem spaces between the creator group and the user group. The verbal utterances during the problem solving session were the major source of data for our experimental study. Along with the verbal protocols, written program protocols and action protocols were also collected, since the use of both verbal and action protocols is expected to provide a more complete trace of problem solving behavior (Rist, 1989).

Two important preconditions of a protocol analysis are to identify an appropriate unit of analysis, and to develop an objective coding system for each unit. In our study, considering the volume of data, we chose to use episodes as the unit of analysis. An episode is a small self-contained phase of highly organized activity (Newell and Simon, 1972). Each episode was classified as one of the following coding schema: Paradigm, Representation, Rule, Instance, Read, or Other.

The Paradigm represents any cognitive activity which provides or modifies the criteria of importance, relevancy, or interest for search in the other three spaces. Comments about a methodology, comments about plans to apply new test methods, and justification about these comments were classified as *Paradigm*. For example, an episode where the subject states that a brute force search algorithm will suffice to solve the problem is classified as a *Paradigm*. *Representation* is the activity of constructing a problem representation. A representation is composed of three elements: entities, relations, and roles. Episodes were classified as *Representation* if activities in the episode were related to building or changing data structure of the program. For example, subjects' underlining nouns or assigning properties to an entity are designated as representations. Also subjects defining the

data structure for the program is classified as representation. Episodes were classified as *Rule* if subjects create or modify any systematic piece of problem solving knowledge that allows entities in the representation to direct search. In order to be classified as a rule, the knowledge should be able to compute the next move or at least direct the search for implementing one or more roles. For example, subjects commenting that a particular monster should be dealt with first can be regarded as a rule. Episodes were classified as Instance if activities in the episode were related to actual traversing of a solution path in the instance space. In order to be regarded as an instance the episode should contain at least one comment about the solution being traced. For example, designing a specific case and running it through to test a rule is classified as an instance. *Read* episodes relate to literal reading from the problem description. Reading is not actually part of the theoretical framework but was included in the coding system for its high frequency, the ease of its identification from the protocol data, and also the difficulty in inferring the motivations behind it. Finally, *Other* episodes represent comments that do not directly relate to the problem solving activity. Irrelevant remarks, interruptions and interactions with the experimenter were classified as Other.

3.4. Analysis Procedure

The problem solving sessions were videotaped and verbal protocol analysis was used to analyze the data (Ericsson and Simon, 1993; Van Someren, Barnard and Sandberg, 1994). The data analysis was performed step by step across all subjects to increase reliability. First, all the verbal protocols were transcribed with timing marks and written comments of the subjects. Then all the transcriptions were verified and segmented into episodes by the second author. The segmented protocols were verified and then coded according to the coding system described in the previous section. The segmentation and codification were verified by the first author, and any discrepancies were resolved before any further work proceeded. Finally, four Problem Behavior Graphs (PBG) and one Space Transition Graph (STG) were built for each subject based on the coded data. The PBG portrays problem solving activities as a series of search within a problem space (Newell and Simon,

1972). Four PBGs were built for each subject to investigate the search behavior in each of the four problem spaces (the Rule Behavior Graph in the rule space, the Simulation Behavior Graph in the instance space, the Representation Behavior Graph in the representation space, and the Paradigm Behavior Graph in the paradigm space). The STG which is similar to the flow model used by Fisher(1987) portrays the problem solving activities as a series of transitions between different problem spaces.

4. Results

4.1. Final Results

All the subjects were successful at writing the pseudo-code with the correct logic. However, the overall performance differed in terms of design time and final output. Table 1 represents the total time in seconds the subjects took to write the final program as well as the average time for each group. The subjects in the user group took approximately three times more time to come up with the final program than the subjects in the creator group ($t(4) = 2.57$, $p<0.05$).

Table 1. Total Design Time

Group	Subject	Time
Creator	CRT1	2973
	CRT2	2266
	CRT3	2532
	Average	2590.33
User	USR1	3670
	USR2	9289
	USR3	8964
	Average	7307.67

The final programs designed by the three subjects in the user group were very similar to one another in that they all used a specific recursive algorithm. Furthermore, these three programs were very detailed in that they all included actual programming constructs such as data types, expressions and statements. However, the programs written by the three subjects in the creator group were very different from one another and also from the programs of the user group. One of the subjects (CRT1) designed a transition algorithm where the algorithm defines a sequence of transitions for each state. Another subject (CRT2) designed a very general search algorithm where the algorithm produces all possible globe changes and then rules out the ones that violate the constraints given in the problem. And finally one subject (CRT3) also designed a general search algorithm. However, the algorithm designed by CRT3 was different from the one designed by CRT2 in that it does not generate all possible globe changes but generates only possible changes. Furthermore, these three programs were written in very loose pseudo-code. They did include some programming constructs such as 'for', 'do' and 'if then' statements but the majority of the pseudo-code was written in natural language.

In summary, design process took much more time for the user group than for the creator group. Furthermore, the subjects in the user group produced final programs that were much more specific and detailed than the programs written by the subjects in the creator group. The next section provides detailed process results to explain why users spent more time and produced more detailed programs.

4.2. Process Results

We coded all the subjects' verbal protocols for both the creator and user groups using the coding scheme. Although the analysis of the verbal protocols most accurately shows the dynamic nature of search in the four problem spaces, it is impossible to present all these data in detail for all the subjects because of the large mass of data. Therefore, we decided to present the analysis of one subject in each group in detail. The subjects to be analyzed in detail were selected because his cognitive processes were representative of the behavior of the group. In this section we will provide

detailed analysis for one subject in each of the two groups. Afterwards we will generalize the findings from the detailed analysis by providing aggregate results for all the subjects in both groups.

4.2.1. Detailed Analysis of one Subject in the Creator Group

One subject in the creator group (CRT2) was selected for the detailed analysis. Subject CRT2 spent 2266 seconds (37 minutes 46 seconds) designing his final solution program. During this period he spent 296 seconds (13.06% of total design time) in the paradigm space, 893 seconds (39.41%) in the representation space, 369 seconds (16.28%) in the rule space, 228 second (10.06%) reading the problem description and 408 seconds (21.18%) with comments that do not directly relate to the problem solving activity. CRT2 did not enter the instance space because he did not run any actual solution paths to develop or test his rules. The Space Transition Graph (STG) in Figure 1 shows his design process in terms of transitions between the four problem spaces.

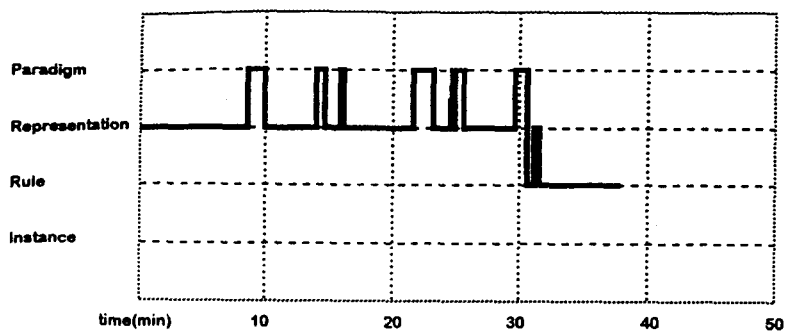

Figure 1. Space Transition Graph(STG) for CRT2

The vertical axis in Figure 1 represents the four problem spaces: paradigm, representation, rule and instance spaces: and the horizontal axis represents the time in minutes. The subject (CRT2) constructed his first representation of the problem by considering the data from the problem description and spent about 8 minutes 30 seconds in the representation space.

Afterwards, he transitioned to the paradigm space where he set up his initial problem solving strategy: to solve by hand initially and then think about writing the program. This is important because he did not constrain himself to a particular programming paradigm but opened up a wide variety of options to choose from. Afterwards, CRT2 kept constructing and modifying his representation (he built an object model with monster and globe objects which he though was trivial and a dynamic model with transitions of globe sizes) up until about 14 minutes 40 seconds where he once again entered the paradigm space to fix his interest on the transitions and constraints on the transitions. Then he came back to the representation space where he built up a new representation where the entities were no longer monster and globes but the events that cause the transitions. At around 21 minutes 40 seconds he searched the representation space for alternative representations of the problem and also the paradigm space for alternative methods of programming to match the various representations he had built in the representation space and came up with two alternatives: an AI programming approach with a backtracking algorithm and a brute force search code. At the end of his search in the paradigm space, he selected to adopt a brute force search approach without the backtracking algorithm. He chose this approach by justifying that the problem had only twenty seven possible initial globe states and that one did not need backtracking for such a small problem. He also built up a representation corresponding to the paradigm he had finally selected (the representation built was a tree of path states consisting of monster-globe size pairs). At 30 minutes 40 seconds he began to construct the rules to solve the problem. This process of rule construction was very straight forward. He only exited the rule space once to enter the representation space early in the rule construction process to build a formal representation for the data structure of his solution. The rule construction process lasted 7 minutes 10 seconds and ended his design at 37 minutes 46 seconds.

In short, the subject (CRT2) demonstrated a straight forward top-down design process where he initially constructed many representations while searching in the paradigm space for an adequate paradigm. And once he had fixed his paradigm and representation, he started to develop the rules to solve the problem in a very straight-forward manner. Next we will de-

scribe his search behavior in the representation and rule spaces in detail.

The Representation Behavior Graph in Figure 2 and the Rule Behavior Graph in Figure 3 shows the search process of CRT2 in the two main spaces (representation and rule spaces, respectively). Each square represents a representation (or rule) and the representation episode number (or rule episode number) is denoted inside the square. Refinement of a representation (or rule) is denoted by going to the right one step, whereas backtracking to an earlier representation (or rule) is represented by moving left to the earlier representation (or rule) and down one level.

Figure 2 shows that CRT2 constructed his initial representation (Rp1) and refined it three times (Rp1~Rp4). Then as he became aware that he had constructed a faulty representation, he backtracked a step to construct Rp5, which he refined to Rp16. After Rp16, CRT2 constructed a whole new representation Rp17. That was because he has changed his paradigm to emphasize the events causing the transitions among globes and the constraints on these transitions. He was no longer interested in entities such as globes and monsters. Thus the new representation was constructed with the events that trigger the transitions as the entities of the representation. CRT2 continued to change his representations which was directed by the search in the paradigm space. We can see in the Representation Behavior Graph that CRT2 changed his representations very frequently.

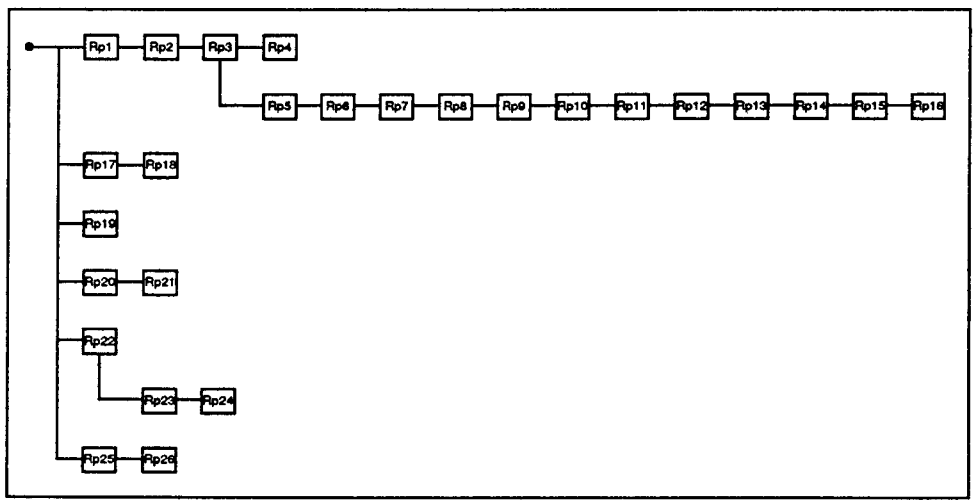

Figure 2. Representation Behavior Graph for CRT2

Figure 3 shows the rule construction behavior of CRT2 in the rule space. CRT2 did not design his rules until he had come up with a satisfactory paradigm and representation. The rules he had developed were very straight forward and without much interruption. We only coded 3 rule episodes for CRT2. In the first rule episode he constructs the outer loop of the program to direct his search algorithm. In the second episode he defines the initialization of the input data, and in the last he designs the actual search algorithm (the inner loop). Furthermore, the actual algorithm of the program was not derived from any search activity in the other spaces (e.g. the instance space).

Figure 3. Rule Behavior Craph for CRT2

In sum, CRT2 showed a very straight forward top down rule design process. However, his search in the representation space was very active in that he had built many new representations without much refinement. Furthermore, the search in the representation space seemed to have been guided by the information from the paradigm space.

4.2.2. Detailed Analysis of one Subject in the User Group

The third subject in the user group (USR3) was selected for a detailed analysis because his problem solving process showed the most characteristic traits of methodology users. The total time consumed by subject USR3 to design the final program was 8964 seconds (2 hours 29 minutes 24 seconds). Of this time, he spent 77 seconds (0.86% of total design time) in the paradigm space, 2482 seconds (27.69%) in the representation space, 2172 seconds (24.23%) in the rule space, 4233 seconds (47.22%) in the instance space, 349 seconds (3.89%) reading the problem statement and 552 seconds (6.16%) making irrelevant comments. The Space Transition Graph in Figure 4 shows the actual problem solving process for USR3 in terms of the transitions be-

tween the various problem spaces and the actual time spent in each space.

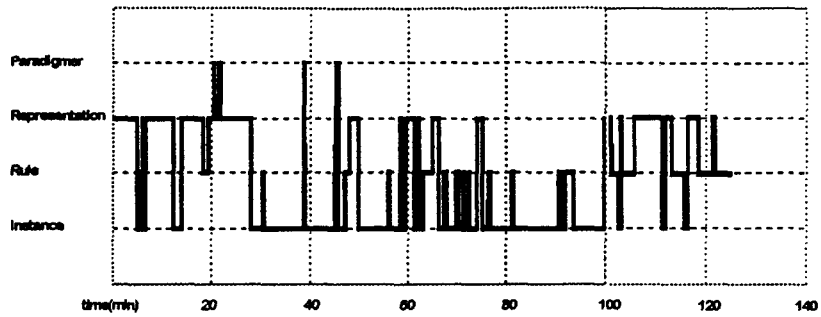

Figure 4. Space Transition Graph(STG) for USR3

The most striking feature of the problem solving behavior is the frequency of the transitions between the various spaces, as opposed to the cut and dried top-down problem solving process of the creator evidenced in the previous section. USR3 spent a large proportion of his time in the instance space. In other words, he had to go through a number of mental simulations to understand the problem situation and construct the appropriate representation and rules to solve the problem. Although the initial representation was constructed from the problem statement, USR3 tended to drop back to the instance space to assure himself that he was on the right track. As a consequence, most of the rules developed by USR3 were inferred from the instances he had solved during the design phase. Another distinguishing characteristic of the problem solving behavior of USR3 is that he constrained himself to the Object-Oriented paradigm from the start. The two transitions into the paradigm space as can be seen immediately after the 20 minute time period are expressions of the subject's preoccupation with conforming to the Object Oriented methodology. This was explicitly stated in his protocols (*"let's see, what is the most object-oriented way of doing this"* and *"and uh and then you gotta compare the numbers which, which doesn't sound too object-oriented to me"*). In other words, the subject did not consider other possible programming paradigms that could have been applied to the solution, but instead immediately jumped into the OO paradigm to solve the problem.

During the first 20 minutes of the design phase, the subject concerns himself with constructing the object configuration of the problem representation. In other words, he designed the classes that were needed, together with their attributes, methods and associations. After getting a general idea of how to configure the object structure the subject proceeded to design the algorithm for solving the problem. From this point, the subject tended to linger in the instance space, trying different problems in an attempt to clarify how the constraints actually affect the algorithm to implement the solution. In short, the design strategy employed by USR3 can be characterized as a generate and test heuristic. This is well described in the complex web of transitions between the different problem spaces. The subject mentally simulates various problem instances, during which process he constructs the object representation and algorithm to solve the problem. In this way the subject continues to refine the object definition and rule configuration into a detailed C^{++} pseudo-code.

USR3's search behavior in the representation and rule spaces will be examined in more detail below. Figure 5 and Figure 6 show the Rule Behavior Graph and the Representation Behavior Graph of USR3.

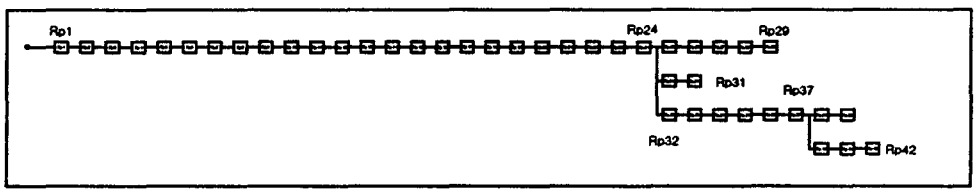

Figure 5. Representation Behavior Graph for USR3

The representation development process of USR3 was relatively straightforward with few backtracking episodes. The initial representation was preserved until the end of the design phase with slight modifications made in the detailed C^{++} code used to implement the object configuration (changes in the object attributes, methods or associations). The representation was based on the object oriented paradigm and the subject simply added the methods and attributes to the objects in the initial phase. The brief episodes of backtracking that are represented in this figure are those times when the

subject deleted an operation or attribute from a class that was defined in an earlier stage.

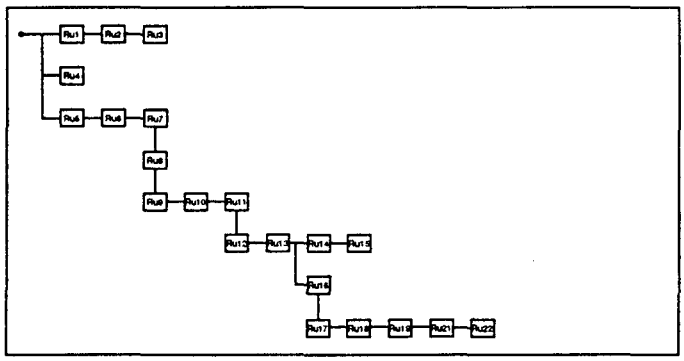

Figure 6. Rule Behavior Graph for USR3

In contrast, the rule behavior graph shows a much more complex solution trail as can be seen in Figure 6. The first three rules are formulated based on a simple problem instance that was done in the beginning. This strand of thinking is modified upon realization that following the rules formulated in the first three episodes would result in major violations of the restrictions. After running through several instances the subject comes to the conclusion that the larger monsters need to be recursively changed to a state that is neither the smaller monster's current state nor the destination state. This thread of thought is maintained in the episodes that lead on to the development of rule 9. This rule is continuously refined in the succeeding rule development episodes. The frequent backtracking that occurs at this point, i. e. for rule 12, indicates a slight change in the ordering of the recursive algorithm so that the smaller of the larger monsters is changed to a uniquely defined state that would allow the smallest monster to change its globe to the proper size. The remainder of the rule episodes represent the time spent by the subject in developing the actual C^{++} algorithm to implement the logical rule developed up to this point. The actual coding process after the logical rule has been completely developed (from Ru17 to Ru22) also shows some backtracking. This occurs as a result of testing the code with sample

instances and then debugging it.

In sum, USR3 showed a very active search in the rule space which resulted in a complex rule development process with frequent refinement and backtracking which was guided by active traversing of solution paths in the instance space. However, the search behavior demonstrated in the representation space was fairly monotonous. In the representation space he showed less refinement and few backtracking: USR3 did not construct entirely new representations from scratch but only slightly different ones based on his initial representations.

4.2.3. Aggregate Results

Thus far we have described the difference between the creator group and the user group in terms of performance (design time) and final output (actual program pseudo-code). We have also described in detail the problem solving behavior of one representative subject from each group. The above analyses suggest that the performance results were due to the fact that the search behavior between and within the four problem spaces were different among the two groups. In this section we will provide the aggregate results to gain more insight into this search behavior.

First of all we compared the actual and relative time spent in each of the four spaces. Table 2 shows the search time spent in the spaces. As we can see in Table 2, the proportion of time spent in the representation and rule spaces was larger than the time spent in the other two spaces. This is obvious because it is in the representation and rule spaces that the actual task of designing the program takes place (designing the data structure and the algorithm to find the solution to the problem). However, it is interesting to note that the creator group spent more time (relative) in the paradigm space than the user group ($t(4)=6.735$, $p<0.05$), whereas the user group spent more time (relative) in the instance space than the creator group ($t(4)=1.693$, $p<0.1$). These results suggest that of the two supporting spaces the creator group referred to the paradigm space more often and the user group referred to the instance space more often and also that compared to the user group, the creator group referred to the paradigm space more, and also the user group referred to the instance space more compared to the creator

group. This implies that in programming the paradigm space is the major supporting problem space for the creator group and that the instance space is the major supporting problem space for the user group.

Table 2. Search Time in the Four Problem Spaces

Group		Space				Total
		Paradigm	Representation	Rule	Instance	
Creator	Average	299.67	720.33	964.33	99.67	2084
	(%)	11.57	21.81	37.23	3.84	80.45
User	Average	33.33	2718	2655.67	1900.67	7307.67
	(%)	0.45	37.19	36.34	26.01	90.76

In order to verify these results we compared the search behavior between the four spaces. Table 3 shows the frequency of between space transitions of the two groups from the most frequent to the least frequent. For example, the creator group showed a total of 15 transitions from the paradigm space to the representation space but did not show any transition from the paradigm space to the instance space. Table 3 shows that the user group demonstrated the most frequent between-space transitions among the representation, rule and instance spaces, whereas the creator group showed the most frequent transitions between the paradigm, representation and rule spaces. Figure 7 shows the between space search patterns of the two groups by representing the four most frequent between-space transitions. The boxes represent the spaces, the arcs represent the between-space transitions and the numbers represent the ranking of the transition in terms of frequency. We can see in Figure 7 that there is a high frequency of transitions between the main spaces (representation and rule spaces), that the major supporting space (the paradigm space for the creator group and the instance space for the user group) interacts strongly with the main spaces, and also that the major supporting space interacts more with one of the two main spaces: for the creator group the paradigm space which is the major supporting space interacts strongly with the representation space of the main

spaces and for the user group the instance space interacts strongly with the rule space. These results conforms to our theoretical framework of programming by showing that the representation and rule spaces are the main problem spaces and that the major supporting space is different among the two groups.

Table. 3 Between Space Transition Ranking

Creator		Ranking	User	
Freq.	Transition		Freq.	Transition
15	para -> rep	1	29	rep -> rule
13	rep -> rule	2	28	rule -> rep
12	rep -> para	3	22	rule -> ins
8	rule -> rep	4	21	ins -> rule
5	rule -> para	5	18	rep -> ins
3	para -> rule	6	17	ins -> rep
3	rule -> ins	7	3	para -> rep
3	ins -> rule	8	3	para -> ins
2	rep -> ins	9	3	rep -> para
1	ins -> para	10	3	ins -> para
1	ins -> rep	11	1	para -> rule
0	para -> ins	12	1	rule -> para

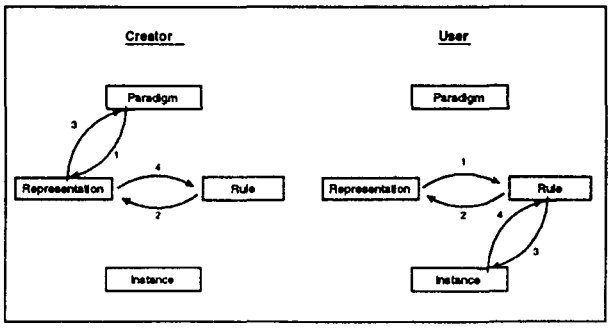

Figure 7. Top 4 Between Space Transition

Up to this point we have observed the difference in major supporting space in programming. Next we analyze the search behavior in the main problem spaces (representation and rule spaces) so as to explain the impacts of the differences between the two groups in major supporting spaces. Table 4 shows the search behavior within the main problem spaces in terms of the number and depth of backtracking in these spaces. The total number of backtracking is determined by counting the frequency of backtracking to an earlier rule or representation in the Problem Behavior Graphs (PBG) of the two spaces. And the depth of backtracking represents the total amount of episode steps the subjects had nullified while backtracking to an earlier episode. We can notice in Table 4 the differences between the two groups for each space. In the representation space, the creator group showed a more active search behavior compared to the user group. The creator group showed a total of 21 backtracking episodes with a total of 54 steps backtracked, while the user group only showed 12 backtracking episodes with 27 backtracking steps (number of backtracking: $t(4)=2.59$, $p<0.05$; depth of backtracking: $t(4)=2.84$, $p<0.05$). As for the search in the rule space the user group showed a total of 21 backtracking episodes with a total of 32 steps backtracked, while the creator group only showed 5 backtracking episodes with 5 backtracking steps (number of backtracking: $t(4)=1.45$, ns; depth of backtracking: $t(4)=1.60$, $p<0.1$).

Table. 4 Amount / Depth of Backtracking in the Rule and Representation Spaces

Group	Backtracking	Space	
		Representation	Rule
Creator	Amount	21	5
	Depth	54	5
User	Amount	12	21
	Depth	27	32

The above results suggest that the creator group performed a more active search in the representation space than in the rule space, while the user group performed a more active search in the rule space than in the representation space. The creators' prior experience with the paradigm space supported the search in the main problem spaces, especially in the representation space. This resulted in the top-down, straight forward and faster design process demonstrated by the creator group. However, the lack of interaction with the instance space resulted in the inability to refine the data structure and the algorithm in more detail. On the other hand, the general programming heuristic ("generate and test") exploited by the user group forced the bottom up rule induction and as a result a more active search in the rule space. Thus, the final program of the user group resulted in a very detailed and specific algorithm. Consequently, the added detail and specificity of the final program resulted in the longer time needed in designing the program. Thus one of the main spaces served as a major main space: for the creator group the representation space is the major main space, while it is the rule space that is the major main space for the user group.

In summary, for the creator group the representation space is the major problem space of the two main spaces (representation and rule spaces) of programming. That's because the paradigm space serves as the major supporting space to guide and help the search in the representation space. As for the user group, the rule space is the major main space, and that is because the instance space serves as the major supporting space by conducting

experiments and mental simulations to guide the development of the rules in the rule space.

5. Discussions

The results of the experimental study show that the difference in performance and final output between the methodology user and the methodology creator can be explained in terms of search behavior in the four problem spaces. The users applied the instance space as the major supporting space that worked closely with the rule space, which accounted for the long design time and detailed final output. On the other hand, the creators used the paradigm space as the major supporting space that worked closely with the representation space, which accounted for the straight forward top down design behavior and thus a more rapid program design and a less detailed program algorithm.

In this paper we propose a new theoretical view of programming. It regards programming as search in four problem spaces, which is similar to the process of scientific discovery. It provides an adequate explanation on why methodology creators are different from methodology users (programmers) in terms of their programming behavior and overall performance. However, the difference between creator-user is not the only phenomenon that the theoretical view can account for. In a related study, Kim, Lerch, and Simon (1995) focused on the rule and instance spaces with two problem isomorphs. The study found that the theory of programming as search in multiple problem spaces could explain the process when professional programmers were writing object-oriented programs from scratch. Also in a related study, Kim and Lerch (in press) focused on the representation space. The study found that the representation-space search makes the task of program understanding and reuse more difficult than similar programming tasks without representation-space search. And the current study provide further empirical validation for the theory of programming as search in multiple problem spaces by focusing on the paradigm space. This means that the theory is comprehensive to cover the various tasks in programming, and the four problem spaces are all needed to explain the complex behavior

in programming.

One practical implication from the study is the development of the system image in software development methodologies. The system image results from the physical structure that has been built for an artifice, including the documentation and instructions (Norman, 1986). The users understand the artifice based on how they interpret the system image. Thus, in many ways, the primary task of the creator is to construct an appropriate system image, realizing everything the user interacts will help to form that image. In a programming methodology, the system image can be text books, user manuals, and coding examples. Whatever the system image is, this paper suggests that the image is not rich enough to faithfully convey the design model to the users. This is especially important for the dominant methodologies such as object-oriented technology, because they become the industry standards to which everyone confirms their products. Therefore, incomplete system image may results in a standard with many loopholes, which in turn may produce industry wide confusion and chaos. We are currently developing a richer way to convey the way how creators approach the given problem in a system image so that programmers can fully understand the methodology creator's intentions. The system image is based on our theory of programming as search in four problem spaces.

Our experimental methodology limits our results in several ways. The experiment used very small programs only in OOP. It also used Tower of Hanoi isomorphs that might not be typical of the majority of information system problems. Furthermore, only three subjects in each group participated in the experiment, partly because creators were hard to find in the real world. However, complementing studies that examine software development in naturalistic settings with many subjects, we believe that experiments on smaller scales can give deep insights into the cognitive processes employed and can guide the design of appropriate tools.

Reference

[1] Adelson, B. and Soloway, E. (1985), "The Role of Domain Experience in Software Design", *IEEE Transactions on Software Engineering*,

SE-11(11), 1351-1360

[2] Baker, L. and Dunbar, K. (1996), "Problem Spaces in Real-World Science: What are They and How Do Scientists Search Them?", *Proceedings of the 18th Annual Conference of the Cognitive Science Society*, 21-22

[3] Booch, G. (1994), *Object-Oriented Analysis and Design with Applications*, 2nd Ed., Redwood City, CA: Benjamin / Cummings

[4] Burns, B. D. and Vollmeyer, R. (1996), "Goals and Problem Solving: Learning as Search of Three Spaces", *Proceedings of the 18th Annual Conference of the Cognitive Science Society*, 23-24

[5] Burns, B. D. (1996), "Building a Theory of Problem Solving and Scientific Discovery: How big is N in N-space Search?", *Proceedings of the 18th Annual Conference of the Cognitive Science Society*, 19-20

[6] Dahl, O. E., Dijkstra, and Hoare, C. (1972), *Structured Programming*, London: Academic Press

[7] Detienne, F. and Soloway, E. (1990), "An Empirically-derived Control Structure for the Process of Program Understanding", *International Journal of Man-Machine Studies*, 33, 323-342

[8] diSessa, A. A. (1986), "Models of Computation", In D. A. Norman and S. W. Draper (Eds.), *User Centered System Design: New Perspectives on Human-Computer Interaction*, Hillsdale, NJ: Erlbaum

[9] Dunbar, K. (1993), "How Scientists Really Reason: Scientific Reasoning in Real-World Laboratories", in R. J. Sterberg and J. Davidson (Eds.). *Mechanisms of Insight*, Cambridge, MA: MIT Press

[10] Ericsson, A. and Simon, H. A. (1993), Protocol Analysis (2nd ed..). Cambridge, MA: MIT Press

[11] Fisher, C. A. (1987), Advancing the Study of Programming with Computer-Aided Protocol Analysis, In Olson, G., Soloway, E. and Sheppard, S (Ed.), *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex

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

[13] Guindon, R., Curtis, B. and Krasner, H. (1987), *A Model of Cognitive Processes in Software Design: An Analysis of Breakdowns in Early Design Activities by individuals* (MCC Tech. Rep. No. STP-283-87).

Microelectronics and Computer Technology Corporation, Austin, Texas

[14] Hayes, J. R. and Simon, H. A. (1977), "The Understanding Process: Problem Isomorphs", in H. A. Simon (Ed.), *Models of Thought*, Vol. 1, New Haven: Yale University Press

[15] Hoc, J. M. and Nguyen-Xuan, A. (1990), "Language Semantics, Mental Models and Analogy", In J. M. Hoc, T. R. G. Green, Samurcay and D. J. Gilmore (Eds.), *Psychology of Programming*, 139-156, London: Academic Press

[16] Holland, J. H., Holyoak, K. J., Nisbett, R. E. and Thagard, P. R. (1986), *Induction: Processes of Inference, Learning and Discovery*, Cambridge, MA: MIT Press

[17] Jackson, M. (1983), *System Development*. Englewood Cliffs, NJ: Prentice Hall

[18] Jacobson, I. (1995), *The Object Advantage: Business Process Reengineering with Object Technology*, Reading, Massachusetts: Addison Wesley

[19] Jeffries, R., Turner, A., Polson, P. G. and Atwood, M. E. (1981), "The Processes Involved in Designing Software", In J. R. Anderson (Eds.), *Cognitive Skills and Their Acquisition*, 255-283, Hillsdale, NJ: Erlbaum

[20] Kant, E. and Newell, A. (1984), "Problem Solving Techniques for the Design of Algorithms", *Information Processing and Management*, 20, 97-118

[21] Kim, J. and Lerch, F. J. (1992), "The Cognitive Processes in Logical Design: Comparing Object-Oriented Design and Traditional Functional Decomposition Methodologies", *Proceedings of the CHI '92 Human Factors in Computing Systems*, ACM Press, New York, 489-498

[22] Kim, J., Lerch, F, J., and Simon, H. A. (1995), "Representation Construction and Rule Development in Object-Oriented Design", *ACM Transactions of Computer-Human Interaction*

[23] Kim, J. and Lerch, F. J. (in press), "Why is Programming (Sometimes) So Difficult?: Programming as Scientific Discovery in Multiple Problem Spaces", *Information Systems Research*

[24] Klahr, D. and Dunbar, K. (1988), "Dual Space Search During Scientific Reasoning", *Cognitive Science*, 12, 1-48

[25] Klahr, D., Fay, A. and Dunbar, K. (1993), "Heuristics for Scientific Ex-

perimentation: A Development Study", *Cognitive Psychology*, 25, 111- 146

[26] Kuhn, T. S. (1970), The structure of scientific revolutions, 2nd edition. Chicago: University of Chicago Press

[27] Langley, P., Simon, H., Bradshaw, G. and Zytkow, J. (1987), Scientific Discovery: *Computational Explorations of the Creative Processes*, Cambridge, MA: MIT Press,

[28] Letovsky, S. (1986), "Cognitive Processes in Program Comprehension", In E. Soloway and S. Iyengar (Eds.), *Empirical Studies of Programmers*, 58-79, Norwood, NJ: Ablex Publishing

[29] Mark, W. (1986), "Knowledge-Based Interface Design", In D. A. Norman and S. W. Draper (Eds.), *User Centered System Design: New Perspectives on Human-Computer Interaction*, Hillsdale, NJ: Erlbaum

[30] Mayer, R. E. (1981), "The Psychology of How Novices Learn Computer Programming", *Computing Surveys*, 13, 121-141

[31] Newell, A. and Simon, H. A. (1972), *Human Problem Solving*, *Englewood Cliffs*, NJ: Prentice-Hall

[32] Norman, D. A. (1986), "Cognitive Engineering", In D. A. Norman and S. W. Draper (Eds.), *User Centered System Design: New Perspectives on Human-Computer Interaction*, Hillsdale, NJ: Erlbaum

[33] Qin, Y. and Simon, H. A. (1990), "Laboratory Replication of Scientific Discovery Processes", *Cognitive Science*, 14, 281-312

[34] Rist, R. (1989), "Schema Creation in Programming", *Cognitive Science*, 13, 389-414

[35] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991), *Object-Oriented Modeling and Design*, Englewood Cliffs, NJ: Prentice-Hall

[36] Schunn, C. D. and Klahr, D. (1995), "A 4-Space Model of Scientific Discovery", *Proceedings of the 17th Annual Conference of the Cognitive Science Society*, 106-111

[37] Schunn, C. D. and Klahr, D. (1996), "The Problem of Problem Spaces: When and How to Go Beyond a 2-Space Model of Scientific Discovery", *Proceedings of the 18th Annual Conference of the Cognitive Science Society*, 25-26

[38] Simon, H. A. and Lea, G. (1974), "Problem Solving and Rule Induction:

A unified View", L. W. Gregg (Ed.), *Knowledge and Cognition*, Hillsdale, NJ: Erlbaum

[39] Simon, H. A. (1973), "The Structure of Ill Structures Problems", *Artificial Intelligence*, 4, 181-201

[40] Simon, H. A. (1981), "The Psychology of Thinking: Embedding Artifice in Nature", *The Science of the Artificial*, *Cambridge*, Massachusetts: MIT Press

[41] Van Someren, M. W., Barnard, Y. F. and Sandberg, J. A. C. (1994), *The Think Aloud Method: A Practical Guide to Modelling Cognitive Processes*, London: Academic Press

[42] Yourdon, E. (1989), *Modern Structured Analysis*, Englewood Cliffs, NJ: Prentice-Hall