Let Numbers Point Students the Way: Address-Based Memory Diagrams for OOP

Franck van Breugel and Hamzeh Roumani
Department of Computer Science and Engineering
York University
Toronto, Ontario, Canada
{franck, roumani}@cse.yorku.ca

ABSTRACT

We argue that the conventional approach of representing pointers as arrows in memory diagrams may have certain limitations for internalizing the semantics of OOP in CS1/CS2. We introduce a new set of memory diagrams that are based on addresses rather than arrows. We show how these diagrams can be applied to reason about object manipulation in a variety of settings, from the simple one-component case to multiclass applications involving inheritance, aggregation, and arrays.

Categories and Subject Descriptors

K.3.2 [**Computer and Education**]: Computer and Information Science Education – *Computer science education.*

General Terms

Languages.

Keywords

Memory diagram; reference; address; CS1; OOP; Computer science education.

1. INTRODUCTION

A picture is worth a thousand words and a good diagram can be a powerful tool to learn complex ideas. For a diagram to be "good", however, it must not only enable the teacher to convey ideas, it must also provide a mental framework in which the student can reason about the ideas and answer self-posed questions. The key to success here is abstraction: the diagram should hide details deemed irrelevant. If the abstraction captures the correct details, it enables the student to internalize the material and empowers her to apply it to new situations. But if the abstraction hides *relevant* details then it may limit the student's ability to reason about certain aspects of OOP in terms of the diagram.

One area in which diagrams can be extremely powerful is the conceptual foundation of OOP. In particular, understanding the difference between an object and an object reference is a threshold concept for the student: Get it right, and everything falls into place; get it wrong, and object manipulation becomes likes magic: assignments of object references, equality of objects, passing or returning an object, all become unrelated concepts rather than manifestations of the same idea. Because of this, most CS1 textbooks like, for example, [1,5,6] use diagrams of one sort or another to explain the difference between an object and an object reference. And although the usage of diagrams in textbooks tends to be ad hoc, it invariably relies on representing the reference and the object as floating shapes connected by an arrow that emanates from the reference and ends at the object. In an attempt to develop such memory diagrams in a uniform and consistent way, Holliday and Luginbuhl [3,4] introduce several types of shapes, e.g. rectangle, oval, diamond, rectangle-in-oval, etc., to represent a larger set of entities. They capture instantiation and invocation using different arrow shapes, e.g. straight, wavy, and double arrows. Nevertheless, the abstraction theme of all these works remains the same: a two dimensional plane represents memory, geometrical shapes represent entities (e.g. objects and references), and arrows connote pointing at an object or invoking a method.

It is our conjecture that this abstraction omits *relevant* details; namely, the value of references. We rely on these values when we argue that two variables are equal, and we use it to reason about parameter passing. When an abstraction omits the value and replaces it with an artifact such as an arrow, one can no longer appeal to the student's preexisting knowledge, such as the intuitive notion of equality. We therefore believe that the notion of a memory address must survive the abstraction process and be captured in the diagrams. To demonstrate this, we introduce new, address-based, memory diagrams and then compare their pedagogical roles with conventional ones.

We introduce our address-based memory diagrams in Section 2 and apply them in Section 3. Section 4 and 5 demonstrate that our diagrams can be used in complicated multiclass applications involving inheritance, aggregation, and multi-dimensional arrays. In Section 6 we present a comparison between our address-based diagrams and arrow-based ones.

2. DIAGRAM TYPES

Suppose that there is a class named `Square` and a class named `Client` that uses it. The situation is depicted in Fig. 1 using a standard UML diagram. `Square` has two static (underlined) features: a private attribute `count` to keep track of the number of times the class has been instantiated and a corresponding public accessor `getCount()`. We will use this scenario as a basic example throughout the paper.

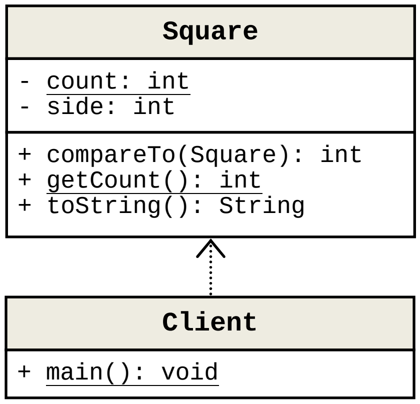

Figure 1. UML of the basic example.

A memory diagram provides a pictorial abstraction of a snapshot of the memory used by a Java application at some point during its execution. We depict memory as a sequence of blocks each of which has an address, an arbitrary yet unique non-negative integer. As the application executes, and depending on the encountered statements, new blocks may get allocated and existing blocks may get updated or de-allocated. There are three types of blocks: *class*, *object*, and *invocation*.

2.1 Class Blocks

A class block is allocated when a class name is first encountered and is never de-allocated thereafter. For example, a class block is allocated when the name `Square` is first encountered in the `main` method of `Client`. In our model, the "cover story" is that a class block is thought to contain all the static attributes of the class along with their values, the class constructors, and all the methods of the class (whether static or not). Not all these features, however, need to be included in the diagram. As in UML class diagrams, one chooses what to include based on the level of details that needs to be revealed. A minimal class block contains only a title compartment, as shown in Fig. 2 for the `Square` class using the (arbitrary) memory address `200`.

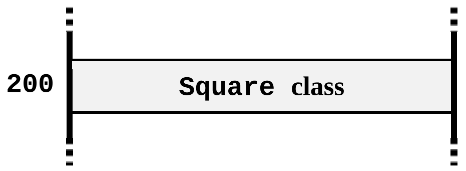

Figure 2. A minimal class block for `Square`.

If additional details need to be exposed then additional compartments are added as needed, exactly as in UML. For example, Fig. 3 shows the same class block but with three additional compartments for the static attribute `count` and its value `0`; the constructor; and the three methods.

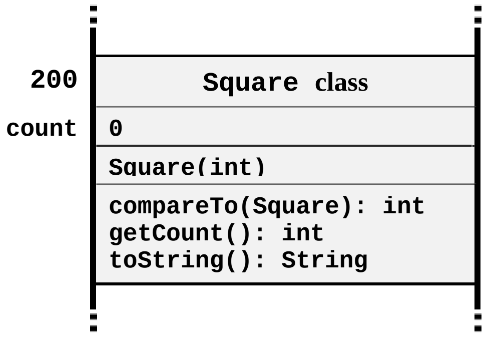

Figure 3. A class block with three compartments.

2.2 Object Blocks

An object block is allocated whenever an object is created and is de-allocated when that object is orphaned. This block is thought to contain the *state* of the object, i.e. the non-static attributes and their corresponding values. Again, the level of details exposed in the diagram depends on the situation being analyzed. For example, if all we know is that an instance of `Square` has been created, we draw a diagram similar to the one shown in Fig. 4. It shows that an object block has been allocated at address 300.

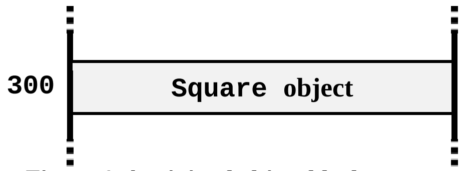

Figure 4. A minimal object block.

On the other hand, the execution of the statement:

```
new Square(3);
```

will trigger the allocation of the object block of Fig. 5.

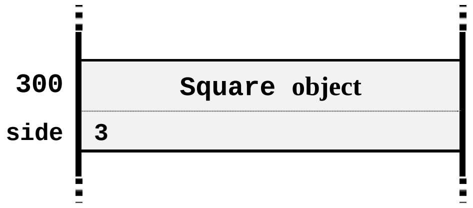

Figure 5. An object block with state.

2.3 Invocation Blocks

An invocation block is allocated when a method is invoked and is de-allocated when that method returns. It is thought to contain all the parameters of the method along with their values and all the local variables of the method along with their values. Note that for non-static methods, the parameters include the implicit parameter `this` (the value of which is the address of the object on which the method was invoked). Note also that the level of details exposed in invocation blocks depends on the concern (or role): Clients can only see the method parameters whereas implementers can also see its local variables. We will see examples of this type of block in the next section.

3. AN APPLICATION

Suppose that the `main` method of `Client` contains the following code fragment:

```
1  int length = 3;
2  Square first;
3  first = new Square(length);
4  Square second = first;
5  second = new Square(5);
6  int flag = first.compareTo(second);
```

How does a student reason about this fragment and internalize the role of each of its statements? The diagram in Fig. 6 depicts memory when execution reaches the end of Line 2. Since an application always starts by invoking `main`, we have an invocation block at address 100 (for example). The invocation block shows the local variables of `main` and their values. When Line 1 is executed, a local variable `length` is declared and is initialized to 3. Line 2 declares `first` but does not initialize it, and hence, the diagram leaves its value blank. Furthermore, Line 2 is where we first encounter the `Square` class name, and this leads us to allocate a class block for it at address 200, for example, with the `count` attribute initialized to 0.

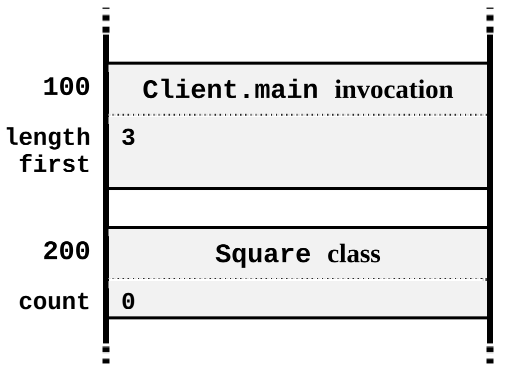

Figure 6. The memory model after Line 2.

Next, let us reason about the next two lines in the fragment, Line 3 and 4, and amend the memory diagram as shown in Fig. 7. The right-hand side of Line 3 creates an object so we allocate an object block with state (`side` = 3) at address 300. The assignment in that line assigns the address of the created object to the object reference `first` so we can now fill in the blank value of that reference and write 300. The creation of the object also leads to incrementing the `count` attribute in the class block at 200. Line 4 declares a new object reference but does not create an object. Hence, no new blocks are allocated and we simply copy the value of `first` (i.e. 300) to `second`. Note that the model leaves no doubt in the mind of the student as to the difference between the object reference and the object at which it points.

Fig. 8 is drawn after executing the next two lines of the fragment, Line 5 and 6, but just before the method `compareTo(Square other)` returns. The `new` operator in Line 5 leads to the creation of the object block at address 500 as shown. And the assignment in that line would then replace the value of `second` with 500. At that point no value has yet been assigned to `flag`, and that is why it is left blank in the figure. The invocation block at 600

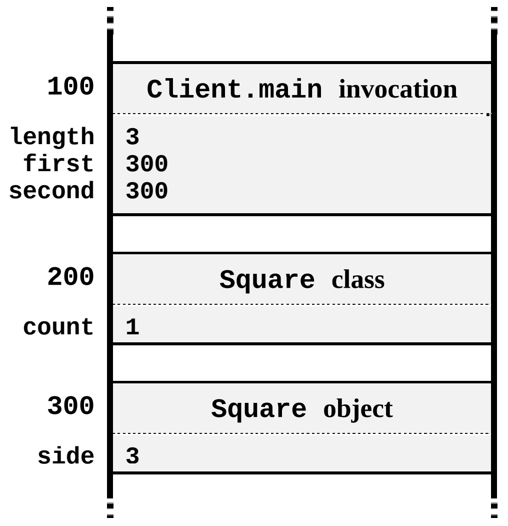

Figure 7. The memory model after Line 4.

assigns 300 (the value of `first`) to the implicit parameter `this` and 500 (the value of `second`) to the parameter `other`. Note how the invocation block models call-by-value: the client's variables `first` and `second` are not in the invocation block, only their values are. Note that once `compareTo` returns and Line 6 gets executed, the invocation block at 600 will be de-allocated and the `flag` variable will pick up a value.

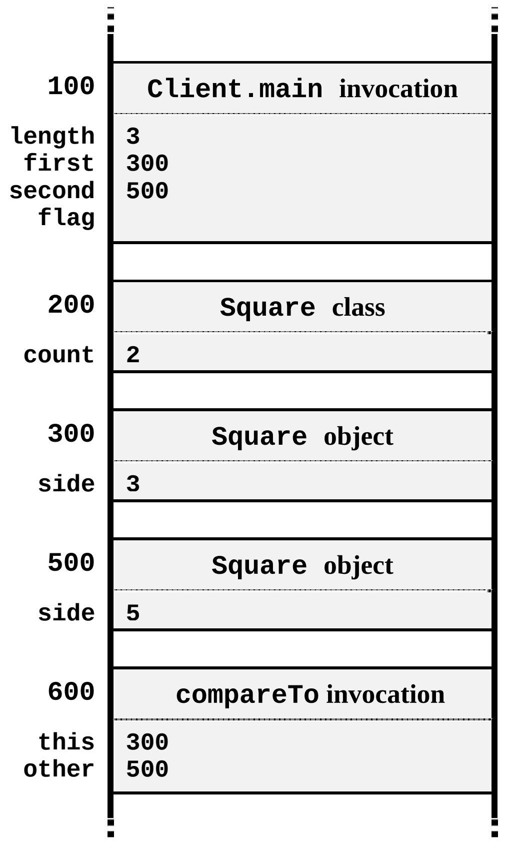

Figure 8. Memory just before `compareTo` returns.

4. INHERITANCE AND AGGREGATION

In this section we show how multiclass applications can be captured in our diagrams. Consider the class `ColoredSquare` that encapsulates a colored square by extending the `Square` class. The situation is depicted in the UML diagram in Fig. 9. We see that the subclass features a (private) color attribute, provides a public accessor for that attribute, and overrides the `toString` method of its superclass.

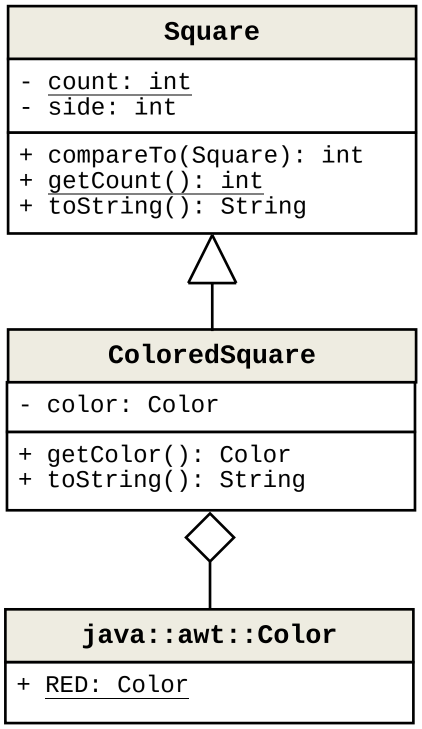

Figure 9. UML for Section 4.

Suppose now that the `main` method of a client of these classes contains the following code:

```
int x = 7;
Color c = Color.RED;
Square s = new ColouredSquare(x, c);
System.out.println(s.toString());
```

That `ColoredSquare` aggregates `Color` is readily handled by our blocks. For inheritance, we extend our class and object blocks. To that end, we adopt the same approach used, for example, by *javadoc,* to create an API document (see Fig. 10): the non-static features of the superclass are appended, without duplication, to those of the subclass[1]. Hence, the methods compartment of the class block at 2100 lists the methods of `ColoredSquare` followed by the non-static methods of its superclass except for `toString` (since it is overridden).

[1] Static features are treated differently in our diagrams than in *javadoc* since they are *not* depicted in the subclass block.

Similarly the object block at 2200 lists the state of the subclass (`color`) followed by that of the parent (`side`). The invocation block at 2500 represents an invocation of a `toString` method on the object at 2200. As the diagram shows this object is a `ColoredSquare` object. Hence, the `toString` method being invoked resides in the class block at 2100. Because of the way the class block is constructed, and because `toString` appears in the first method compartment of the class block, it is the `toString` method defined in the `ColoredSquare` class that gets invoked, rather than the one in `Square`.

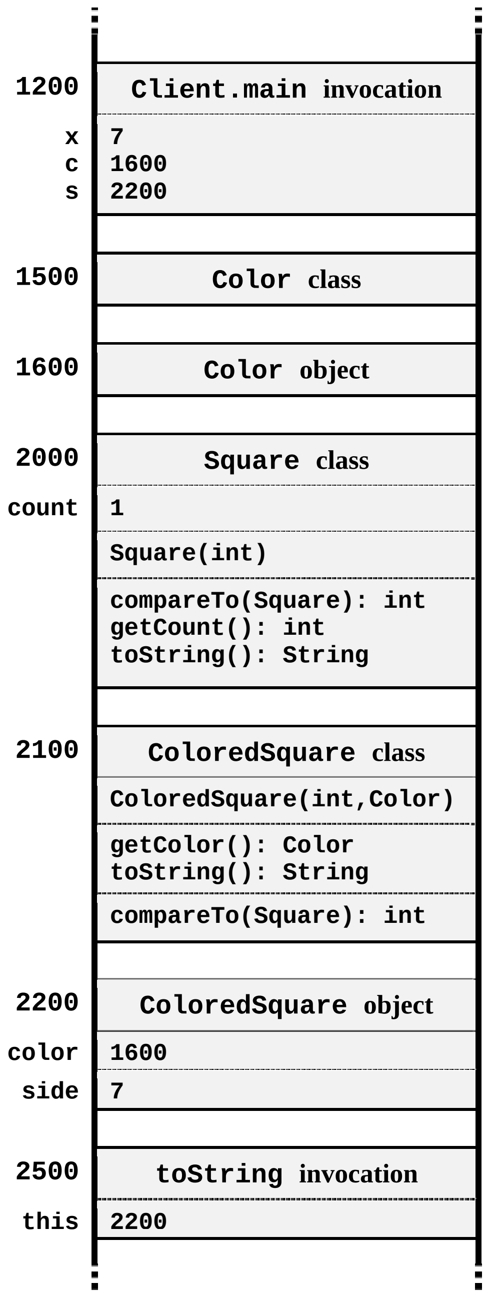

Figure 10. Memory just before toString returns.

5. ARRAYS

Our model can also handle arrays of multiple dimensions. As an example, consider the following code fragment:

```
Square[][] matrix = new Square[3][2];
matrix[0][1] = new Square(5);
```

The corresponding diagram is shown in Fig. 11. The array object block has an attribute `length` plus as many attributes as there are elements. Uninitialized elements default to a value appropriate for the type, which is `null` for class types.

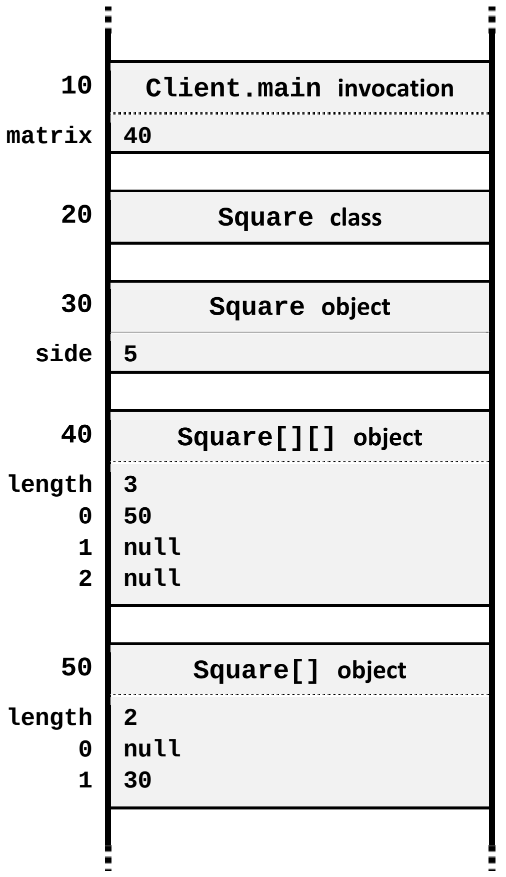

Figure 11. Memory diagram for a 2-D array.

6. RELATED WORK

In this section we compare our diagrams with those based on arrows. Fig. 12 is based on the work of [3,4] but it typifies all the memory diagrams that we have seen in the literature.

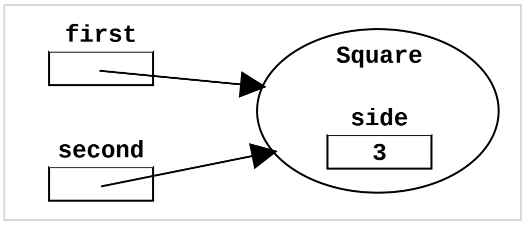

Figure 12. An arrow-based equivalent to Fig. 7.

A student looks at this figure and asks a legitimate question:

"*What is the value of* `first`*?*"

There does not seem to be a convincing answer to this question. If the answer is "*The value is the arrow*", then `first` and `second` cannot be equal since there are two different arrows in the diagram. And if the answer is "*The value is the object*", then this blurs the very distinction that the diagram was meant to assert, namely, that the object and the reference are two different entities. And although the answer "*The reference does not really have a value*" may be tempting, in OOP languages such as Java references do have values [2, Section 4.3.1]. This seems to leave only one answer: "*The value is the endpoint of the arrow*", but this requires that we reason about and manipulate arrows, something new to students, rather than reasoning about and manipulating numbers, something students already know. Contrast all these answers with the one derived from our version of the same diagram, Fig. 7: "The value is 300".

7. CONCLUSION

We presented a new type of memory diagrams characterized by simplicity, intuitiveness, and endurance. The model is simple because it upholds the premise that every variable has a value, and hence, enables the treatment of all variables on equal footing in assignment and parameter passing. It is intuitive because it leverages the student's understanding and/or familiarity with API documents, UML, and computer memory. And as we showed, it can be applied to simple as well as complex scenarios involving multiple, inter-related classes. We think it is important that a model endures and scales across courses and we believe ours does. We are exploring the usage of its invocation blocks to reason about recursion, stack frames, and multithreading in CS2 and the O/S course.

8. REFERENCES

[1] Cohoon J. and Davidson J. *Java 5.0 Program Design: an introduction to programming and object-oriented design*, McGraw-Hill, 2006.

[2] Gosling, J., Joy, B., Steele, G., Bracha, G. *The Java Language Specification, Third Edition*, Addison-Wesley, 2005.

[3] Holliday, M. and Luginbuhl, D. CS1 Assessment Using Memory Diagrams. *In Proceedings of the SIGCSE Technical Symposium on Computer Science Education*, ACM Press, 2004, 200-204.

[4] Holliday, M. and Luginbuhl, D. Using Memory Diagrams When Teaching Java-Based CS1, *In Proceedings of the 41st Annual ACM Southeast Conference,* ACM Press, 2003, 376-381.

[5] Horstmann C. *Java Concepts*, John Wiley & Sons, 2005.

[6] Wu, C. T. *An Introduction to Object-Oriented Programming with Java*, McGraw-Hill, 2006.