Hayashi, Y., et al. (Eds.) (2017). Workshop Proceedings of the 25th International Conference on Computers in Education. New Zealand: Asia-Pacific Society for Computers in Education

Proposal of a Stepwise Support for Structural Understanding in Programming

Kento KOIKE[a*], Takahito TOMOTO[a] & Tsukasa HIRASHIMA[b]
[a]*Faculty of Engineering, Tokyo Polytechnic University, Japan*
[b]*Graduate School of Engineering, Hiroshima University, Japan*
*c1418030@st.t-kougei.ac.jp

Abstract: The importance and effectiveness of stepwise learning in programming education has long been advocated. However, there has been little concrete discussion of what kind of stepwise learning should be conducted. This research thus examines the properties of knowledge and comprehension processes in learning programming. Previous studies have also proposed "learning to read stepwise" and "learning to construct stepwise," and integrating these concepts is worthwhile. We thus propose their combination, "learning to understand stepwise." Although it was possible to show its usefulness in individual works, we could not investigate the effectiveness of the process as a whole.

Keywords: Programming education, stepwise understanding, structural understanding.

1. Introduction

In general, system design is chosen as necessary to achieve system requirements, and developers must be skilled in thinking about what kind of design will satisfy the given requirements. In other words, design skill in programming can be expressed as a problem-solving skill aimed at satisfying requirements.

In problem solving, a certain solution for a problem cannot be used for other problems. Therefore, solution itself are not reusable. However, when multiple problems are solved, a solution for a common term between problems may be obtained. For example, a learner who has learned how to code a selection sort or a bubble sort may notice that two values are always rearranged in the source code. Likewise, they might notice that rearrangement of these two values involves swapping them. Recognition of such meaningful commonalities urges the structuring of knowledge, improving skills at recognizing the source code used to achieve algorithmic steps. This structured knowledge allows partial application to other problems (i.e., code reuse). The concept of improving reusability by componentization and modularization is common in structured programming and is useful for structural design.

In recent studies of learning to program, there has been a focus on mastering syntactic knowledge (Egi & Takeuchi, 2007; Ishikawa, Matsuzawa & Sakai, 2014; Kanemune, Nakatani, Mitarai, Fukui & Kuno, 2004; Miura, Sugihara & Kunifuji, 2009) and understanding algorithms (Matsuda, Kashihara, Hirashima & Toyoda, 1995; Matsuzawa, Yasui, Sugiura & Sakai, 2014; Sugiura, Matsuzawa, Okada & Ohiwa, 2008; Yano, Fujisaki, Hirashima & Takeuchi 2001), but few studies focus on improving design capability, particularly the ability to design for reusability. Procedural design can be achieved by acquiring syntactic knowledge and understanding of algorithms, but it is not possible in this way alone to support structural design improvement, such as improving the reusability of parts by converting them into meaningful chunks of statements. To make a statement into a meaningful chunk, one must understand the relations between statements and recognize meaningful chunks of multiple statements. We grouped these skills and positioned them as structural understanding. Structuring of learners' prior knowledge is indispensable to acquiring structural understanding. We have thus focused on reconstruction of prior knowledge in programming (Koike & Tomoto, 2017), but there remains insufficient knowledge regarding the properties of prior knowledge and what needs to be concrete in its reconstruction.

This research thus examines a method for reconstructing knowledge for structural understanding, with the goal of improving design skills in programming.

2. Knowledge and Understanding in Programming

2.1. Distinction of Knowledge

According to Shneiderman and Mayer (1979), knowledge in programming can be broadly divided into semantic and syntactic knowledge. Semantic knowledge in programming is knowledge of algorithms and does not depend on knowing specific programming language. Examples include low-level concepts such as the operation performed in an assignment statement, the meaning of an array, the distinction of data types, exchanging the places of two values, and totaling the contents of an array. These can be expanded to higher-order concepts. In contrast, syntactic knowledge is more focused on detail, such as language-specific knowledge of the syntax of assignment or conditional statements or the names of library functions. It is easier to acquire new syntactic knowledge when existing semantic knowledge can be applied.

Kanamori, Tomoto, and Akakura (2013) also distinguish between knowledge types, proposing that a conversion process occurs between requirements, abstract operations (such as steps in a flow chart), and concrete operations (i.e., source code) (Fig. 1). Considering the classifications by Shneiderman and Mayer, the conversion process from requirements to abstract operations draws on semantic knowledge, and the conversion from abstract to concrete operations uses syntactic knowledge. This is also pointed out by Watanabe et al. Although the process of converting from requirements to abstract operations involves problem solving, the process of converting from abstract to concrete operations is a straightforward conversion of language.

As stated in Chapter 1, it is important to discover common semantic terms for structural understanding, so we think that reconstruction of semantic knowledge is necessary for learners.

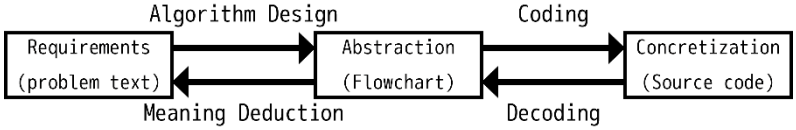

Figure 1. Process of programming. (Kanamori et al., 2013)

2.2. Comprehension Process of Programmers

In reconstructing semantic knowledge, it is essential to examine how programmers use semantic knowledge in the program comprehension process. Shneiderman and Mayer (1979) proposed a model of the cognitive process when programmers actually perform programming. Figure 2 shows the integration of knowledge discrimination into their model. Shneiderman and Mayer suppose that a programmer presented with a problem uses programming knowledge to internally construct a semantic structure that contains the multiple layers to be expressed in the program, and that this acts as a model. With the highest-level semantic structure, it is necessary to understand the requirements that the program is to satisfy. For example, the requirement may be to sort groups of input of fixed length, or to output the words most frequently occurring in an input. In addition, this higher-level understanding can be satisfied even when the low-level semantic structure is not fully understood. In the low-level semantic structure, there is an understanding of the chunks of source code needed to implement familiar algorithms. Similarly, understanding the low-level semantic structure does not provide understanding of the overall operation. When these semantic structures are understood, programmers do not memorize or understand programs on a statement-by-statement basis.

What is important here is that the high-level and low-level semantic structures are usually used independently. For example, (1) if the concept of value swapping is understood, then a learner can identify a swap included in the source code for a bubble sort, but this does not imply understanding that the whole source achieves swapping. The reverse is also true: (2) even when the

concept of sorting is understood, the programmer does not necessarily realize that swapping will be included. In the case of such a separated semantic structure, a programmer has the following problems. In the case of (1), it is impossible to recognize the requirements of the entire source code by combining low-level concepts. Moreover, in the case of (2), it is not possible to partition internal behavior from the requirements of the whole source code, which does not lead to an understanding of detailed behavior. From these problems, it is important to reconstruct the relations between low-level and high-level semantic structures to enable structural understanding. Therefore, we think that stepwise support in moving from a low-level semantic structure to a high-level one is important.

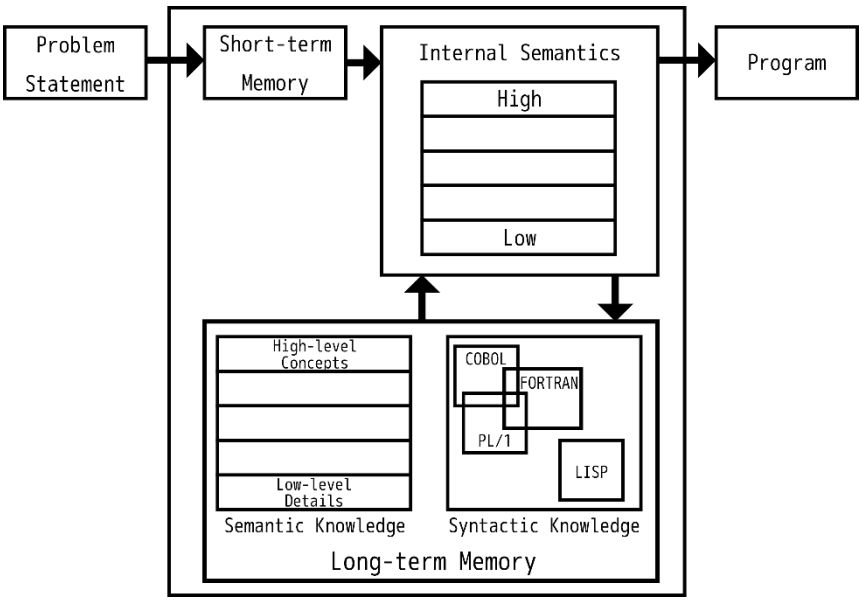

Figure 2. Comprehension process of programmers. (Shneiderman & Mayer, 1979)

3. Stepwise Learning

3.1. Stepwise Abstraction

We examine the stepwise learning method for reconstructing semantic knowledge. The learning method here refers to concrete learning methods for “meaning deduction” and “algorithm design” in Kanamori’s process of programming (Fig. 1) (Kanamori et al., 2013). Watanabe’s “stepwise abstraction” method has been proposed as an additional learning method for supporting the meaning deduction process (Watanabe et al., 2015). This stepwise abstraction has been proposed in reference to learning support using a stepwise refinement process. Shinkai and Sumitani (2008) referred to this as “stepwise subdividing the program from requirements.”

The stepwise abstraction process iteratively reads a given statement, gathers a set of parts considered to be a series of operations in the statement, and then incorporates these, acting as a stepwise reading support process. For example, in a program consisting of three assignments—(1) c = a, (2) a = b, and (3) b = c—although each statement is a simple assignment, considering the meaning of the three steps together reveals the concept of swapping. In this stepwise abstraction, low-level semantic structures are constructed by combining source code statements one by one, and the low-level semantic structure is converted to a high-level semantic structure by aggregating these low-level semantic structures. In this way, programmers can acquire new concepts by reading. However, there is no stepwise support for learning to read code. Without stepwise support, programmers will read code by using separate semantic structures (low-level and high-level semantic structures) as described in Section 2. As a result, they will be unable to construct a semantic structure in a stepwise fashion. Therefore, the authors think that stepwise presentation of abstraction will promote structural understanding among learners.

3.2. Expandable Modular Statements

The authors have attempted to support learning via expandable modular statements targeting the algorithm design process (Koike & Tomoto, 2017). In an expandable modular statement (Fig. 3), a learner first constructs parts in meaningful chunks from each statement in the program. New processes are added to the constructed parts, available parts are added, and parts are changed by partially modifying already constructed parts. Parts to be constructed, added, or modified here are presented to the learner. In the expandable modular statement, programs are constructed by combining statements and other parts with the presented parts. Therefore, at each step, a semantic structure one level higher than the lowest semantic structure is presented, and the learner is required to decompose the semantic structure into a lower-level semantic structure. For example, referring to Fig. 3, the construction "swap a and b" is first requested. When this construction is completed, "sort a and b" is then requested as the next semantic structure. In this way, semantic structures are presented in a stepwise fashion. By repeatedly changing these parts, it is possible to expand the semantic structures of the parts in steps, which encourages learners to understand the relations among parts. Expanding the parts step by step additionally trains learners to make large parts themselves, which we consider useful for actual design.

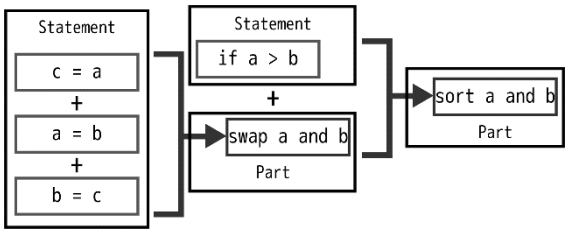

Figure 3. The expandable modular method. (Koike & Tomoto, 2017)

3.3. Proposed Learning Method

Stepwise abstraction (Watanabe et al., 2015) and expandable modular statements (Koike & Tomoto, 2017) are not exclusive approaches; we consider them to act as a series of techniques to facilitate the reconstruction of semantic knowledge in programming. They are important not only for stepwise reading but also for actually reading contents stepwise to construct the contents in steps and to understand the structure of large programs. Therefore, we propose learning to understand stepwise as a method of knowledge reconstruction in programming. The learning comprises stepwise abstraction for learning to read programs and expandable modular statements for learning to construct programs.

Learning to understand stepwise follows the procedure shown in Fig. 4. First, learners are presented the source code as a problem (similar to stepwise abstraction). Second, learners put together statements that they think are meaningful to the source code. Third, learners think about the request that the summarized part satisfies. Next, learners are required to rebuild the statements and parts in the order of the summarized source code for their requirements. In the process of abstraction, understanding of the program structure is facilitated by stepwise learning, and skill at reading programs can be improved. In the process of construction, learners reuse parts of the program as understood by stepwise learning.

We suggest that the program is understood in stages through repeating the learning process and that this contributes to the acquisition of structural understanding as described in Section 1. We use a preliminary experiment to verify the validity of the proposed method.

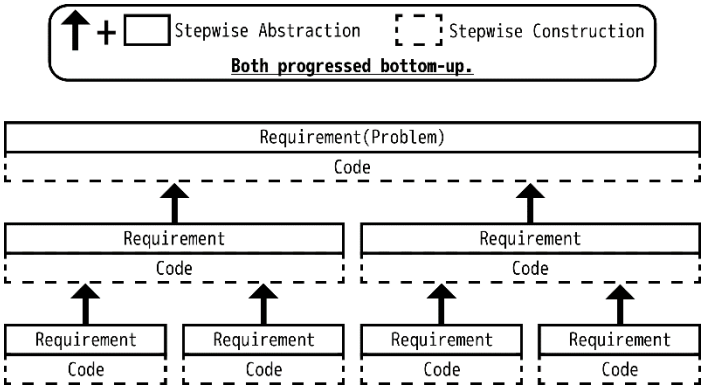

Figure 4. Learning to understand stepwise.

4. Preliminary Experiment

Preliminary experiments were conducted to investigate the effectiveness of the method proposed in Section 3.

4.1. Experimental Method

The subjects were nine students who had studied programming for three years in programming lectures and acquired basic concepts such as “for” and “if” statements, algorithms such as sorting, and use of functions. We conducted experiments after choosing four students for the control group and five students for the experimental group such that academic skills were evenly divided in the opinion of the experimenter. We divided the pre-test into a ten-minute “constructing task” and a five-minute “reading task.” The “constructing task” presents three problems describing the code requirements. In the “reading task,” the source code that was the answer to the problem presented in the “constructing task” is presented and the requirement is described. That is, the answer to the constructing task corresponds to the answer of the reading task. The learner moves to the learning task for 60 minutes after the pre-test. In the experimental group, we presented learning to read stepwise and learning to construct stepwise tasks for 2 out of 3 problems presented in the pre-test. In the control group, we presented learning to read stepwise and learning to construct as usual tasks for the same problems as the experimental group. After the learning task, we carried out a post-test of both groups using the same problem and response time as in the pre-test. After completion of the post-test, we administered a five-minute questionnaire regarding the four stages. All tests / tasks were presented in paper media.

4.2. Experimental Results and Consideration

Table 1 shows the scores for the constructing task of the pre- and post-tasks (full score is 3 points per problem) and the reading task (full score is 3 points per problem).

The results of the pre-test (see Table 1) show that subjects who have undergone third-year university lectures in both groups cannot construct novel programs by themselves. Also, the subjects cannot understand programs even if they read them, which shows that the statements cannot be understood as meaningful chunks. The results of the pre-test show that in both groups, a program once learned can be constructed similarly to how they were presented. Similar improvement of results was seen in both groups, so program reading skills were improved. Since the scores of both groups increased between the pre- and post-tests, learning to read stepwise appears effective for learning.

However, there was no difference in average score between the experimental and control groups. For that reason, we could not evaluate the effectiveness of learning to construct stepwise. The reason for this is that a ceiling effect is seen in the pre- and post-test scores, and it seems that a task of constructing a program in more detail was necessary. In addition, the time for learning task may be too short. Further consideration is needed regarding difficulty levels, time limits, and the method of distribution between experimental and control groups.

Both results are likely to be a memorization effect because 3 out of 2 problems in the test content are presented in the learning task and the same test was repeatedly used. However, since the experimental group had better memorize than the control group in reading learning, it is likely that there are several good factors in the learning task of the experimental group.

Table 1: Test score results

	Constructing task			Reading task		
	Pre-test	Post-test	Difference	Pre-test	Post-test	Difference
Experimental group	0.40	2.20	1.80	0.80	2.60	1.80
Control group	0.00	2.50	2.50	0.50	2.25	1.75

4.3. Questionnaire Result and Consideration

Table 2 shows the questionnaire results for both groups using a 4-point scale (4: Totally agree; 1: Do not agree at all), with the exceptions described below. Table 2 shows the question items and the average responses from the experimental and control groups.

"Learning to read stepwise" was accepted as a method by both groups, as reflected in the test results. However, is the groups differed strongly in their assessment of the construction teaching method, which was learning to construct stepwise for the experimental group and learning to construct as usual for the control group. Also, it is clear that there is a difference in feelings related to teaching materials between the groups, even in other questions. In addition, responses to questions that were posed to the experimental group only indicated that stepwise learning of construction was highly evaluated. With regard to the question "After stepwise reading, is stepwise construction or the usual construction most effective?" respondents preferred stepwise construction (coded as 4).

To summarize the questionnaire results, (1) a high evaluation was obtained for stepwise learning, and (2) after stepwise reading, the same results occurred for "stepwise construction" and "the usual construction." From this, we evaluated stepwise learning of construction.

Table 2: Questionnaire results

Question	Exp. Avg	Ctrl. Avg
Does learning with this material lead to structural understanding?	4.00	3.00
Does learning with this material lead to recognizing programs as parts?	3.80	3.25
Did you recognize the importance of structurally understanding through experiments?	3.80	3.25
Were the learning contents in the teaching materials effective in solving the post-test?	3.40	2.75
Is stepwise reading effective for understanding the program?	3.80	3.50
Is stepwise construction effective for understanding the program?	4.00	-
Is stepwise reading and construction together effective for understanding the program?	3.80	-
After stepwise reading, is stepwise or usual construction more effective?	3.40	-

5. Evaluation of Learning to Construct Stepwise

Although preliminary experiments on paper media in Section 4 suggested the effectiveness of learning to read stepwise, the evaluation of learning to construct stepwise did not have sufficient power. We therefore developed a system for stepwise learning of construction and performed evaluations using that system (Koike & Tomoto, 2017).

5.1. Learning Support System Overview

Figure 5 shows a screen of the system. In this system, each part is positioned as a block, each conventional programming statement is defined as a standard block, and parts extended by adding new statements and existing parts to a standard block are defined as an advanced block. The system is intended to support programming instruction that builds structural understanding of programs by setting advanced blocks to be constructed as goals and promotes learning by showing combinations of blocks that are easier to conceive before more difficult ones.

In the operation of this system, first, the minimum unit algorithm to be learned from the system at the center upper part of the screen is presented as an advanced block (ex. sort two variables). Second, to construct the advanced block, the learner adds the block (ex. advanced block of "swap", standard block of "if") from the block list on the left side of the screen to the work area in the center of the screen. Third, the learner aims to build an advanced block by freely changing the value, order and hierarchy of the added block in the work area. Finally, the student answers from the answer button on the upper right of the screen, and if it is wrong, it adjusts again in the work area. In this series of work learners can freely obtain hints and partial answers from the system.

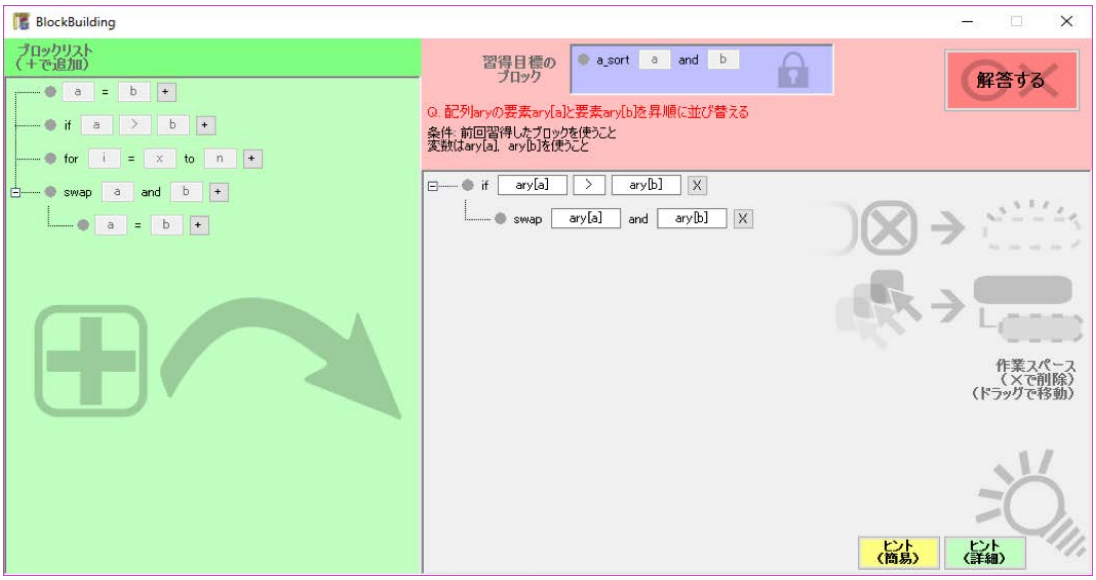

Figure 5. Screenshot of the “learning to construct stepwise” system. (Koike & Tomoto, 2017)

5.2. Experimental Method

The subjects were 17 students who had studied programming for three years in programming lectures and had already acquired basic concepts such as “for” and “if” statements, sorting algorithms, and use of functions. We administered a 15-minute pre-test that contained 8 questions that can be developed in series (including 4 basic problems and 4 learning-transfer problems) to measure fundamental programming and design skills. For the basic problems, we prepared problems in the range of learning tasks. For learning-transfer tasks, we prepared problems beyond the scope of learning materials but that could be solved by transferring learning. Also, the pre-tests called for structuring such as functionalization in each problem to the extent possible and instructed that respondents reuse earlier

answers in other problems. These evaluations made two types of evaluations: a simple score that evaluated whether the procedure is correct, and a structured score that evaluated functionalization and reuse in other problems. Based on the results, we divided participants into an experimental group (9 subjects) to learn using the proposed system and a control group (8 subjects) to learn the same problem via a paper medium. Groups were chosen such that the average score and distribution of scores were as similar as possible. The subjects first learned using the system or paper media after studying the material for 30 minutes. After that, we carried out a 15-minute post-test with the same contents and evaluation as the pre-test. After the post-test, we conducted a questionnaire using a 4-point evaluation on the learning methods and teaching materials.

5.3. Experimental Result and Consideration

Tables 3–7 show the scores of the pre- and post-tests of the experiment and control groups and the results of the test. All tests were evaluated at a significance level of 5%.

Table 3 shows the simple and structured score evaluations for the pre- and post-tests. The simple score data in Table 3 shows that the experimental group had a lower score than did the control group in in the pre-test, but a higher score than the control group in the post-test. Table 4 shows the results of ANOVA on the simple scores of the basic and learning-transfer problems in the pre- and post-tests. In Table 4, there were significant differences in test timing. The structured scores in Table 3 show that in the pre-test, although scores in the experimental group were higher than those in the control group, the average value of the experimental group in post-testing is higher than that in the control group, and the difference in test timing scores is larger than the control group in the post-test. To investigate between-group effects, Table 5 shows mean values for A–B interaction in the simple scores for the basic and metastasis problems. A significant trend is seen in the metastasis problem for the experimental group. Table 6 shows the results of ANOVA for the structured score of the basic and learning-transfer problems in the pre- and post-tests. In Table 6, significant differences were found for group, test timing, and A–B interactions. Table 7 shows the mean values for A–B interaction in the structured scores for the basic and learning-transfer problems. The table shows significant differences in the experimental group not only for basic problems in the learning range but also for the learning-transfer problems exceeding the learning range. Therefore, from the results of the pre- and post-tests, significant results were obtained for structuring in the experiment group, so the system can be considered useful for structural understanding of the program. Also, since there was a significant difference in the experimental group for test timing, the score in the experimental group improved more than that in the control group.

Table 3: Test score results

	Simple score			Structured score		
	Pre	Post	Difference	Pre	Post	Difference
Experimental group	1.44	4.67	3.22	0.33	3.22	2.89
Control group	1.50	3.88	2.38	0.25	0.63	0.38

Table 4: Results of ANOVA for the simple scores

	Basic problem			Learning-transfer problem		
Factors	SS	df	F	SS	df	F
Group	0.08	1	0.05	1.83	1	0.65
Error	24.86	15		42.05	15	
Test timing	23.14	1	39.17*	11.12	1	21.02*

Interaction	0.08	1	0.14	2.30	1	4.34
Repetitive error	8.86	15		7.94	15	

*: p (<0.05)

Table 5: Means for A–B interaction for the simple scores

	Basic problem			Learning-transfer problem		
Factors	SS	df	F	SS	df	F
Group (pre-)	0.00	1	0	0.01	1	0.01
Group (post-)	0.16	1	0.14	4.12	1	2.47
Test timing (experimental)	10.25	1	17.35*	11.76	1	22.23*
Test timing (control)	12.97	1	21.96*	1.65	1	3.13

*: p (<0.05)

Table 6: Results of ANOVA for the structured scores

	Basic problem			Learning-transfer problem		
Factors	SS	df	F	SS	df	F
Group	3.46	1	6.67*	4.17	1	4.56*
Error	7.78	15		13.72	15	
Test timing	6.90	1	17.67*	4.50	1	10.49*
Interaction	3.61	1	9.24*	3.09	1	7.20*
Repetitive error	5.86	15		6.44	15	

*: p (<0.05)

Table 7: Means for A–B interaction for the structured scores

	Basic problem			Learning-transfer problem		
Factors	SS	df	F	SS	df	F
Group (pre)	0.00	1	0.00	0.04	1	0.06
Group (post)	7.07	1	15.54*	7.22	1	10.75*
Test timing (experimental)	10.25	1	26.23*	7.53	1	17.54*
Test timing (control)	0.26	1	0.68	0.07	1	0.15

*: p (<0.05)

5.4. Questionnaire Results and Consideration

Tables 8 and 9 show the results of the questionnaires given to both groups using a 4-point scale (4: Strongly agree; 1: Do not agree at all) and the results as analyzed by a chi-squared test to show differences between the distribution of both groups.

In the chi-squared test, scores of 3 or 4 were evaluated as positive, and scores of 1 or 2 as negative, at a significance level of 5%. Table 8 shows the results of a questionnaire inquiring as to the importance of each question item. In these results, differences in distribution did not appear in either the experimental group or the control group for each question item, and there was no notable difference for each skill. However, as Table 9 shows, there were differences in the results of a questionnaire asking whether each of the learning materials used for the experimental and control groups would improve each skill, indicating that the learning materials for the experiment group were more useful. There was no score less than that for the learning material of the control group.

Table 8: Results from a questionnaire on importance of skill

Questions	Exp. Avg	Ctrl. Avg
Skill in recognizing chunks of programs as parts	3.78	3.50
Skill in understanding relations between programs	3.78	3.88
Skill in increasing reusability for each program	3.78	3.75
Skill in structurally understanding programs	3.78	3.88

*: p (<0.05)

Table 9: Results from a questionnaire related to learning materials

Questions	Exp. Avg	Ctrl. Avg
Did the material lead to understanding programming?	3.00*	2.25
Did the material lead to understanding relations between programs?	2.89*	1.63

*: p (<0.05)

6. Discussion and Future Works

This research showed that structural understanding is important for understanding relations between statements and for recognizing meaningful clusters of multiple statements in order to perform structural design in designing system development. Also, for structural understanding, we considered it necessary to reconstruct prior knowledge. Prior knowledge is divided into semantic and syntactic knowledge, and we consider that reconstructing relations in semantic knowledge leads to structural understanding. It is thought that low-level and high-level semantic structures are separate in the understanding process of programmers, and that it is necessary to reconstruct semantic structures with stepwise support. From previous research, stepwise abstraction has been proposed for stepwise learning of reading, and expandable modular statements have been proposed as a tool for stepwise learning of construction. Contributing to the reconstruction of semantic structures by integrating these methods is considered as learning to understand stepwise. Preliminary experiments were conducted to verify this method. These experiments showed an effect for learning to read stepwise, but no effect for learning to construct stepwise. For that reason, we evaluated a system for stepwise learning of construction and obtained significant results, indicating that stepwise learning is effective for both reading and construction. We therefore suggest that learning to understand stepwise is possible.

Future works will include system development using the proposed method and devising experimental methods to verify its learning effects.

Acknowledgements

This study was partially funded by Grants-in-Aid for Scientific Research (C) (15K00492) and (B) (K26280127) in Japan.

References

Egi, T., & Takeuchi, A. (2007). An Analysis on a Learning Support System for Tracing in Beginner's Debugging. In Proceedings of the 2007 Conference on Supporting Learning Flow Through Integrative Technologies (pp. 509–516). Amsterdam, The Netherlands, The Netherlands: IOS Press. Retrieved from http://dl.acm.org/citation.cfm?id=1565478.1565573

Ishikawa, Y., Matsuzawa, Y. & Sakai, S. (2014). A Prototype of Workbench for Understanding the Concept of Polymorphism in Object-Oriented Language. *Transactions of Japanese Society for Information and Systems in Education*, *31*(2), 208–213. http://doi.org/10.14926/jsise.31.208 (in Japanese)

Kanamori, H., Tomoto, T. & Akakura, T. (2013). Development of a Computer Programming Learning Support System Based on Reading Computer Program. (S. Yamamoto, Ed.) *Human Interface and the Management of Information. Information and Interaction for Learning, Culture, Collaboration and Business*. Springer Berlin Heidelberg. http://doi.org/10.1007/978-3-642-39226-9_8

Kanemune, S., Nakatani, T., Mitarai, R., Fukui, S. & Kuno, Y. (2004). Design and Implementation of Object Sharing for Dolittle Language. *Journal of Information Processing*, *45*(5), 81. Retrieved from http://ci.nii.ac.jp/naid/110002712352/en/ (in Japanese)

Koike, K. & Tomoto, T. (2017). Proposal of Granularity Expand Method in Parts for Structural Understanding on Programming and Development Leaning Support System. *Japanese Society for Information and Systems in Education - Research Report*, (8), 211–221. (in Japanese)

Matsuda, N., Kashihara, A., Hirashima, T., & Toyoda, J. (1995). An Instructional System for Constructing Algorithms in Recursive Programming. Advances in Human Factors/Ergonomics, 20, 889–894. http://doi.org/http://dx.doi.org/10.1016/S0921-2647(06)80140-1

Matsuzawa, Y., Yasui, H., Sugiura, M. & Sakai, S. (2014). Seamless Language Migration in Introductory Programming Education through Mutual Language Translation between Visual and Java. *Journal of Information Processing*, *55*(1), 57–71. Retrieved from http://ci.nii.ac.jp/naid/110009660234/en/ (in Japanese)

Miura, M., Sugihara, T. & Kunifuji, S. (2009). A Workbench for Understanding Relationship between Variable and Data in Object Oriented Programming Language. *Journal of Information Processing*, *50*(10), 2396–2408. Retrieved from http://ci.nii.ac.jp/naid/110007970523/en/ (in Japanese)

Shinkai, J. & Sumitani, S. (2008). Development of Programming Learning Support System Emphasizing Process. *Japan Journal of Educational Technology*, *31*, 45–48. http://doi.org/10.15077/jjet.KJ00004964344 (in Japanese)

Shneiderman, B. & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming*, *8*(3), 219–238.

Sugiura, M., Matsuzawa, Y., Okada, T. & Ohiwa, H. (2008). Introductory Education for Algorithm Construction: Understanding Concepts of Algorithm through Unplugged Work and Its Effects. *Journal of Information Processing*, *49*(10), 3409–3427. Retrieved from http://ci.nii.ac.jp/naid/110007970228/en/ (in Japanese)

Watanabe, K., Tomoto, T. & Akakura, T. (2015). Development of a Learning Support System for Reading Source Code by Stepwise Abstraction. In *International Conference on Human Interface and the Management of Information* (pp. 387–394). Springer.

Yano, M., Fujisaki, K., Hirashima, T., & Takeuchi, A. (2001). Prolog Learnig Assistant System with Structured Diagram. Transactions of Japanese Society for Information and Systems in Education, 18(3), 319–327. Retrieved from http://ci.nii.ac.jp/naid/10007406289/en/