Using Memory Diagrams When Teaching a Java-Based CS1

Mark A. Holliday and David Luginbuhl
Department of Mathematics and Computer Science
Western Carolina University
Cullowhee, NC 28723
+1 (828) 227-3951
{holliday, drl}@cs.wcu.edu

ABSTRACT

Understanding the execution of an object-oriented program can be a challenge for a student starting a CS1 course. Diagrams are a key technique to help students understand object abstraction through visualization, as well as a useful metric for student understanding of object-oriented concepts. We believe that a type of diagram that we call a *memory diagram* can be an effective visualization tool for the beginning object-oriented programmer.

Memory diagrams differ from more well-known object diagrams in that they focus on how, in an abstract sense, the memory of the machine changes as the program executes. Though memory diagrams are a simple idea, by careful use of shape and placement, a number of key points about the meaning of a program fragment can be conveyed visually. We illustrate how memory diagrams help students understand programming by showing how they cover some of the key concepts in an object-oriented CS1 course.

We have also found a correlation between a student's ability to construct these diagrams and that student's comprehension of object-oriented concepts. We are proposing to investigate this correlation further, which may lead to a practical metric for determining a student's future successes in CS1 and CS2, as well as pinpointing where certain students may require further assistance.

Versions of memory diagrams have been used by other authors. However, we appear to have developed this approach further than others have.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming

D.3.3 [**Programming Languages**]: Language Constructs and Features - Classes and Objects

K.3.2 [**Computers and Education**]: Computer and Information Science Education - Computer Science Education

General Terms

Measurement, Languages

Keywords

Computer Science Education; Object-Oriented Programming; Java; CS1; Memory Diagrams; Student Assessment

ACMSE'03 Savannah, Georgia USA

1. INTRODUCTION

An object-oriented CS1 course naturally lends itself to a visual description of such concepts as objects, references, and variables. One approach to visual description in CS1 is to introduce class diagrams, object diagrams, and use case diagrams from the Unified Modeling Language (UML) [2]. Such UML diagrams are useful in a CS1 textbook. They introduce software development to a beginner, and they acclimate students to their use in later courses within the computer science curriculum.

The UML diagrams are language-independent and do not focus on the state of memory during the execution of a program. This makes sense since UML was originally intended for modeling, not for learning how to program.

For learning how to program in a particular language, the student is helped by a type of diagram that 1) takes into account the specific features of that language and 2) helps in visualizing the state of the computer as the program executes. In a CS1 course "state of the computer" is synonymous with the state of the memory in an abstract sense. By "abstract" we mean that students do not need to understand such low level details as registers or representation of values in binary, but they do need to understand that programs create "space" to hold variables and objects.

We believe that having a grasp of these memory diagrams is key to understanding object-oriented programming concepts. All programming, and especially object-oriented programming, requires an ability to think abstractly (among other proficiencies). Thus, students who are able to create these diagrams to represent a specific computation (or series of statements) are well on their way to successful CS1 course completion. We are exploring the correlation between student comprehension of these memory

diagrams and understanding of other programming concepts. It is possible that these diagrams can lead to a useful metric for determining a student's success in completing CS1 and CS2 (as well as an entire CS curriculum). The diagrams may also help an instructor pinpoint how to better assist those students who are struggling.

The sections that follow are organized by the order in which the memory diagrams are introduced within our CS1 course and correspond to the order of complexity of the diagrams. In each section we introduce some aspects of the diagrams and relate those aspects to the CS1 concepts that they help the student understand, and we relate the diagrams to previous work reported in the literature. The diagrams are designed for teaching Java. However, with minor modifications they can be used to teach other object-oriented languages.

In section 2 we introduce the most basic constructs of memory diagrams: local reference variables, objects, references, and method calls. Section 3 introduces the diagramming of fields and primitive types and how diagrams handle visibility rules and parameter passing. Section 4 introduces the diagramming of static fields and methods. Section 5 covers the representation of arrays using memory diagrams. Section 6 describes how we think memory diagrams may serve as a metric for student comprehension of programming concepts. In section 7, we review related work. We close with a summary in Section 8.

2. OBJECTS, VARIABLES, AND REFERENCES

The ordering of topics in our course corresponds to their order in the textbook by Arnow and Weiss[1]. At the start the memory diagrams are simple, as we merely represent local variables of a reference type and the creation of instances of the String class. The String class serves as an attractive initial example because the state of a String object is merely the value of the string. From this the concept that objects can contain fields can be deferred.

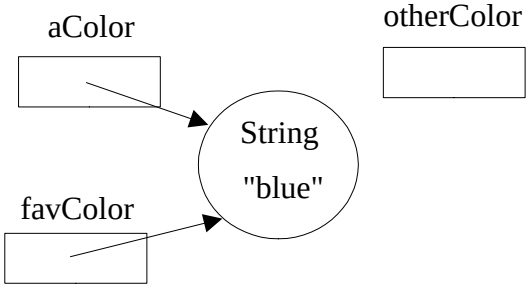

Figure 1: Reference variables, references, and String objects.

Figure 1 shows an example of two variables holding references to the same String object, as well as a variable that does not hold a reference. Though simple, this example illustrates several key points. The first point is that shape matters. A key assumption of ours is that a beginning CS1 student learns more effectively if we use different shapes for each type of entity being modeled. The diagrams should represent each type of entity with a clearly distinguishable shape. Variables, objects, and classes are three key types of entities and so should have shapes as different as possible as an aid for the beginning CS1 student. Unfortunately, rectangles represent classes in UML class diagrams and rectangles represent variables and objects in UML object diagrams. Shape is not being used in UML to distinguish between these concepts.

In contrast, our memory diagrams use rectangles for variables, circles for objects, and diamonds for classes. The representation of a class itself in a memory diagram occurs only in the second half of our CS1 course because a class only needs to be represented when static fields and static methods are introduced.

On a related note, entities that are fundamentally the same should use the same shape as a visual hint to the student that they are the same. In particular, local variables, formal parameters, and fields are all variables. Thus, they are all represented as rectangles.

The second key point is that we emphasize that the arrow for the references originates *within* the variable's rectangle. Starting within the rectangle reminds the student that the reference is what the variable contains. That a variable can reference only one object is clear as there is space for only one arrow within the variable's rectangle. Thus, when a different reference is assigned to a variable, the old reference must disappear since there is no space for it. Furthermore, until a reference is assigned to a variable, the variable is empty. Attempts to dereference the variable must fail since there is nothing inside the rectangle. We postpone addressing the fact that a reference variable initially has the null value.

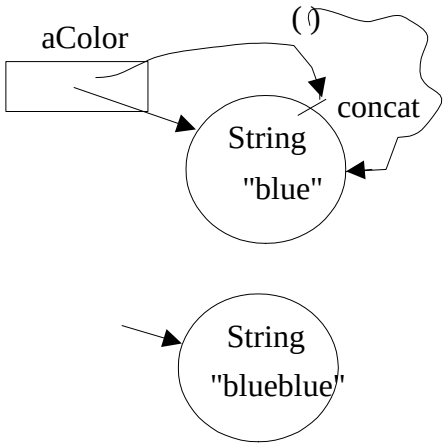

Figure 2: A concat method call returning a reference to a new object. The method call being represented is aColor.concat(aColor). The top half of the figure represents the call on the method concat. The bottom half of the figure represents the new object and new reference that are created.

The third point the diagrams illustrate is that while variables have names, objects do not have names. Note that each rectangle (variable) is labeled above with a name, while objects are only accessed by following a reference arrow. The name of the variable is not inside the rectangle since the name is not what the variable holds. Although objects do not have names, they do belong to a specific class, so we label objects (inside at the top) with the class to which they belong.

Once the roles of names, variables, references, and objects are clear, the next step, depicted in Figure 2, shows how action takes place. A public instance method is represented by a short line crossing the border of an object. It is on the border since it is public and thus needs to be visible outside the object. The method in Figure 2 is with the object since it is an instance method. Method calls appear as wavy arrows (to contrast with the straight arrows of references) beginning at the reference being used to determine the target object and ending at the line for the method attached to the target object.

In the middle of the wavy arrow are the parentheses for the argument list. If there are any arguments, the arguments are shown within the parentheses. Since all values addressed so far have been references, having an argument means having a reference arrow within the parentheses that points to the object being referenced. If a method call returns a reference to a new object, then the diagram shows the new object and the reference to it as depicted in Figure 2 (the lower of the two String objects).

3. PRIMITIVE TYPES, FIELDS, VISIBILITY, AND PARAMETER PASSING

Because variables of primitive type hold actual values as opposed to references, we treat them differently in the memory diagrams. Whereas the rectangles representing reference variables contain the beginning of a reference arrow, rectangles representing primitive variables contain the actual primitive value. Until this point, our variable depictions have all dealt with local variables, but fields are also variables and as such, they are also represented using rectangles. We draw the rectangle inside the associated object to represent private instance fields.

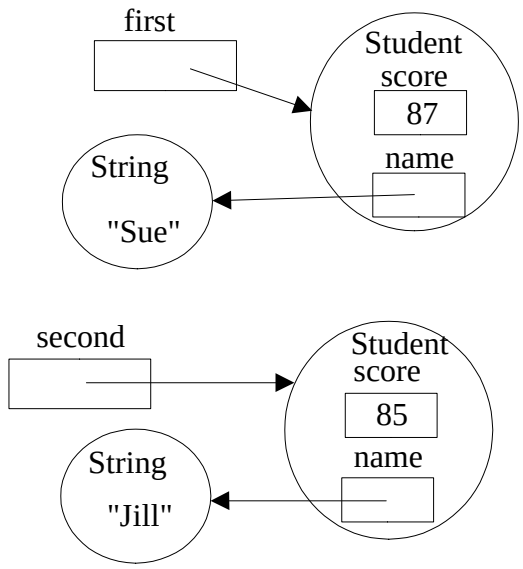

Figure 3: An example of the representation of primitive types, instance fields, and multiple instances of the same class.

Figure 3 shows the difference between primitive type variables and reference variables as well as the representation of instance fields. There are two objects of the same class . Each object has two private instance fields. The score field holds a value of an int primitive type. The name field holds a value which is a reference to a String object.

Diagrams such as those in Figure 3 make clear the difference between primitive type variables and reference variables. They also make clear that the fields of two different objects of the same class are distinct. For example, changing the value in the field named score in the object referenced by variable first does not change the value in the field named score in the object referenced by variable second.

We depict the visibility rules for fields and methods by the placement of the field or method with respect to the associated object. Private fields and methods are shown inside the object. Thus, it makes sense intuitively to the students that such objects are not accessible outside of the object. Public fields and methods are shown on the border of the object to reinforce that they are accessible from outside of the object.

Parameter passing can be a subtle point for many CS1 students. Memory diagrams help in understanding the implications of Java's call-by-value semantics. The key is to have a diagram that combines both the memory diagram and the relevant code with the rectangles for the variables placed next to the corresponding declaration in the code fragment. The code for the called method as well as the code for the calling method are shown as illustrated by Figure 4.

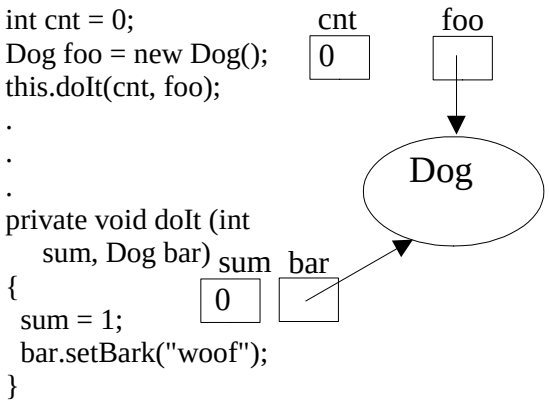

Figure 4: Illustrating that Java parameter-passing is call-by-value. Diagram shows state <u>before</u> execution of sum=1;.

The rectangles for the parameters sum and bar of the method doIt are shown next to the method header to remind the student that those variables exist and are visible only within that method. The memory diagram shows the state immediately before the execution of the statement sum = 1;. Thus, both of the parameters of the doIt method have been initialized to the values passed from the variables in the calling code.

If the value passed is a reference the implications differ from the implications if the passed value is of a primitive type even though call-by-value semantics are being used in both cases. As Figure 4 illustrates, assigning to the variable sum in procedure doIt has no effect on the variable cnt. However, calling the method

bar.setBark does have an effect on the object referenced by the variable foo.

4. STATIC FIELDS AND METHODS

Static fields and methods are often glossed over in CS1 textbooks. We think this is a mistake. CS1 students generally see a number of static fields and methods in Java code examples. For instance, in and out are static public fields of the System class. The methods of the Math class that are commonly used (such as pow and random) are static. Math class named constants such as PI are static public fields. Explaining the meaning of static avoids confusion on the student's part (e.g. "Why is there a "." in System.out?") and is straightforward with memory diagrams.

As mentioned earlier, we represent classes using diamonds in order to distinguish class fields and methods from those associated with objects. The diamond for the class is drawn roughly the same size as the circle for an instance of that class. The diamond is labeled at the inside top with the name of the class. Each private static field is represented as a rectangle inside the diamond. Public static methods are represented as short lines crossing the border of the diamond. Figure 5 is an example illustrating the method call System.out.println("Hello world");.

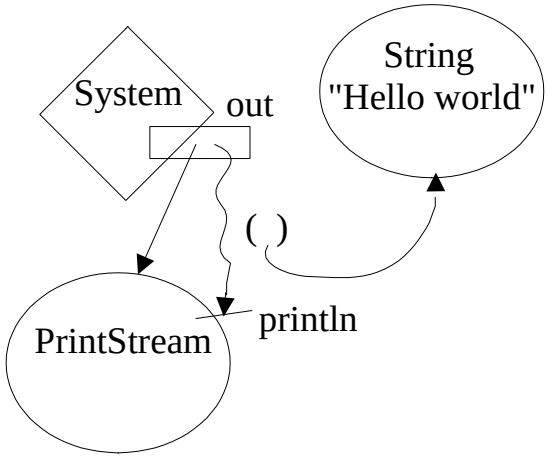

Figure 5: The memory diagram for the statement System.out.println("Hello world");. This is an example of representing static fields and illustrates how common static fields are in CS1.

How a static int field can be used to assign a unique identifier to each instance of a class is clear with a memory diagram. The incrementing of the static field is done within the field's rectangle in the diamond for the class instead of within a rectangle of the circle of some instance of the class.

5. ARRAYS

Memory diagrams help in understanding how arrays are implemented in Java. In particular, arrays are objects so they are represented by circles with references pointing to them. Consider the following Java code fragment where the variable grades holds a reference to a two-dimensional array.

```
for (int row = 0; row < grades.length; row++)
  for (int col = 0; col < grades[row].length; col++)
    grades[row][col] = 0;
```

That grades.length is a field and grades[row].length is also a field is much less confusing to a student who considers Figure 6. Figure 6 shows that the Array object for the first dimension has a length field and the Array object for each row has its own length field. This figure also makes clear why ragged arrays (where each row can be of a different length) are possible.

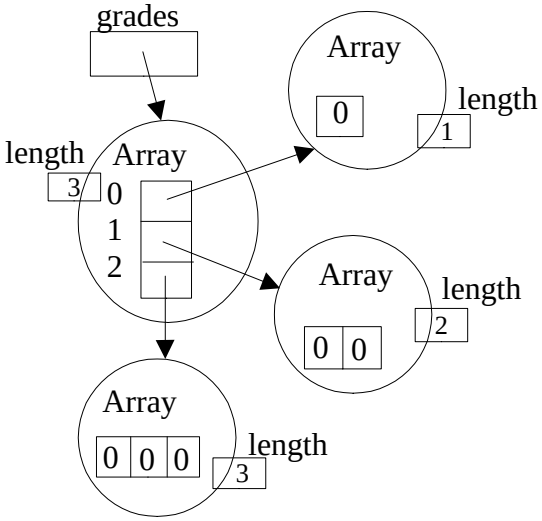

Figure 6: The memory diagram for a two-dimensional ragged array of int values.

6. USING DIAGRAMS TO MEASURE STUDENT COMPREHENSION

At Western, CS1 is the first course required for the CS major. As such, it attracts many students who are still undecided about a major and are not quite sure what is involved in computer programming. We believe that our diagrammatic approach to teaching helps students understand programming concepts, but by requiring students to create these diagrams themselves, we also have a useful tool for determining how well students are able to think abstractly. This allows us to use the diagrams to measure student comprehension of object-oriented programming concepts.

In addition to the instructor displaying diagrams to illustrate certain ideas like object construction, variable assignment, and the difference between primitive and reference variables, we require students periodically to construct diagrams on their own to show that they understand the concepts. We ask students to draw diagrams in groupwork and in directed laboratories. We also include memory diagram constructions as test problems.

We thus see these diagrams both as teaching tools and as measurement tools. We propose to study further just how effective diagrammatic techniques are for measuring programming comprehension. This will involve more rigorous use of these diagrams in a variety of evaluation instruments. We describe some preliminary results from Fall semester 2002 below.

In one class, the final exam included a problem that showed the following code as well as a description that explained that the Dog class has one String field called **name**.

```
Dog spot;          // figure a)
spot = new Dog("spot");
       // right hand side is figure b)
       // left hand side and equal sign is figure c)
System.out.println(spot.toString());      //  figure d)
```

As indicated by the comments in the code the student is to draw a series of figures. Requiring a series of memory diagrams is important since it makes clear that the student understands the execution sequence as well as what the final result of the code fragment looks like.

Figure a) should display just the rectangle for the reference variable. The typical problem is that students often think that the Dog object is also created by such a declaration. By making the students draw memory diagrams, we can detect this misunderstanding. By the time of the final exam none of the students still had this misunderstanding. Eleven of the thirteen students who took the final exam drew it correctly. The other two just made the mistake of labelling the rectangle with the name of the class instead of the name of the variable.

Figure b) involves creating the instance of the Dog class, indicating that it has a field called name that is a reference variable pointing to a String object whose value is "spot", and also creating the reference to the Dog object. There are several common misunderstandings that this memory diagram detects. One is not realizing that besides creating the Dog object, it also creates the reference (the arrow) to the Dog object. A second is prematurely assigning to the variable spot the reference to the Dog object (this should be in figure c). A third is failing to put a rectangle in the Dog object for the name field (recall that a field is represented by a rectangle, since a field is a variable; Figure 3 above shows an example with two fields inside an object). A fourth common error is forgetting that a String, "spot", is an instance of a String object and so a circle for that String object must be created and a reference to that String object must be placed inside the name field of the Dog object.

Four of the thirteen students drew the diagram with no errors. Two students had only one mistake: they put "spot" inside the name field instead of realizing that "spot" is a separate String object (a circle) connected by a reference (an arrow). Four others, besides misunderstanding that "spot" is a String object, did not show that a field called name exists. In other words, there was no rectangle for the name field inside the Dog object circle. The remaining three students had more serious problems.

Figure c), representing assignment, involves placing the reference to the Dog object inside the rectangle for the variable named spot. All but one of the students drew this diagram correctly.

Figure d) is more challenging. It represents method composition, so it involves several different events. First is a call to the toString method of the Dog object referenced by the variable spot. Second is the result of that method call, which is the creation of a String object and a reference to that String object. Third is passing that just created reference as the argument in the call to the println method of the PrintStream object that is referenced by the static public field out, which is a field of the System class (similar to Figure 5 above). Clearly a student who can diagram this statement correctly has an excellent understanding of the material in a Java CS1 course.

All but three of the students correctly showed the first event, which is the call to the toString method of the Dog object. Of the remaining ten students, two had completely correct diagrams except that they just showed System.out as a rectangle instead of as a static public field (which would require a diamond labelled System that has a rectangle on its border for the field named out). The eight students in the middle split into three groups. One student showed nothing correctly beyond the toString method call. A second group (five students) correctly diagrammed a call to the println method of the PrintStream object from a rectangle representing the System.out variable. However, they did not diagram correctly how that method call relates to the other method call. Their key problem was that they did not show the creation of the String object resulting from the toString method call. The reference to that object is what is passed to the println method call. The third group did the opposite. They showed the created String object but did not know how to diagram the println method call.

How well did performance on this question predict performance on the entire test? As shown in Table 1 there is a strong correlation between the two scores. Anomalies, however, can occur. For example, the student with a test score of 39 made an 8 on the question and the student with a test score of 78 made a 4 on the question.

Table 1. Score on the question (the Q row) versus the score on the test (the T row).

Q	***11***	***11***	***10***	***9***	***8***	***8***	***8***	***7***	***7***	***6***	***5***	***5***	***4***
T	97	97	85	97	89	74	39	83	77	77	41	30	78

If our preliminary and anecdotal results are any indication, these diagrams are useful for identifying which concepts students do not understand and are also useful as a predictor for success in a CS1. We do not mean to imply that students who perform poorly at constructing these diagrams should be ushered out of our program. Rather, poor performance at diagram construction may require alternative approaches for reinforcing the abstracting skills required of a student in a CS program.

6. RELATED WORK

Memory diagrams can be viewed as one means of using visualization to help the beginning programmer understand program execution. They are diagrams that use shape and placement to describe how specific language features are reflected in the execution of particular program statements. They differ from algorithm animations [6] [9] since such animations focus on visualizing more abstract, algorithmic aspects of a program in a non-language-specific manner. A number of program visualization tools for beginning programmers have also been developed[3][5]. Such tools tend to emphasize a programming environment that enhances the quality of visual debugging and testing. Their focus is not on a diagrammatic technique for visually representing the differences in language features that occur in a program fragment.

A further clear difference from the above related work is that we are proposing a pedagogical technique, not a software tool. More closely related are other diagrammatic techniques. The most

widely known technique is the Unified Modeling Language (UML). UML diagrams are appearing increasingly in CS1 textbooks for Java [7]. They are useful but have limitations for the student starting to learn Java. UML class diagrams are most meaningful in capturing relationships such as inheritance and the use of abstract classes and interfaces. However, we do not reach those features in our CS1 until near the end of the semester. For most of the semester the students deal with one or more independent classes. During this time the class diagrams are basically a well-defined table format for listing the fields, constructors, and methods of a class.

UML object diagrams can be confusing to the beginning Java CS1 student since they associate a name with an object. We find that a key point for the student to understand is that objects in Java do not have names, because giving objects names distracts from the notion that an object must be referenced by a variable to be useful. Variables have names and variables may refer to an object, but the object itself does not have that name. Also, UML object diagrams do not take advantage of the many possible visual hints of shape and placement that aid the beginning student.

One such example of non-UML diagrams used to illustrate the state of memory during program execution exists for linked lists [4]. Wu [10] comes closest to our approach. Our memory diagrams can be viewed as an extension of his state-of-memory diagrams though they were developed independently. Wu makes two contributions. First, he uses shape to distinguish between variables (rectangles) and objects (ovals). Second, he uses placement to make clear that reference variables hold references and primitive type variables hold primitive values. We also use memory diagrams to visualize method calls (including arguments), to distinguish between static and instance fields and methods, to distinguish between public and private fields and methods, to represent fields, to show that fields, parameters, and local variables are all variables, to represent arrays, and, in general, to show how more complex code fragments can be analyzed visually and diagrammatically.

Regarding use of these diagrams for student evaluation, we acknowledge that much work has been done in innovative assessment techniques (for example, [8] describes an automated environment for detecting specific hurdles to student understanding). But there does not appear to be much research into having students create abstract representations in order to assess comprehension of object-oriented programming concepts.

7. SUMMARY

The level of abstraction present at the start of a CS1 course teaching an object-oriented language in an objects early approach can be a challenge for many students. We have found that if the students can diagram what is happening in memory as a fragment of object-oriented code executes, they can more easily and more deeply understand the meaning of that program fragment. Such memory diagrams represent memory in an abstract sense. Similar diagrams have appeared in the literature. However, over several years of using them in teaching CS1 we have extended the diagrams and identified how to use them more extensively than has been done previously.

By use of different shapes for different entities and by careful use of placement the student can use the diagrams to visualize a significant amount of the meaning of the effect of the execution of a program fragment. For example, different shapes remind the student of the differences between variables (rectangles), references (straight arrows), objects (circles), classes (diamonds), methods (short straight lines), and method calls (wavy arrows).

Placement of fields and method, whether within or on the border of an object or class, reminds the student of the difference between public and private. Whether a field or method is placed with a circle (an instance of a class) or with a diamond (the class itself) reminds the student of the difference between instance fields/methods and static fields/methods. That variables have names and objects do not is reflected in the placement of the name next to the variable, not the object. That the name is not what the variable contains is reflected by the reference or primitive value being placed within the variable's rectangle while the variable name is next to the rectangle.

Furthermore, the subtle implications to the beginning programmer of call-by-value are made clearer by the placement of the variables in the proper places in the corresponding code fragment. Once the memory diagram is drawn, the feasibility of multi-dimensional arrays changes from a mystery to common sense.

Finally, having students construct these diagrams can serve as a useful assessment tool. If a student can think abstractly enough to render these memory diagrams, chances are better that the student will successfully complete the CS1 and CS2 sequence. We hope to study this correlation further to develop a more rigorous assessment approach using these diagrams.

8. REFERENCES

[1] Arnow, D. and Weiss, G., *Introduction to Programming Using Java: An Object-Oriented Approach*, Addison-Wesley, 2000.

[2] Booch, G., Rumbaugh, J., Jacobson, I. *The Unified Modeling Language User Guide*, Addison Wesley, 1999.

[3] Dershem, H.L. And Vanderhyde, J., *Java Class Visualization for Teaching Object-Oriented Concepts*, Proc. of SIGCSE 1998, pp. 53-57.

[4] Dershem, H. and McFall, R., *Animation of Java Linked Lists*, Proc. Of SIGCSE 2002, pp. 53-57.

[5] Haddad, H. Curtis, E. and Brage, J. Visual Illustration of Object-Orientation: A Tool for Teaching Object-Oriented Concepts, *The Journal of Computing in Small Colleges*, 12, 2, (Nov. 1996), 83-93.

[6] Naps, T.L., Eagan, J.R., Norton, L.L., JHAVÉ--An Environment to Actively Engage Students in Web-based Algorithm Visualizations, Proc. of SIGCSE 2000, pp. 109-113.

[7] Riley, D. *The Object of Java*, Addison Wesley, 2002.

[8] Satratzemi, R., Dagdilelis, *V.*, and Evagelidis, G. *A System for Program Visualization and Problem-Solving Path Assessment of Novice Programmers*. Proc. of ITiCSE 2001, pp. 137-140.

[9] Stern, L. and Naish, L., *Visual Representations for Recursive Algorithms*, Proc. of SIGCSE 2002, pp. 196-200.

[10] Wu, C. T. *An Introduction to Object-Oriented Programming with Java*, McGraw-Hill, 2001.