Wong, L.-H. et al. (Eds.) (2013). Proceedings of the 21st International Conference on Computers in Education. Indonesia: Asia-Pacific Society for Computers in Education

enPoly: Workbench for Understanding Polymorphism in Strong Typed Object-Oriented Language

Yoshiaki MATSUZAWA[a*], Yukiko ISHIKAWA[a] & Sanshiro SAKAI[a]
[a]*Faculty of Informatics, Shizuoka University, Japan*
*matsuzawa@inf.shizuoka.ac.jp

Abstract: Polymorphism is a crucial concept in creating programs using object-oriented languages. Although understanding polymorphism requires learners to capture dynamic (behavioral) aspects of objects, current tools provide only static (structural) aspects. To address this limitation, we developed a workbench tool called “enPoly” that is a redesign of the “Anchor Garden” proposed by Miura et al. (2009). Our tool has the following two key features: (1) it shows learners behavioral aspects of objects in an animated fashion and (2) it shows the distinction between the definition and implementation of methods through visualization, thereby promoting the understanding of the Interface concept in Java. An experimental study was conducted in which 12 students were divided into two six-student groups, one of which was the control group. All six students in the experimental group succeeded in solving the given programming task using polymorphism even though they did not succeed in their initial state. In contrast, the six students in the control group made no improvements.

Keywords: Polymorphism, object-oriented programming, behavior, animation, tool, scaffolding

1. Introduction

Polymorphism is a fundamental concept underlying object-oriented (OO) programming languages. Understanding polymorphism is a strong foundation for computer science and engineering. Applications of polymorphism promote robust and extensible programs (Meyer, 1990).

In this study, we use the term polymorphism to describe a system in which behavioral results of message passing can be changed by instances. We limit the application of our claim within the Java programming language, which is a strongly typed language. Therefore, we view polymorphism as the behavioral change using common interfaces and implementations. Figure 1 illustrates polymorphism in Java, which gives a comparison of the use of polymorphism to an equivalent if statement. The code in the right-hand box of the figure is an approach using an “if statement” in which the program first tests what type of instance has been encountered, and then selects an appropriate branch. The code in the left-hand box of the figure is a solution using polymorphism in which the program automatically selects an appropriate branch by simply calling the greeting() method.

by Polymorphism

```
Person person = new Japanese();
person.greeting();
```

by If Statement

```
Person person = new Japanese();
if(person instanceof Japanese){
  System.out.println(`Konnichiwa`);
} else if (person instanceof American){
  System.out.println(`Hello`);
}
```

Figure 1. Illustrating polymorphism by comparing its implementation to that of an if statement.

Although polymorphism is useful to create quality object-oriented programs, it can be difficult to master the concept and properly apply it. Ross (2005) claimed that the information supporting polymorphism in textbooks is limited when compared to other OO concepts such as inheritance and encapsulation. Liberman (2011) reported that the concept of inheritance and polymorphism are misunderstood by teachers who are trained in non-object-oriented languages, primarily in teacher training education. Our experience has shown that students who have learned programming skills via C failed to apply polymorphism given the problem illustrated in Figure 1. All the students came up with the if statement approach rather than applying polymorphism even when we asked them to apply polymorphism and students had already learned the concept.

In this study, we therefore present a tool to support the learning of polymorphism. Our software provides students a visualization tool for the world model based on the OO concept in which polymorphic aspects are highlighted via animation and clear representations of the interface.

2. Related Work

Miura et al. (2009) proposed a workbench tool to support students learning OO concepts in programming, especially type, objects, and pointers. They defined "workbench" as a means not only to visualize models representing the concepts to help students understand but also to provide a high-level interactive user interface for learners such that they can directly touch the models. Miura et al. claimed that the users of Anchor Garden (AG) can actively learn via interactions between users and a workbench, in which trial and error leads students to discover how a visualized world can be changed by their operations. Our research builds upon this workbench model. As AG supported only static structural models, we expanded it to behavioral models in order to support learning the polymorphism concept.

We also reviewed other learning environments that could be considered a support to learning polymorphism. First, Squeak (Dan et al., 1997) has an object inspector with which users can not only watch variables within objects but also interact with objects by directly sending messages from the user interface; however, the system does not provide any visual models of message passing. Smalltalk is one of the weakly typed languages in which polymorphism would be implicitly and naturally implemented, and it may prevent students from explicitly learning polymorphism. Although the choice of languages to best teach students in programming education is controversial (e.g., the procedure-first or object-first controversy (Lister, 2004; Lister, et al., 2006)), we should at least help engineers and computer scientists who have mastered strongly typed languages (e.g., Java or C++) and are being driven by requirements from industry.

Alkazemi et al. (2012) proposed a method to learn polymorphism using BlueJ, which is a developmental environment presented by Kolling (2003) designed for object-oriented programming education. The system provides functions to edit class diagrams and object models in support of writing source code in Java; however, the system supports only static structural models—i.e., behavioral models are not supported. Tarsem et al. (2006) proposed an approach to teach polymorphism in the game programming context. We agree that the theme is attractive and strongly motivates learners; however, the proposed environment does not provide any functions to directly support learning polymorphism.

Our research could easily be categorized as a visual debugging approach for novice-level OO programming education. Many tools have already been proposed in this field, including OOP-Anim (Esteves, et al., 2003), ALICE (Cooper, et al., 2003), JGrasp (Hendrix, et al., 2004), Jeliot3 (Moreno, et al., 2004), JIVE (Gestwicki, et al., 2005), and jList (Fossati, et al., 2009). OOP-Anim is designed to learn the concept of the class and instance relationship. ALICE is a scripting system to develop algorithm construction skills via the creation of three-dimensional animations. JGrasp and jList are both designed to scaffold the understanding of data structures and algorithms by visualizing models that represents those concepts. Although these tools are available for OO languages, the design of the tools is not for understanding OO concepts. JIVE has functions to support learning by visualizing the object system; the tool can visualize any user program written in Java by executing it with the Java Debugger Interface; however, visualization of the behavioral aspects is limited to static representations using Sequence Diagrams, which is considered less intuitive for novice learners. Jeliot3 provides a descriptive visualization of the program interpretation and execution; however, the mechanism of calling methods is not visualized; therefore, it is limited for learning polymorphism. As Ben-Ari et al. (2011) summarized regarding visual debugging, a descriptive visualization approach is limited because learners cannot focus on their misunderstandings. Another problem with the visual debugging approach is that these tools do not allow learners to operate the models to further their understanding. In additional, as these visual debugger approaches require completed source code for visualization, it is not appropriate for the learner who cannot yet write any full programs.

3. enPoly: Polymorphism Visualization System

3.1 Purpose and Targets

In this paper, we present a system called enPoly that allows users to more easily learn polymorphism. The origin of the name is that the system enables learners to understand Polymorphism. The target audiences of the system are as follows: (1) novice-level software designers and programmers who have basic programming skills in strongly typed languages, such as Java and C++; and (2) learners who have learned polymorphism at least at a basic level (e.g., via lecture or book) but who have limited practical understanding and cannot apply the concept successfully.

3.2 Functions

Our enPoly system was built based on AG (Miura et al., 2009), which provides fundamental functions to visualize a static object system. We expanded AG by adding new functions, as illustrated in Figure 2. The left-hand image of the figure shows the original AG and the right-hand image shows enPoly. Both systems have basic functions with which users can create and operate an object system, including variables, although they are limited in the prepared class type. Our enPoly system has the following three additional features: (1) polymorphic animations; (2) interfaces and inheritance; and (3) anchor representation. Further descriptions of these features are included in the subsections below.

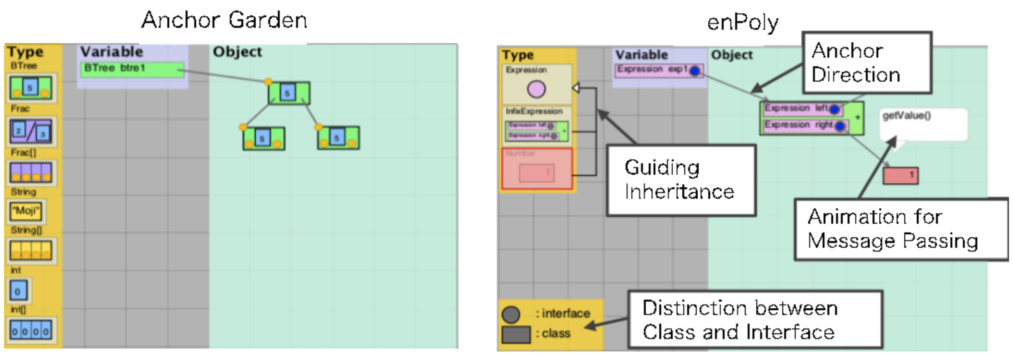
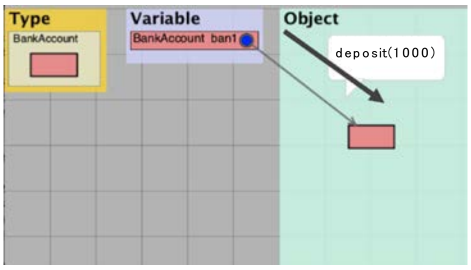


Figure 2. Characteristics of enPoly in comparison with Anchor Garden.

3.2.1 Polymorphic Animations

As visualizations of message passing are considered crucial for understanding dynamic and polymorphic aspects of an object-oriented system, we propose an animated representation for message passing in enPoly. Our system uses a balloon representation, which is a popular message model in manga, for visualizing messages. A balloon flows on the link connected between a sender object and a receiver. For method calls, a balloon flows from a sender object to a receiver, and then flows in the opposite direction for a return value. For example, the left-hand image of Figure 3 shows calling method deposit() from the variable ban1 to a BankAccount object. Users can send messages by right-clicking a variable and selecting the type of message to send.

Another feature for visualizing polymorphic behavior is the use of a balloon as a result of method execution. When a receiver object receives a message, the object puts a result balloon on the upper right-hand corner of the object. For example, the right-hand image of Figure 3 shows calling method greeting() from variables per1 and per2 to objects whose classes are inherited from the Person class (i.e., Japanese and American). Given these visual cues, users learn polymorphism by observing the differences in behaviors when these users send the same message to two different types of objects. In the given example, a Japanese object says “Konnichiwa,” which is a greeting in Japanese, whereas an American object says “Hello.”



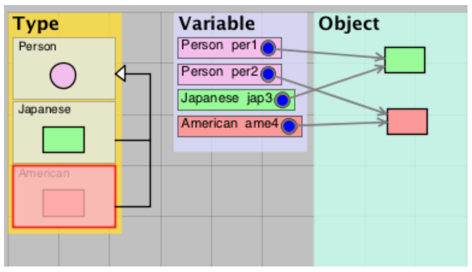


Figure 3. Screenshots of enPoly: BankAccount example (left) and Greeting example (right).

3.2.2 *Interfaces and Inheritance*

To further promote the understanding of the Java interface and inheritance (including implementation of the interface in Java), we incorporated guides for these concepts into enPoly. First, we designed class and interface representations using different figures by which users can easily distinguish between them. More specifically, classes are represented as rectangles and interfaces are represented as ellipses. Because we cannot create objects from interfaces, the warning message “cannot generate any interface’s object” would be shown in the object field if users attempt to create an object.

Second, the relationships of inheritance between classes are indicated in the Unified Modeling Language notation in our system. This therefore requires users to make a link from a variable typed as a super interface to an object whose class is a concrete implemented subclass of the interface. Third, when users try to link an object to a variable by dragging an anchor tab, enPoly guides the object that can be assigned to the variable by changing the color to red if possible; otherwise, blue is used.

3.2.3 *Anchor Representation*

In AG, the anchor tab is a draggable point that is associated with each variable. Miura et al. (2009) claimed that users can easily make a link connection by dragging an anchor tab from a variable to a target object; however, we redesigned this functionality so that our anchor is associated with an object. This design is important for learning polymorphism because not only can the system easily guide learners in the direction of message passing but it can also provide the type-checking function, as described in the previous subsection.

Using enPoly’s pointer representation, a recursive object structure can certainly be modeled. For example, Figure 4 shows the object model of numerical formula expressions. The model in the figure represents “6 /(2 + 1),” which is written in infix order. The interface Expression is defined as a super-interface of InfixExpression and Number. InfixExpression has two expressions as left and right, which are defined as Expression variables. Therefore, the InfixExpression object “/” can have the “6” Number object and the “2 + 1” InfixExpression object. In this case, a message the parent “/” object received would be recursively passed to the child objects.

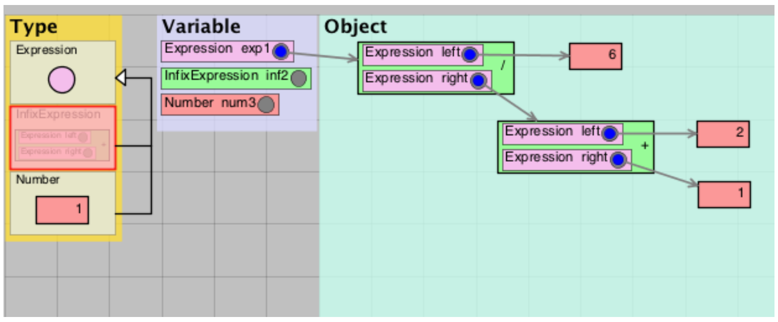


Figure 4. A screenshot of enPoly illustrating a recursive example of a calculator.

4. Experimental Study

We conducted an experimental study in which subjects were given programming tasks designed to assess the level of understanding of polymorphism.

4.1 *Task Description*

We designed the programming tasks to assess the level of understanding of polymorphism by ensuring that the task could not be solved without polymorphism. An example of the task is shown below.

Task 3: The given program (Calculator.java) is a part of the calculation program. Students have to complete the program by adding some appropriate classes, but they cannot edit the given program. A part of the given program is shown in Figure 5, as well as the class diagram for the task, although the class diagram is not given to students.

```
...
Stack<Expression> stack = new Stack<Expression>();
for (String token : tokens) {
    if (isOperator(token)) {
        Expression right = stack.pop();
        Expression left = stack.pop();
        stack.push(new InfixExpression(token, left, right));
    } else {
        int number = Integer.parseInt(token);
        stack.push(new Number(number));
    }
}
...
```

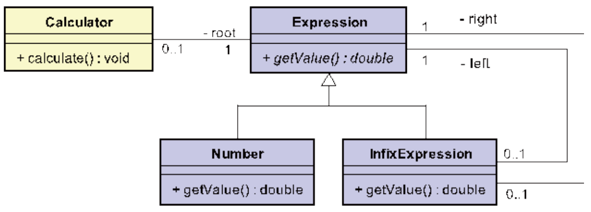


Figure 5. Task 3: An example task used in our experimental study.

4.2 *Hypothesis*

The hypothesis of the study is that "enPoly can guide students who cannot solve the programming tasks in their initial state to understand polymorphism and apply it to complete the task." In the experimental (tool-assisted) group, students were given opportunities to use enPoly in their own way. The process we assumed the students would follow via enPoly is described as follows:
(1) Create static object structure
(1-a) Students create variables in enPoly for all classes shown, as well as create instances
(1-b) Students create anchors by connecting objects and variables and examining what type of objects can be assigned in a variable
(2) Observe dynamic object behaviors
(2-a) Students observe behaviors of the object system by calling methods (repeatedly)
(2-b) Students examine differences of behaviors by calling the same method for objects of different types (i.e., observing polymorphic behaviors)
Hence, this experimental study examines whether the proposed functions described in Section 3 above are workable.

4.3 *Experimental Study Plan*

The experimental study was conducted with 12 master's and undergraduate students who have programming skills in C and Java. They were assigned to one of the following two groups: (1) the experimental group in which students engaged in a task with enPoly and (2) the control group in which students did the same task without the tool. Although we tried to randomly assign students to each group, all three master's students who were already skilled were included in the control group.

We prepared three tasks at three levels, as summarized in Table 1.

Table 1: Tasks used in the experimental study.

	Task 1	Task 2	Task 3
Example	Greeting	1 Operator Calculator	Multi Operator Calculator
Including Components	method call	return value	return value; recursion
Estimated Time	15 min	20 min	30 min
Estimated Lines of Code	10	20	20

The procedure of the experimental study for each subject is shown in Figure 6. All subjects were given three tasks in the same order from lower to higher level. First, a subject tried to solve the first task without a tool in both groups in order to evaluate the skills for the initial state. Students who successfully completed the task went to the next task, and those who failed the task retried the same task after learning it. The learning process is different for the two groups. The enPoly system was given to subjects in the experimental group, whereas a lecture by a teaching assistant was given to the control group. If the subject completed the task in retrial, the subject went to the next task; otherwise, the subject exited the experiment.

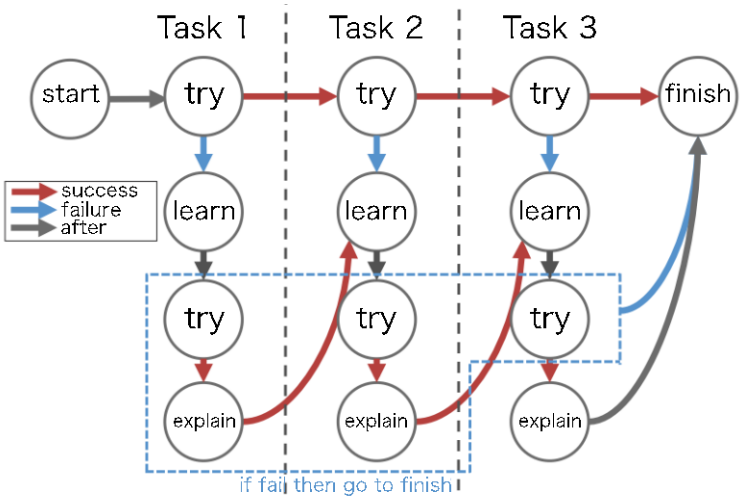


Figure 6. Procedure of the experimental study for each subject.

5. Results

5.1 Results

Results for the experimental group are summarized in Table 2; results for the control group are summarized in Table 3.Cells in the "Subject" column show the IDs of the subjects. The string below a student ID shows the grade of university student. For example, M1 means 1st grade master's student and B4 means 4th grade bachelor's student.

Cells in the "Task" column show the results using the following notation:

✓ – Student completed the task
△ – Student failed the task, but completed creating the structure of the objects with enPoly
× – Student failed the task
— – Not applicable

The "No Ast" column indicates that students were not given tool assistance, and "enPoly" indicates that students were assisted by enPoly. Numbers below the results show the time taken for the task in minutes, and numbers inside parentheses show the time taken to use enPoly.

The "Level" column shows how each student's understanding level changed. The left-hand number in the "Level" cell shows the level in the initial state for the student and the right-hand number shows the level after finishing. For example, "1 → 3" means that the student was at level 1 in the initial state and changed to level 3 after learning and completing the tasks. Furthermore, the level of a student's understanding was identified by which level of task the student completed. For example, the student who failed Task 1 was identified as level 0, and the student who completed Task 2 but failed Task 3 was identified as level 2.

Table 2: Results for the experimental group (with enPoly).

Subject	Task 1		Task 2		Task 3		Level
	No Ast	enPoly	No Ast	enPoly	No Ast	enPoly	
A (B3)	✓ 15	—	× 20	✓ 25(10)	—	✓ 35(10)	1 → 3
B (B3)	× 15	✓ 15(8)	—	✓ 30(5)	—	✓ 25(10)	0 → 3
C (B3)	× 15	✓ 5(2)	—	✓ 20(7)	—	✓ 25(6)	0 → 3
D (B3)	× 15	✓ 15(5)	—	✓ 30(13)	—	✓ 25(12)	0 → 3
E (B4)	× 15	✓ 15(6)	—	✓ 30(8)	—	✓ 25(13)	0 → 3
F (B4)	× 15	✓ 20(10)	—	△ 30(18)	—	—	0 → 1.5

Table 3: Results for the control group (without enPoly).

Subject	Task 1		Task 2		Task 3		Level
	No Ast	enPoly	No Ast	enPoly	No Ast	enPoly	
G (M2)	✓ 5	—	—	—	✓ 35	—	3 → 3
H (M2)	✓ 5	—	—	—	✓ 30	—	3 → 3
I (M2)	✓ 10	—	—	—	✓ 75	—	3 → 3
J (B4)	× 20	—	—	—	× 75	—	0 →0
K (B4)	× 30	—	× 40	—	× 60	—	0 → 0
L (B4)	× 20	—	× 35	—	× 75	—	0 → 0

5.2 Qualitative Analysis of the Learning Process

The screen that the subjects were operating was recorded via a screen capture application and was analyzed qualitatively to discover system usage patterns in performing the tasks. In this section, we present two narratives of the learning process involving enPoly, tagging the fragments of the processes with 1-a, 1-b, 2-a, and 2-b, as described in Section 4.2 above.

5.2.1 Case I: Task 1

In this case, we describe the learning process for subject E on Task 1. The subject was creating the object structure in enPoly. At first, the subject created three variables for Person, Japanese, and American (1-a). Then, the subject tried to create instances, but then realized that Person could not be instantiated because it is an interface (1-a). Subsequently, the subject created a Japanese instance and assigned it to the Japanese variable first (1-b). The subject tried to assign the instance to both Person and American variables, and then realized that the instance (of Japanese) can be assigned to the Person variable (1-b). The same examination was done for an American instance (1-b). Finally, the subject looked back to the program, which is given in the task, and found that the visualized structure is the same as what the task requires (1-b). The status of enPoly for the completion of these operations is shown in the left-hand image of Figure 7.

Subsequently, the student observed the behavioral aspects for the created object structure. At first, the subject examined the call methods for all variables (2-a). In that process, the same method was repeatedly called for more than two instances (2-a). Through this process, the subject realized differences in the object behaviors between the Japanese and American instances when he sent the same message to the two instances (2-b). As a result, the subject understood the location where the method should be implemented (2-b), which led to completing the task.

We observed a similar process, as described above, for five of the six subjects. The final subject had almost the same process, although the process of (1-b) was omitted.

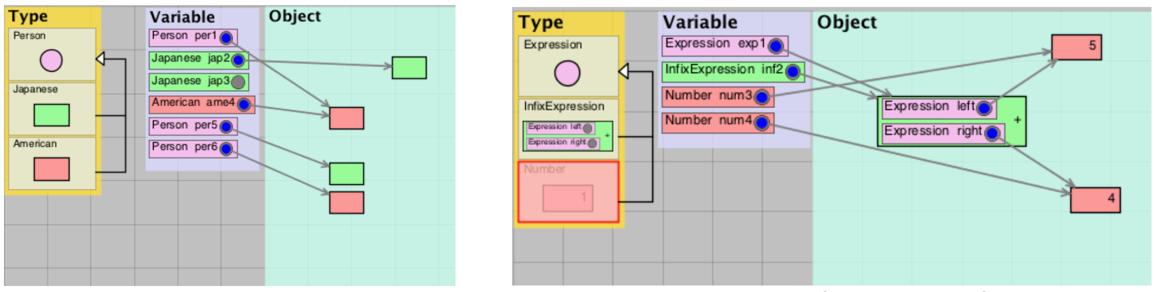


Figure 7. Screenshots created by students in our experimental study for Task 1 (left) and Task 3 (right).

5.2.2 *Case II: Task 3*

In this case, we describe the learning process for subject A on Task 3, although we omit the descriptions of the object-creation processes (1-a and 1-b) because the processes were quite similar to those of Task 1. The status of enPoly after the process is shown in the right-hand image of Figure 7.

After creating the object structure, the student observed the behavioral aspects of the created object structure. The subject examined the call methods of getValue() twice from the variable of Expression (2-a). The subject examined calling the same method from an instance of InfixExpression as well (2-a). Next, the subject realized that the differences in the callable method set between Expression and InfixExpression (2-b). The subject then repeatedly examined the call method of getValue() by changing the value of the Number instance or the operator of the InfixExpression instance. As a result of this exploration, the student succeeded in completing the task.

We observed a similar process, as described above, for four of the five subjects who attempted Task 3. The fifth subject had almost the same process, although the process of (1-b) was omitted.

6. Discussion

6.1 *Evaluation of enPoly*

As shown in the results summarized in Tables 2 and 3 above, the understanding levels of the subjects in the experimental group changed from 1.67 to 2.75, on average, whereas in the control group, all subjects had no change in their level of understanding. We believe this difference is significant; that is, enPoly supported students in effectively learning polymorphism.

This claim is reinforced by the results of the qualitative analysis of the learning process with enPoly, as described in section 5.2 above, in which we observed that students could use enPoly in our hypothesized procedure. We observed students exploring behavioral actions with enPoly by repeatedly calling methods (to trigger a message passing animation), and such actions were taken even though the teachers did not require or mention it. We think such autonomous student exploration promoted a clear understanding of the polymorphic system, leading to the result that almost all students succeeded in completing the task after using enPoly, which indicates that one of enPoly's features, namely polymorphic animations, worked effectively.

The results of our qualitative analysis showed that enPoly's interfaces and inheritance features also effectively worked. Students who had no understanding of the interface concept could not understand the fact that a subclass' instance could be assigned to a variable defined as superclass. The enPoly system succeeded to scaffold such students by supporting their exploration of what types of instances could be assigned to the variable. Nonetheless, we did not assess whether this understanding rose to the full conceptual level of the interface, but we could at least provide students a successful case to form initial understandings of the concept.

6.2 Limitations

A critical problem in our experimental study was the difference of understanding levels between the experimental and control groups. Three master's students who had more experience than the undergraduates were included in the control group; however, the deviation should be affected as negative bias. At the very least, we can find the difference of the results for the students initially at the same level between the two groups. Hence, our claims are still effective, though limited by a few subjects.

The fundamental feature of our enPoly approach lies in its animated visualization of the dynamic aspects of the modeled system. Although we could show the effectiveness of this approach and its acceptability for learners, results do not indicate that our approach has an advantage over the static visualization approach for illustrating dynamic aspects, such as using sequence diagrams. It may be an advantage for learners to explore details at their own speed.

Finally, the differences between the experimental group and control group have been assessed even if the experimental group subjects were scaffolded by enPoly. Our final goal should assess whether the experimental group students were able to complete the task without enPoly; however, we still believe enPoly provided the environment necessary to understand the polymorphism concept as the qualitative analysis of their learning process indicated.

Acknowledgements
A part of this work was supported by JSPS KAKENHI Grant Number 25730203.

References

Alkazemi, B. Y., & Grami, G.M. (2012). Utilizing BlueJ to Teach Polymorphism in an Advanced Object-Oriented Programming Course, *Journal of Information Technology Educatio*n, 11, 271-281.

Ben-Ari, M., Bednarik, R., Levy, B. R., Ebel, G., Moreno, A., Myller, N., & Sutinen, E. (2011). A decade of research and development on program animation: The Jeliot experience, *Journal of Visual Languages and Computing*, 22(*5*), 375-384.

Cooper, S., Dann, W., & Pausch, R. (2003). Teaching objects-first in introductory computer science, *ACM SIGCSE'03*, 19-23.

Esteves, M., & Mendes, A. (2003). OOP-Anim: a system to support learning of basic object oriented programming concepts, *Proceedings of the 4th international conference conference on Computer systems and technologies e-Learning -CompSys-Tech'03*, 573-579.

Fossati, D., et al. (2009). Supporting Computer Science Curriculum: Exploring and Learning Linked Lists with iList, *IEEE Transactions on Learning Technologies*, 2(*2*), 107-120.

Gestwicki, P., & Jayaraman, B. (2005). Methodology and architecture of JIVE, *SoftVis '05 Proceedings of the 2005 ACM symposium on Software visualization*, 95-104.

Hendrix, T., II, J. C., & Barowski, L. (2004). An extensible framework for providing dynamic data structure visualizations in a lightweight IDE, *ACM SIGCSE Bulletin*, 387-391.

Ingalls, D., Kaehlei, T., Maloney, J., Wallance, S., & Kay, A. (1997). Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself, *OOPSLA '97*, 318-326.

Kolling, M., & Quig, B. (2003). The BlueJ system and its pedagogy, *Computer Science Education*, 13(*4*), 249-268.

Liberman, N., Beeri, C., & Ben-David Kolikant, Y. (2011). Difficulties in Learning Inheritance and Polymorphism, *ACM Transactions on Computing Education,* 11(*1*), 1-23.

Lister, R. (2004). Teaching Java first: experiments with a pigs-early pedagogy, *Proceedings of the Sixth Australasian Conference*, 177-183.

Lister, R., et al. (2006). Research perspectives on the objects-early debate, *Working group reports on ITiCSE on Innovation and technology in computer science education - ITiCSE-WGR'06*, 146-165.

Meyer, B. (1997). *Object-Oriented Software Construction 2nd Edition*, Prentice Hall.

Miura, M., Sugihara, T., & Kunifuji, S. (2009). Anchor Garden: An Interactive Workbench for Basic Data Concept Learning in Object Oriented Programming Languages, *Proceedings of 14th ACMSIGCSE Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE2009)*, Paris, 141-145.

Moreno, A., et al. (2004). Visualizing programs with Jeliot3, *Proceedings of the working conference on Advanced visual interfaces - AVI'04*, 373-376.

Ross, J. M. (2005). Polymorphism in decline?, *Journal of Computing Sciences in Colleges*, 21(*2)*, 328-334.

Tarsem S., Purewal, Jr., & Bennett C. (2006). A framework for teaching polymorphism using game programming, *Journal of Computing Sciences inf Colleges*, 22(*2*), 154-161.