DE GRUYTER
OPEN

STUDIES IN LOGIC, GRAMMAR
AND RHETORIC 40 (53) 2015
DOI: 10.1515/slgr-2015-0014

Adam Drozdek
Duquesne University

OBJECT-ORIENTED PROGRAMMING AND REPRESENTATION OF OBJECTS

Abstract. In this paper, a lesson is drawn from the way class definitions are provided in object-oriented programming. The distinction is introduced between the visible structure given in a class definition and the hidden structure, and then possible connections are indicated between these two structures and the structure of an entity modeled by the class definition.

Keywords: objects-oriented programming, representation of objects, computer science, visible and hidden structures.

Object-oriented programming (OOP) is a paradigm currently used in programming, and it is allowed by certain languages (such as C++) or enforced by them (such as Java). In OOP, data and operations on them are put together (encapsulated) in a class definition used to generate (define) objects of this particular class type. Important and useful as this paradigm is in computer science, it also gives some philosophically interesting insights.

Consider a task of defining a class that allows for processing rectangles (Figure 1). A class can be defined as follows:

```
class Rectangle1 {
   double s1, s2, s3, s4, x1, y1, x2, y2, x3, y3, x4, y4;
   double perimeter() {
      return s1 + s2 + s3 + s4;
   }
   . . . . . . . . . .
}
```

In this class, four pairs of coordinates would be entered by the user, and the lengths of the sides would be computed inside the class (in a constructor), e.g.,

$$s_1 = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

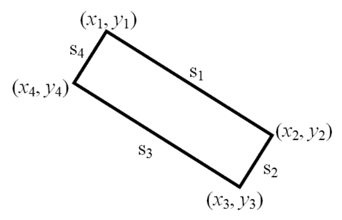

Figure 1

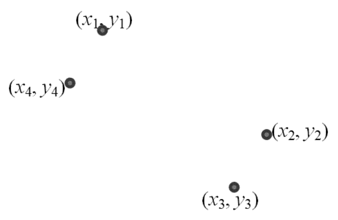

Figure 2

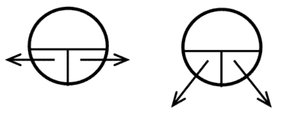
Figure 3

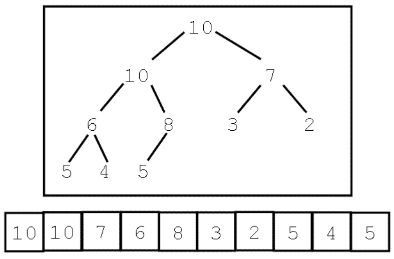

Figure 4

The data members in this class reflect the structure of a rectangle after these data members assure that the two opposite sides are of the same length and that the neighboring sides are at a right angle (which can be accomplished by checking that a diagonal equals $\sqrt{s_1^2 + s_2^2}$. Note that if the goal is to directly reflect the structure of the rectangle, then, keeping only the coordinates, as in:

```
class Rectangle2 {
   double x1, y1, x2, y2, x3, y3, x4, y4;
   . . . . . . . . . .
}
```

would be insufficient, since this would correspond to Figure 2, which gives a very good idea about the rectangle but is not quite it. However, the data members of `Rectangle1` are just twelve real numbers that by themselves can refer to anything, say, twelve temperature measurements. The fact that we can speak here about rectangles is determined by the context, which is reflected in properly chosen names for the class, data members (the names `side1`, etc. would be even better) and operations (procedures/functions/methods) inside the class, and the comments that frequently accompany class definitions. The class name allows us to understand x's and y's as coordinates and s's as side lengths, although, the name of the class itself obviously does not make twelve numbers into a representation of a rectangle. There is an unspoken assumption here that the coordinates are on the Cartesian plane and that the coordinates are connected by straight line intervals.

In many situations, the position of a rectangle is irrelevant; therefore, a definition of a rectangle class can be reduced to this:

```
class Rectangle3 {
   double s1, s2;
   . . . . . . . . .
}
```

By itself, the definition is unstructured as only it can be; thus, the structure is imposed by operations on these two numbers. For instance,

```
class Rectangle3 {
   double s1, s2;
   double perimeter() {
      return 2*(s1+s2);
   }
   double area() {
      return s1*s2;
   }
   double diagonal() {
      return Math.sqrt(s1*s1 + s2*s2);
   }
   . . . . . . . . .
}
```

The way the perimeter is computed indicates that a 4-sided polygon is intended (including polygons with two equal sides that are neighbors); the formula for the area indicates that only parallelograms are intended (where s_1 is the length of one side and s_2 is the height, not the length of the other side); and the formula for a diagonal requires that the parallelogram is a rectangle. The specification of the structure of a rectangle is embedded in the operations; it is, in fact, defined by them. And thus, in another class there can also be only two real numbers, s_1 and s_2, and yet the class could represent isosceles triangles, or complex numbers, or points on a plane.

With class definitions various data structures are introduced in computer science. For example, a node of a structure can be defined as:

```
class Node {
   int info;
   Node r1, r2;
   . . . . . . . . .
}
```

to indicate the fact that one node holds one integer and references to two other nodes. Then, another definition can be used:

```
class Nodes {
   Node start;
   . . . . . . . . .
}
```

to allow for combining nodes together into particular entities. The structure of these entities depends on the operations defined in class `Nodes`, and, normally, there are at least four such operations: insertion, deletion, search, and update. These operations impose a structure onto a collection of interconnecting nodes, and, in the case of `Nodes`, two such structures can be a doubly-linked list and a binary search tree, which is reflected in the interpretation of references in each node, as illustrated in Figure 3.

The rectangle and node examples point to the prominence of operations performed on particular entities. The operations shape them, bring them into particular form; they determine their structure. When carried to the extreme, it is possible that when defining particular entities, their actual structure can be entirely left to operations, and their structural details are determined only after all the operations are well defined. This is an approach used in the definition of abstract data types. An *abstract data type* is defined in terms of operations. The form of the entity itself is determined later in such a way that all (at least, most) operations can be performed efficiently – that is, first of all, quickly – and secondarily, in a minimum amount of space. Consider the definition of a queue, a very important data structure. A queue is simply a waiting line and operations should define our expectations about a waiting line; if it is a waiting line in a grocery store, a newcomer should go to the end of the line; the next person served is at the front of the line; cutting into the middle of the waiting line is not allowed; and no one from inside a line can be served if there are people closer to the front of the line. The expected behavior of a queue can be defined by delineating operations through simple descriptions, as in:

```
interface AbstractQueue<T> {
   void enqueue(T el); // put the element el at the end of the
                          queue;
   T dequeue();        // remove the first element from the queue;
   T firstEl();        // show the first element in the queue;
   void clear();       // clear the queue;
   boolean isEmpty();  // check to see if the queue is empty;
}
```

Descriptions in this interface suggest that a queue should be a one–dimensional structure with a front and an end, like a line interval. The actual structure is fitted in to accommodate the operations. One possibility is using an array as the storage of items, as in:

```
class Queue1 implements AbstractQueue<Object> {
   Object[] storage;
   int firstAvailableCell;
   . . . . . . . . .
}
```

With this implementation, after dequeuing an element from the queue all remaining elements would have to be copied to the preceding cell in the array to reflect the fact that the very first element has been removed. This is potentially a very time–consuming operation, and although the array implementation allows for performing all the operations, at least the dequeuing operation is inefficient and thus it becomes a potential bottleneck.

Another possibility is to use a linked list, which allows for immediate enqueuing and immediate dequeuing:

```
class Queue2<T> implements AbstractQueue<T> {
   LinkedList<T> list = new LinkedList<T>();
   . . . . . . . . .
}
```

This is done at the price of using a significant amount of additional space for reference fields in nodes that constitute the linked list. Yet another possibility is to use a circular array, as in:

```
class Queue3 implements AbstractQueue<Object> {
   Object[] storage;
   int firstEl, lastEl;
   . . . . . . . . .
}
```

In this implementation, no additional space for reference fields would be needed and there is no need to shift items in the array after enqueuing and after dequeuing (Drozdek, 2013, sec. 4.2). In a way, this is the combination of the two first approaches: an array can be viewed as a linked list in which no reference fields need to be maintained, since the neighbor of each cell (that plays the role of a node) is next in the computer memory and can be accessed by properly incrementing an indexes `firstEl` and `lastEl` used to access array cells. Also, this is, as it were, a circular linked list of fixed length, only part of which is occupied by items currently stored on the queue. And yet, the circular array is, physically, not at all circular. It is only its conceptual view in which the first cell of the array is considered the neighbor of the last cell of the array.

It turns out that a circular array is a very poor choice for structuring a queue if priorities are associated with items in the queue. If an array is used as a structure, then enqueueing would be immediate, but dequeuing would require an exhaustive search to find an item with the highest priority and items following that one would have to be moved by one position. If a linked list is used, the exhaustive search is not avoided; if the list is maintained in descending order, enqueuing would still require finding the proper place for a new arrival. A significant speed-up is accomplished when a priority queue is implemented with a particular version of a heap, which is really an array but is viewed as though it were a tree with each node holding an item with a priority larger than or the same as its descendants as illustrated in Figure 4 (Drozdek, 2013, sec. 6.9.1). Note that only the array is physically used, but it is viewed as though it were a tree.

The two queue examples – `Queue3` and a priority queue – indicate that the driving force for specifying a structure of these queues are operations: they, as it were, force the hand of the programmer to choose a particular form of the queue to assure a prompt execution of these operations. And yet, these structures are apparently absent in the definition of `Queue3` and of the priority queue. They are just ways of viewing simpler entities and imposing an order on them from above: a circular array does not exist in computer memory, just a plain array does; a tree-like structure of a heap does not exist either, it is also a plain array. It is just an array in both cases and yet these arrays are viewed in vastly different ways. A plain one-dimensional array is interpreted as a circular array in one case, as a tree in another case; a pretense is made, as it were, that we are really dealing with a genuine circular array or a genuine tree. And if the matter is pushed further, even a plain array is illusory since in the computer memory there are no cells with, say, numbers residing in them, but only an immensely large collection of gates with electrical impulses feeding them; and there are no digits in the computer, a digital device as it is, but only low or high voltages.

An observation has been made that in the object-oriented paradigm, data structures and their behavior "are packaged together as informational objects. Discrete objects are self-contained collections of data structures and computational procedures" (Floridi, 2011, p. 359). This observation has to be amplified with some qualifications. As our examples indicate, we should distinguish between a *visible* or *surface structure* in a class definition, which is explicitly given by data members defined inside the class. We also have a *hidden structure* defined by operations in the class. A class definition models a fragment of reality (rectangles, queues, etc.). What is the relation between the two structures and the real structure of an entity that a class definition intends to model? There are four possibilities. Both the visible structure and the hidden structure directly reflect the structure of an entity, which is possible when the visible and hidden structures are the same. Another possibility is the opposite, when none of them reflects the entity they purport to model. This can be due to inadequate understanding of the entity and thus of its structure, which leads to an imperfect reflection of this structure in a class definition; or, the entity simply does not exist and the class definition refers to nothing. A more interesting situation is when only one of the two structures, visible or hidden, directly maps onto the structure of the modeled entity. If it is a visible structure, then the hidden structure is introduced by the programmer purely for efficiency purposes: to allow for the fast execution of procedures. It is also possible that, for exactly the same reasons, the visible structure of a class has apparently little to do with the structure of the modeled entity but this correspondence is established through hidden structure. These four possibilities are summarized in Table 1 (Yes – the visible/hidden structure closely resembles the structure of an entity that a class definition intends to represent).

In `Rectangle3`, two simple variables and some methods are packaged together to represent rectangles, but the structure – the two variables – is, by itself, useless to represent rectangles, i.e., to convey their rectangular structure. Just an investigation of this structure – the presence of the two real numbers – would give a completely erroneous impression about

Table 1

visible structure	hidden structure	example

Yes	Yes	`Rectangle1`
		`Queue1`
		`Queue2`

Yes	No	`Queue3` priority queue class with heap
No	Yes	`Rectangle2` `Rectangle3`
No	No	a class definition for phlogiston

the nature of a rectangle. The inner, real structure is hidden in the workings of operations on rectangles, i.e., in the definition of methods in `Rectangle3`. The intended structure permeates the entire class definition; the structure cannot be separated from the methods. However, the outward appearance does not have to be deceptive: in the case of a queue implemented with a circular array and a priority queue implemented with a heap, the hidden structure defies the intuitions about what a queue is: whoever heard of a circular waiting line or, worse yet, a waiting line that looks like a tree? However, the plain array fairly closely corresponds to what a queue is.

One sweeping statement suggests that "computer science is distinct from both natural and social science in that *it creates its own subject matter*" (Colburn & Schute, 2010, p. 98) and thus "there is no doubt whatsoever that all object structures within the OO paradigm, modelled or instantiated within an OO application, are through and through human-made entities or artifacts – they could never be mind-independent or external by nature" (McKinlay, 2012, p. 226). However, this does not mean that a programmer has free rein. Certain limitations need to be taken into consideration. A programming language imposes certain limitations: not all languages allow for concurrent programming, not all languages allow for using graphics or graphical user interface. Hardware can also impose some limitations, such as speed and available memory. Most importantly, class definitions are seldom introduced arbitrarily; they are designed as representations of certain entities and the program that includes these class definitions is designed as a discovery tool to reveal some properties of these entities or as a tool allowing us to perform certain operations on these entities. Although "the programmer *prescribes* laws in the realm of the abstract" (Colburn & Schute, 2010, p. 106) (such as the workings of an entity introduced by a class definition), it does not mean that any law can be introduced in any realm.

This may also be an indication that we may encounter a similar problem in other areas of science. In science, a scientist tries to uncover regularities and express them in laws given, say, by differential equations. The scientist tries to see the structure of the investigated domain, if only indirectly. The laws formulated by the scientist are such means. These laws are the manifestation of the workings of the universe, and the structure of the investigated realm is also in these laws reflected. This can be quite complicated and, at times, quite confusing. Some experiments indicate that atoms are particles; some point in the direction of a wave. What is it really? Not an easy question to answer. Models are built in which atoms are shown as particles, and yet the procedures performed on them indicate that there is also a hidden structure allowing for wavelike structure.

A lesson from OOP indicates that hidden does not necessarily mean correct; maybe the surface structure indicated by a certain class definition is more adequate. But the lesson is also that neither a surface nor hidden structure needs to directly correspond to the entities this definition attempts to represent. However, will we ever be certain how it is in a particular case? In OOP, the structure in a class defining a rectangle can be directly compared with our understanding of the rectangle. However, a direct comparison of a model and a structure proposed by it with the noumenal world (or with a "deep structure" of the world (Leplin, 1977, p. 26)) is another problem altogether. Barring the existence of a special kind of intuition that allows one to have an insight into reality unmediated by any cognitive limitations and unfiltered by any theoretical assumptions, a direct comparison between reality and the way scientists capture it in their models is not available. Does the model adequately reflect reality? And if it does, then which part is a better reflection: the visible structure that forms the model or its hidden structure? Or maybe none? The many applications of the heap implementation of the priority queue suggest that the class definition is adequate. But what exactly is adequate: the visible structure or the hidden structure? Empirical confirmation of many models of reality indicates that these models as a whole adequately capture the modeled fragment of reality, but which fragment of the model corresponds to which fragment of the modeled domain – that may not always be easy to resolve, and may not even be possible. Emil du Bois-Reymond's exclamation, *ignoramus et ignorabimus* is not very easy to dismiss if only because "one of the things about which we are most decidedly ignorant is the detailed nature of our ignorance itself" (Rescher, 2006, p. 107).

R E F E R E N C E S

Colburn, T., & Shute, G. (2010). Abstraction, law, and freedom in computer science. In P. Allo (Ed.), *Putting information first: Luciano Floridi and the philosophy of information* (pp. 97–115). Malden: Wiley-Blackwell.

Drozdek, A. (2013). *Data structures and algorithms in Java.* Singapore: Cengage Learning.

Floridi, L. (2011). *The philosophy of information.* Oxford: Oxford University Press.

Leplin, J. (1997). *A novel defense of scientific realism.* New York: Oxford University Press.

McKinlay, S. T. (2012) The Floridian notion of the information object. In H. Demir (Ed.), *Luciano Floridi's philosophy of technology critical reflections* (pp. 223–241). Dordrecht: Springer.

Rescher, N. (2006). *Philosophical dialectics: An essay on metaphilosophy.* Ithaca: State University of New York Press.