Anchor Garden: An Interactive Workbench for Basic Data Concept Learning in Object Oriented Programming Languages

Motoki Miura
Kyushu Institute of Technology
1-1 Sensui, Tobata,
Kitakyushu, Fukuoka
804-8550, Japan
miura@mns.kyutech.ac.jp

Taro Sugihara
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa
923-1292, Japan
sugihara@jaist.ac.jp

Susumu Kunifulji
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa
923-1292, Japan
kuni@jaist.ac.jp

ABSTRACT

We propose Anchor Garden (AG), an interactive workbench software for learning fundamentals of data structures with the concepts of type, variable, object, and their relations in a strongly typed object-oriented programming language (OOPL) such as Java or C#. Learners can approach the basic data-handling concepts of OOPLs by direct manipulation of graphical models in AG. In addition, the learner can correlate his/her manipulation and notation of source code because AG automatically generates appropriate source-code corresponding to this manipulation. Experimental results showed a tendency of a learning effect with AG and high correlations between concept understanding and programming ability. Thus, AG has the potential to enhance the programming ability of novice programmers.

Categories and Subject Descriptors
K.3.2 [Computer and Information Science Education]: Computer science education; D.1.7 [Programming Techniques]: Visual Programming

General Terms
Experimentation, Human Factors, Measurement

Keywords
CS1, object-orientation, reference, data type

1. INTRODUCTION

Novice programmers generally learn a programming language through repetitive tasks of inputting/modifying sample codes, compiling, and execution. The latest integrated development environments (IDEs) reduce the burdens of these tasks and help learners to notice syntax errors and missing declarations. However, to properly associate source code notation with its meaning in execution is still hard for novice programmers because they are not familiar with abstract programming concepts.

Several studies of algorithm animation have been conducted to facilitate this association and understanding of program behavior. Jeliot[7] accepts arbitrary Java source code and provides intuitive visual representations. JAWAA[1] is suitable for generating algorithms such as sorting and tree traverse. For interactive programming environments, Alice[8] uses a 3D world and characters to confirm procedural operations. Scratch[6] and Squeak eToy[4] provide a graphical interface for children to program without editing source code. Although these approaches are effective for understanding algorithms and control flows, and provide intuitive feedback on statements, basic abstract concepts of data handling in OOPLs are not covered by these approaches.

Other popular environments such as BlueJ[5] and the jGRASP[3] object viewer are also significant for enhancing language learning through advanced technology. However, both BlueJ and jGRASP focus on relatively advanced topics such as class design study and code debugging in terms of data structure.

We focus on learners’ acquisition of the basic abstract concepts of data handling in OOPLs such as Java and C#. These concepts include relationships among type, variable, and object, and their properties. In this study, we introduce Anchor Garden (AG), an interactive workbench to facilitate this acquisition for novice programmers.

2. ANCHOR GARDEN

AG is an interactive workbench that facilitates acquisition of basic abstract data model concepts for novice OOPL learners. AG provides graphical representation of type, variable, and object models as well as concrete manipulation of the models.

2.1 Fundamental Operations

Figure 1 shows the initial screen of AG. The screen is vertically separated into three fields: [Type], [Variable], and [Object]. The fields contain data types, variables, and objects, respectively. The learner can select a type by clicking the type button in the [Type] field. While the type is selected, the learner can generate variables and objects by clicking the [Variable] and [Object] fields, respectively.
When a primitive type (int) is selected, the learner can generate only variables (Figure 2), not objects. When the learner selects a String or an array, both variables and objects are generated by clicking.

Although the learner can easily generate objects such as a seal for a rubber stamp, the generated objects gradually disappear in 10 s. This animation implies garbage-collecting behavior. To prevent the disappearance, the learner should link the object to a variable. Figure 3 shows the linking operation, which involves dragging a knob on the variable and dropping it into the target object.

The linking operation is accepted if the variable and object types are the same (or subclass relations). Otherwise, the knob goes back to the variable by shrinking the link with a buzzer sound. Such linking operations intuitively represent the characteristics of associations among variables and objects. The learner can practice the linking rules experimentally. Of course, the learner can de-link by detaching the knob from the object. An object that loses all references also disappears shortly (in about 5 s).

In summary, these simple creating and linking operations allow learners to understand the following topics:

- Meanings of type, variable, and object (data).
- Differences between primitive type (int) and object type (String).
- Reference rule of variables. One variable can refer to only one object.
- One object can be referred to by many variables (Figure 4). In such cases, the substantial data is one. ([str2/str3] in Figure 4 indicates the variables referring to the object.)
- References can be managed.
- An object with no references is disposed of by a garbage-collecting function.

Garbage collection is not essential for learners’ understanding, but the fundamental behavior of the framework should be considered. AG can make learners aware of this behavior with a simple representation. As shown in Figure 1, AG limits the selectable types in the initial mode (int, String, and their array types). Learners who fully understand the content provided in the initial mode can proceed to an advanced mode. The advanced mode (Figure 5) presents base- and sub-classes (Object, Oval, Rectangle, etc.). The learner can understand the reference rules, including subclass relations.

Figure 1: Initial screen of AG.
Figure 2: Create variable.
Figure 3: Linking to object.
Figure 4: Variables linked to object.
Figure 5: Advanced mode and source code window.

2.2 Design

Similar to previous works[6, 4], we employ interactivity as a key component to support understanding. The Language-Independent Visualization Environment (LIVE) system[2] also provides a manipulation function for data structure learning with visual representation. However, the system uses popup menus to interact with the model, and the target domain of the LIVE system is more advanced than that of AG.

We also use concrete models corresponding to target domain knowledge of programming. In this workbench, all possible manipulations and operations by learners correspond to valid modifications of the models. Therefore, learners can practice with the behavior of the workbench, and the experience leads to better recognition of the rule underlying the domain. We consider that this approach is effective if difficulties arise in text description of the rule and its recognition.

In addition to the above factors, we limited the target domain covered by the workbench. AG supports only basic data concepts in OOPLs. However, the learner should acquire domain knowledge step by step, and the advanced mode should be provided gradually according to the learner’s skill level.

2.3 Generating Source Code from Operations

The above functions are designed to teach fundamental phenomena of data in OOPLs and let learners acquire an overview. However, novice programmers should learn the OOPL’s notation in addition to the overview. To aid learn-
ing, we added a function that generates source code statements in AG. A source code window (Figure 5 right) shows the source code statement corresponding to the learner’s manipulation of the visual model of variables/objects.

When the learner creates a String variable, `String str1;` appears in the first line. When a String object is created, `new String("Moji");` appears in the second line. If a link from the variable to the object is established, the second line is rewritten as `str1 = new String("Moji");`. If the object is dismissed, the second line is also deleted. If the link is detached from the object, `str1 = null;` is inserted. In this manner, AG updates the source code to maintain consistency with the model and operation. Learners can observe the source code statements and associate them with their operations. Also, learners naturally grasp the meaning of the “new” and “null” keywords.

In the advanced mode, type casting is automatically shown if necessary. Assume that an Object variable “obj1” links to an instance of Oval that is referred to by another Oval variable “oval1” (Object is a superclass of Oval). In such a case, `obj1 = (Object) oval1;` is generated. Therefore, learners can try several patterns of type casting through operation. Moreover, they can learn that type casting is a notation for reference, not for converting object types.

In the main screen, the order of [Type], [Variable], and [Object] fields from left to right is designed for consistency with the typical appearance of source code statements. Incidentally, if multiple possible notations are generated, one is selected at random, for example, when linking to an object that is already referenced by two variables.

### 3. LEARNING SUBJECTS

AG can be adopted for learning the following subjects, all of which require learners to have a concept of “reference.” We consider references important for programming, and the concept can be supported by AG.

#### 3.1 Arrays

We often begin teaching arrays in primitive data types such as ‘int.’ The declaration of an int array variable as `int[] intAry1;` is straightforward for novice learners. However, to instantiate an array object such as `intAry1 = new int[4];`, the `new` keyword prevents comprehension. Usually learners choose to either (1) study the `new` keyword, or (2) consider it as a special word such as “charm.” With AG, the learner can master the meaning of the `new` and “=” keywords as compared to the String case: `String str1 = new String("Moji");`.

#### 3.2 Array of Objects

After recognizing the int array, the learners proceed to arrays of Objects, such as String arrays. Typical learners imagine arrays as boxes including String objects. To correct this wrong image, AG is effective because it provides visual representation of String arrays by links from an array object to the element object. The link concept is consistent with the reference of variable. Thus, the learner can master the correct image of String arrays without confusion.

#### 3.3 Shallow Copy and Deep Copy

Regarding 3.2, a novice programmer tends to misunderstand between copying an int array and object array. For an int array, the following code works:

```java
for (int n = 0; n < intAry1.length; n++)
    intAry2[n] = intAry1[n];
```

But for an object array, the similar code below does not copy the array’s data completely (shallow copying).

```java
for (int n = 0; n < strAry1.length; n++)
    strAry2[n] = strAry1[n];
```

AG can explicitly express the difference between shallow and deep copying, as shown in Figure 6. AG is effective to avoid such misunderstandings because the meaning of equal (=) is different between primitive and object types.

![Figure 6: Shallow copying and deep copying.](image)

#### 3.4 Swapping Objects

Some learners also have difficulty with swapping objects. A typical method of swapping is the following code, with the temporary variable ‘temp.’

```java
temp = str1; str1 = str2; str2 = temp;
```

With AG, the learner can simulate the process by handling models. If the learner performs the handling in the wrong order, the object will disappear and be lost. Figure 7 shows a snapshot of how swapping is handled.

![Figure 7: Swapping objects.](image)

### 4. EXPERIMENT

We conducted an experiment to examine the effect of AG on novice programmers’ learning.
Table 1: Schedule of experimental lectures

<table>
<thead>
<tr>
<th>Day</th>
<th>A: without AG</th>
<th>B: with AG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apr 9, 16</td>
<td>Introduction, Architecture</td>
<td></td>
</tr>
<tr>
<td>Apr 21, 23</td>
<td>Variable, Data Types, Operators</td>
<td></td>
</tr>
<tr>
<td>Apr 28, 30</td>
<td>if, while, for</td>
<td></td>
</tr>
<tr>
<td>May 7</td>
<td>Mid-term Test</td>
<td></td>
</tr>
<tr>
<td>May 12</td>
<td>Array(1)</td>
<td>Introduction 60min</td>
</tr>
<tr>
<td>May 12</td>
<td>Test(1) 30min</td>
<td></td>
</tr>
<tr>
<td>May 14</td>
<td>Array(2)</td>
<td>Simple Sorting</td>
</tr>
<tr>
<td>May 19</td>
<td>Introduce reference by PPT &amp; WB 60min</td>
<td>Test(2) 30min</td>
</tr>
<tr>
<td>May 19</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>May 21, 26</td>
<td>Function (Tests returned in May 26)</td>
<td></td>
</tr>
<tr>
<td>May 28, Jun 2</td>
<td>Pre and Final Exam.</td>
<td></td>
</tr>
</tbody>
</table>

4.1 Overview

We expect that AG improves understanding of object references and the ability to program with arrays. Thus, we compared test scores on these subjects for two student groups who learned with and without AG, respectively.

We employed 22 graduate students in our institute (JAIST) as participants in the experiment. All participants took a basic programming course. The goal of the course is to understand how the program works and to acquire fundamental programming skills. In this course, we used Visual Studio .NET 2005 to write C# console programming as practice. Fifteen 90-min lectures were conducted 15 times throughout the course. The course consisted of two parts. In the first half, we introduced data types (int, String) and their variables, operators, conditions, and control statements (if, while, for). To avoid complexity, we explained a concept of assigning values to variables using a “box which holds one value” metaphor. In the latter half, we first introduced arrays with the “reference” concept.

4.2 Procedure

At the beginning of the course, we conducted a questionnaire survey regarding PC skill and experience, including programming. Considering the skill level revealed in this survey, we separated the participants into two groups (A: no AG, B: with AG). The course followed the schedule shown in Table 1. On May 12, test(1) was conducted only for group A. Test(1) for group B was conducted after the introduction of AG. In the introduction, we explained the basic functions of AG in the initial mode. The content of test(1) was the same for both groups. We mentioned that the results of test(1) do not affect the final course score. Test(1) requires the participant to write a simple program on a sheet of paper. The requirements of the program were: (1) define an int array, (2) initialize the array, and (3) print the value of the array in ordinary and reverse order.

On May 19, we introduced the reference concept to both groups. During lecture, we presented the differences between (1) primitive type and object type, and (2) shallow copying and deep copying. We distributed the same print material, but we drew figures on a white board for group A, whereas AG was projected on a screen for group B. Moreover, we encouraged the participants of group B to operate AG themselves. After the 60-min introduction, test(2) was conducted for both groups. Test(2) consisted of three sections (Q1-Q3). Q1 was true or false questions for checking recognition of primitive and object types, and shallow and deep copying. Q2 asked students to fill in blanks in source code. Q3 asked them to draw pictures of a reference after executing a code snippet. The answer time was limited to 30 min for both tests. Excepting the participants who missed the test session, we received 18 participant test results (group A: 10, group B: 8). To prevent the inequality in final exam, we explained AG and the experiment on May 26.

4.3 Results

Table 2 shows the distribution of scores for test(1) and (2). The averages are also shown in Table 2, even when the variances were large. From the results of test(1), participants were separated into upper and lower groups. According to test(1), the scores of group B (with AG) tend to be lower than those of group A. Group A had an advantage because they could take the test just after the experiment. Some participants of group B misunderstood that AG might solve the question. In fact, AG just provides the method of initializing an array and the meaning of suffixes, not the answers.

According to the results of test(2) (Table 2 right), the groups were similar. The result of a t-test on the distributions revealed no significant differences. A possible reason is the lack of experience with AG. The participants of group B did not fully understand the meaning of AG.

Table 3 shows the distribution of scores of the final exam, held on June 2. The final exam consisted of five sections. Q1-Q3 required students to write a program with Visual Studio .NET 2005. Q4 was true or false questions similar to Q1 of test(2). Q5 required students to draw pictures of references after executing a code snippet, similar to Q3 of test(2). The maximum points were 10 each for Q1, Q2, and Q4, and 20 each for Q3 and Q5. Thus, the maximum point total was 70.

Table 2: Result of tests (1) (2): n: number of participants.

<table>
<thead>
<tr>
<th>Test(n)</th>
<th>A (n)</th>
<th>B (n)</th>
<th>Test(n)</th>
<th>A (n)</th>
<th>B (n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test(1)</td>
<td>6</td>
<td>5</td>
<td>Test(2)</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>12-13</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>10-11</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
<td>8-9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>6-7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4-5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Ave.</td>
<td>5.6</td>
<td>3.0</td>
<td>Ave.</td>
<td>8.45</td>
<td>9.19</td>
</tr>
<tr>
<td>Var.</td>
<td>1.10</td>
<td>4.0</td>
<td>Var.</td>
<td>5.14</td>
<td>15.00</td>
</tr>
</tbody>
</table>

Table 3: Final exam results

<table>
<thead>
<tr>
<th>Total (Q1-Q5)</th>
<th>max. 70 pts.</th>
<th>Q5 (max. 20 pts.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pts.</td>
<td>A (n)</td>
<td>B (n)</td>
</tr>
<tr>
<td>60-70</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>50-59</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>40-49</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>30-39</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>20-29</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>10-19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0-9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ave.</td>
<td>40.2</td>
<td>52.0</td>
</tr>
<tr>
<td>Var.</td>
<td>140.6</td>
<td>302.9</td>
</tr>
</tbody>
</table>
Table 4: Pearson’s correlations of final exam scores (Group A and B). (* and ** indicates 5% and 1% significance, respectively.)

<table>
<thead>
<tr>
<th>Group A (without AG)</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1</td>
<td>0.70*</td>
<td>0.60</td>
<td>0.14</td>
<td>0.59</td>
</tr>
<tr>
<td>Q2</td>
<td>0.70*</td>
<td>1</td>
<td>0.69*</td>
<td>-0.23</td>
<td>0.43</td>
</tr>
<tr>
<td>Q3</td>
<td>0.60</td>
<td>0.69*</td>
<td>1</td>
<td>0.31</td>
<td>0.83**</td>
</tr>
<tr>
<td>Q4</td>
<td>0.14</td>
<td>-0.23</td>
<td>0.31</td>
<td>1</td>
<td>0.50</td>
</tr>
<tr>
<td>Q5</td>
<td>0.59</td>
<td>0.43</td>
<td>0.83**</td>
<td>0.50</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group B (with AG)</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1</td>
<td>0.51</td>
<td>0.93**</td>
<td>0.17</td>
<td>0.76*</td>
</tr>
<tr>
<td>Q2</td>
<td>0.51</td>
<td>1</td>
<td>0.36</td>
<td>0.60</td>
<td>0.84**</td>
</tr>
<tr>
<td>Q3</td>
<td>0.93**</td>
<td>0.36</td>
<td>1</td>
<td>0.23</td>
<td>0.70</td>
</tr>
<tr>
<td>Q4</td>
<td>0.17</td>
<td>0.60</td>
<td>0.23</td>
<td>1</td>
<td>0.54</td>
</tr>
<tr>
<td>Q5</td>
<td>0.76*</td>
<td>0.84**</td>
<td>0.70</td>
<td>0.54</td>
<td>1</td>
</tr>
</tbody>
</table>

The left side of Table 3 shows the distribution of total scores on the final exam. The right columns show the distribution of Q5 scores. The result of a t-test on the total score revealed no significant difference between the groups ($t(16) = -1.64, p$ (two-sided)$=0.127$). However, the result of a t-test on the distributions showed a tendency of significant differences in average points on Q5 ($t(16) = -2.94, p$ (two-sided)$=0.063$). Based on this result, the level of proficiency in AG may influence the recognition of references in OOPL. The significance may increase with more participants.

In addition, Table 4 show the correlations of exam sections in the final exam for each participant group. The correlations with Q4 were relatively low, but those of Q1- Q3 and Q5 were relatively high. In particular, the correlations of Q3-Q5 tended to be high. Q3 required application of the programming skills. The significance probability of correlation for Group B’s Q3-Q5 (0.70) was .052. From the data, we confirmed that the ability to draw a picture of a reference is significantly correlated with programming ability. Since most of the participants in our experiment were novices, their natural programming ability did not affect the correlations. Therefore, AG, which facilitates recognition of fundamental reference concepts, tends to improve programming ability.

5. CONCLUSION

We developed an interactive workbench that facilitates recognition of fundamental data handling in OOPLs. Intuitive and simple operations (creation and linking) on a visualized model allow learners to understand the acceptable manipulations in the world. Typical behavior such as garbage collection is presented in simple animation. Although presentations of AG do not express behavior inside the interpreter accurately, it is effective for familiarizing learners with the characteristics of the environment. The source code generating function also assists learners’ associations with programming notation. Overall, AG can fill the gap between learners’ imaginations and system behavior, and it lowers the skill levels required to begin programming.

AG is useful not only for learners but also for teachers. The visual behavior of AG can ease the teacher to explain concepts by showing concrete figures.

We have conducted an experiment and evaluated AG with a small group of learners. From the experiment, we found a tendency of improvement in the ability to draw references. We also found that the correlation between ability and programming skill is high. From these points, we can conclude that AG has the potential for positive impact on programming skills.

As future work, we will add other data types such as hashtable and binary tree to AG. The function of calling methods through variables should also be supported. We will enhance the system to cover more advanced topics without losing its advantages (interactivity and consistency). Data type classes in AG are realized as subclasses of a base class that includes visualizing and handling codes. Thus, adding the data type is relatively easy. We will also examine the effectiveness of AG, especially the relationship between usage time and understanding.

Anchor Garden software is available from the URL: http://anchorgarden.mydns.jp/

Acknowledgment

Our research is partly supported by Grant-in-Aid for Scientific Research (20680036) from the Ministry of Education, Culture, Sports, Science, and Technology of Japan.

6. REFERENCES


