State- and Data-Oriented Models in Program Visualization

Michael Weigend[1]

[1] Westfälische Wilhelms-Universität Münster, Germany, michael.weigend@uni-muenster.de

Abstract: Data can be represented by quasi-material entities or by immaterial states. This paper investigates some facets of state-oriented and data-oriented intuitive models, which students use when they interpret or create computer programs. Meaningful states in a program visualization can reduce the cognitive load and thus facilitate the understanding of the algorithmic idea.

Keywords: Algorithms, Computer Science, Modeling, Thinking, Didactics

1. Introduction

Intuitive models are self-evident, Gestalt-like mental concepts about the world [3, 4, 5, 11]. For example, we know for sure that an object might change its state and still continues to be the same entity. When you form a piece of paper to a ball it looks different but it is still the same piece of paper. Intuitive models are persistent (we never forget them) and may be unconscious. But even then they influence our thinking and bear the risk of tacit misconceptions. People apply intuitive models, when they try to use, understand, explain, verify or develop computer programs [11]. They are explicated (and communicated) in many different ways: verbally in program documentation or comments explaining the algorithmic idea of a function, visually in formal diagrams, drawings or movies.

Novices use intuitive models as a vehicle to get intellectual access to new programming concepts. New knowledge is built using already familiar concepts. But also professionals apply intuitive models in software design patterns [6] and project metaphors [2] as a means to cope with complexity, since they can embody a lot of information in one holistic Gestalt. In this paper I discuss some drama-like visualization of programs adopting state-oriented and data-oriented models of information processing. They are mostly taken from a collection of web-based game-like applications including the Python Visual Sandbox (PVS) and the V-Quiz that I have been developing since 2005. These systems have been designed as interactive media for informatics education but also as research tools to

investigate mental models and misconceptions in the context of program comprehension [10, 11].

2. How to Represent Data

This section is about different data representations in program visualizations and interdependencies between them and intuitive models of functions.

You can imagine a value - say a number - as a quasi-material entity of its own or as a state of some object. Figure 1 shows screenshots of two animations, which visualize the execution of the following two assignments (Python):

```
a = 3
b = a
```

In the first model the value is represented by a data-entity, a card with the number 3 written on it. During the execution of the first statement a box labeled with "a" appears and the card is put into it. Then a copy of this card emerges and moves into a second box. The second model visualizes the number 3 by an immaterial state of an object: the position of a disc with numbers. 154 high school students from Germany and Hong Kong (average age 17 years, 34 female, 120 male) saw these movies (among others) and had to decide for each, whether it is an appropriate model or not. 133 (86.4%) of them accepted the first model with moving data entities, but only 100 (64.9%) the second [11]. Why? A problem of the state-oriented model in this context is, that it does not visualize the transport of information from `a` to `b`, which is an important facet of the meaning of the two assignments.

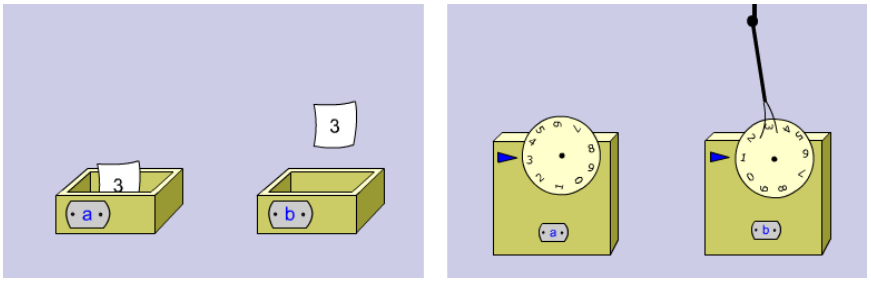

Figure 1 Screenshots from Flash movies visualizing assignments [11].

State-oriented representations of information imply a different way of algorithmic thinking than data entities. Like any material thing a data entity can be created (for example by copying) or destroyed, it can be moved from a source to a target, it can be possessed and stored in a container.

On the other hand a state is an immaterial aspect of an existing entity. You cannot create or destroy a state. An entity is always in some state, but its state may change.

The concept of data entities fits to the factory model of function calls. This intuition models a function as an entity with input and output facilities. When the function is called, it incorporates data entities (input) like a factory, which is supplied with some raw materials. The function then processes these data, produces new data entities and outputs them to its environment. Computer scientists' jargon and even some keywords of programming languages support this notion. For instance, a function is said to "return" a value.

Nevertheless sometimes students visualize a function call by a device, which does not incorporate and output data entities but changes the state of an object in its environment. Figure 2 shows a drawing created by a high school student (17), which visualizes the Java statement

```
b = a.toUpperCase();
```

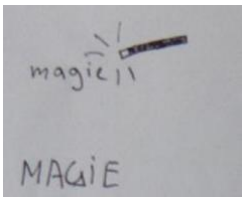

Figure 2 A student's visualization of a function call causing a change of state [11].

A word written in small letters is touched by a magic stick and changes to the same word written in capital letters. It is still the same entity (it continues to exist), it just looks different now. It has changed its state. If you consider the semantics of Java, this is a clear misconception. The method call `a.toUpperCase()` does not change the object a. Instead a new String object is created and returned. Note that the name of the method `toUpperCase()` implies a state concept of letters: A letter must be in one of the states "lower case" or "upper case".

I call this state based intuition of functions the tool model. A tool does not incorporate entities but it changes states of entities in its environment. Graphical editors commonly adopt the tool model for visualizing function calls. The user can select a tool (function), then the cursor changes its look, now representing the selected function. For instance it might look like a paint bucket. The user applies the tool on some entity on the screen by clicking on it.

3. Meaningful States and Reduction of Cognitive Load

This section starts with some arguments, why state-oriented models are sometimes easier to understand than data-oriented visualizations. In the second part it is

discussed in which way a state-oriented model (explicating an algorithmic idea) might inspire a program development.

A state can be a rich concept, wrapping many aspects of a system in one Gestalt. A meaningful state is embedded in a semantic context. It is associated to certain properties of an entity, activities and other states, to which a direct transition is possible. For example, when you are sick, you usually feel weak and look pale (properties). You are not supposed to work, but stay at home, consult the doctor and take some medication (activities). Typical connected states are "healthy" or "dead". The use of meaningful states implies a reduction of complexity and cognitive load [9], because algorithmic details can be ignored on a high level, but can be reconstructed from the meaning of the state later, when necessary. In program visualization meaningful states are important, when an analogy is used to illustrate the idea of an algorithm. As an example consider a Java class that schedules the travel of an elevator. The following listing shows some fragments of the class definition.

```
class Scheduler {
 boolean[] orders =
           [false, false, false, false];
 ...
 int get_order (int floor, boolean up)  {
    int i = floor;
    if (up) {
       while (i < (orders.length)) {
         if(orders[i]) {return i;}
         else {i++;}
       }
       i--;
       while (i >= 0) {
         if(orders[i]) {return i;}
         else {i--;}
       }
    }
   ...
```

Objects of the class `Scheduler` have an attribute called `orders`. It is a Boolean array representing the floors the elevator has to go to. When someone presses a button to order the elevator to floor number `n`, `orders[n]` is set to `true`. The method `get_order()` returns the number of the next floor the elevator has to visit (if the elevator has been ordered at all). The first argument `floor` tells the present position of the elevator and the second indicates whether it is on the way up (`true`) or down (`false`). The method is supposed to return the number of the next requested floor in the direction to which the elevator is currently moving. If the elevator has not been ordered along that way at all, the number of the next requested floor in the other direction is returned.

Figure 4 shows screenshots from a movie illustrating the algorithmic idea of the Java program. (It is reduced to the case that `up` is `true`.) The Boolean array is visualised by a block with holes (representing the value false) and red Cylinders (representing `true`). The object `s` receives the message `s.get_orders(2, true)` visualised by a speaking bubble. First a pin labelled with "floor" moves to position 2 of the block. Thus the value of the first argument (2) is represented by a state. A little car (a blue cuboid with wheels) is put to the position of this pin. It moves to the right until it has reached a ramp (figure 4, upper picture). This is a critical state, covering at least two algorithmic facets in one Gestalt:

- The car is beyond the array and cannot continue its motion to the right.
- It will now move to the opposite direction.

In fact in the animation the car moves from the ramp backwards to the left until it hits the cylinder (figure 4 lower screenshot). This is another state with rich algorithmic meaning:

- The search (motion of the car) has been stopped permanently.
- The number of the next requested floor has been found. It is represented by the present position of the car.

Note that all these visible activities are state changes and do not include processing explicit data entities.

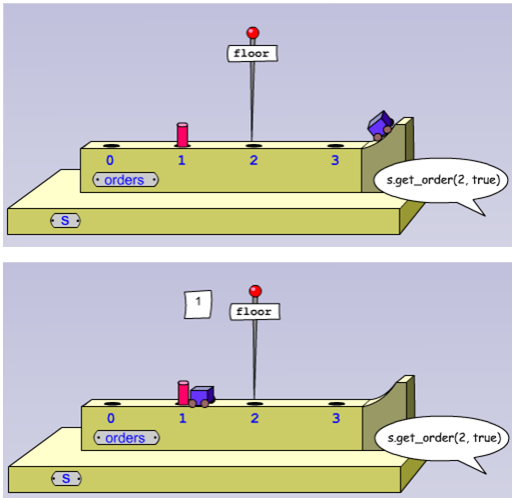

Figure 3 Screenshots from Flash-movies visualizing iterations

Let us suppose Anna has a state-oriented model like the one shown in figure 3 in her mind, before she starts to write the Java program. This model is intuitive to her, which means she is absolutely certain, that it is logically correct. Otherwise she would not use it as starting point. Fishbein calls this kind of model (or to be more specific: this role of a model) anticipatory intuition.

What kind of cognitive activity is necessary on the way from the model to the program? And what are difficulties, Anna has to cope with? Roughly speaking she has to "read" the model in some way and search for "hints" which help to construct fragments of code and put them together. As already mentioned, the track with holes and cylinders represents a Boolean array. The moving car corresponds to a variable (in the Java program named i) storing a number. The continuous movement must be seen as a simplified representation of a discontinuous movement, a series of atomic steps from hole to hole and – at the end of the array – to the ramp. In this view a step to the right (change of state) corresponds to the incrementation of `i` (Java statement: `i++`). Since it is done several times, a control structure like a while-loop should be used.

As long as the car is on the track, its position represents a certain index of the array. But when it is on the ramp (beyond the track), its position represents a number *greater then the largest index of the array*. Note that this is a more meaningful (and more abstract) concept than any explicit value (like 4) that might occur in some concrete program run. It should make it easier to find an appropriate condition for the while loop.

Bumping against a cylinder means that the job is done and the value indicated by the current position of the car is the result of the function call and must be returned. This corresponds to the Java-statement

```
if(orders[i]) {return i;}
```

These were just a few thoughts that might emerge during a program development inspired by an analogy. They imply a transformation of state-related knowledge to formal program text. The findings mentioned in section 2 suggest that people (at least novices) tend to think of data entities (instead of states), when they interpret program statements like assignments. This indicates a possible cognitive barrier a programmer has to overcome.

4. Mixing Data and State Concepts

This section gives an example illustrating that it is sometimes efficient to mix data-oriented and state-oriented visualisations. Consider a Python program that sorts a list of four numbers according to the straight selection sort algorithm.

```
s = [10, 4, 1, 3]
for i in range(len(s) - 1):
  for j in range(i+1, len(s)):
     if s[i] > s[j]:                    #1
        s[i], s[j] = s[j], s[i]         #2
```

There are two nested for-loops using the variables i and j, which get successively values out of the intervals `[0, 1, … length(s) - 2]` resp. `[i+1, … length(s) - 1]`. In line #1 the two elements `s[i]` and `s[j]` are compared. If the first is larger, they exchange their positions in the list (#2).

Figure 5 shows a screenshots from a Flash-movie that visualizes the execution of this program. The list is represented by a box with compartments containing cards with numbers written on them.

The screenshot in figure 5 corresponds to line #1 of the program. Two elements named `s[i]` and `s[j]` are selected from the list and are going to be compared. But the names `s[i]` and `s[j]` are not mentioned in the animation. Instead two cards are pulled a little bit out of their compartments to indicate that these two are now of interest. Thus the indices of two selected items are represented by a state "pulled out". During the sorting process the first part of the list $[s_0, \ldots, s_{i-1}]$ is not changed any more, since it is already sorted. In the animation this part is covered and "protected" by a glass box labeled "ok". This box is getting bigger and bigger until it covers the whole container at the end.

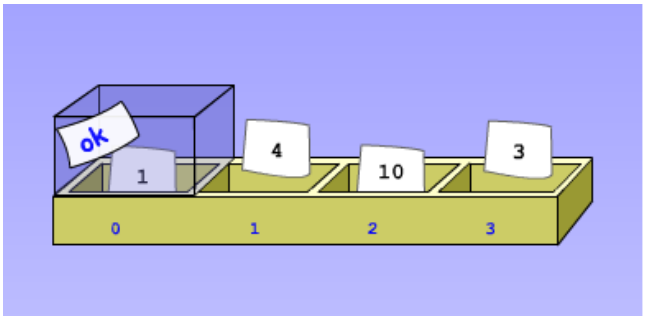

Figure 4 A visualization of the straight selection sorting algorithm using states

This Flash movie belongs to a collection of four different animations, which have been shown to 16 students (average age 18 years, 3 female, 13 male, 1 university student 15 high school students). 9 out of them decided they would use this model to explain the Python program to someone else [11]. What is the secret of its success? Two state-oriented features might be important:

(1) The marking of two list elements by pulling them out of their compartments focuses the attention to the algorithmic idea. Empirical findings in multimedia instruction research indicate that integrated presentation of information (for instance text fragments integrated in a diagram) - instead of splitting it - facilitates understanding since this reduces the necessary cognitive load [1]. In contrast, another animation (see figure 6), which used separated entities for the indices, was preferred only by 2 out of 16 students. To understand this visualization, a mental integration of three different visible sources is required. It should be mentioned, that the visualization of the two "stepper" variables `i` and `j` was designed as proposed by Sajaniemi [8]. In the movie you can see all possible values of `i` and `j` on stripes, which is supposed to clarify their roles as steppers in the program. Nevertheless the students preferred the state-oriented representation, although it

has a greater distance to the program since it does not explicate the variables `i` and `j`.

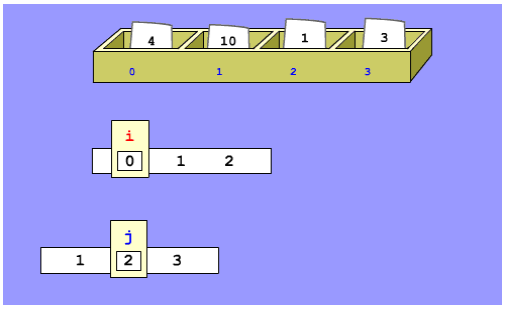

Figure 5 A visualization of straight selection sorting using data-entities for indices

(2) The glass box visualizes the state "sorted" of the covered part of the list. This aspect of the algorithm is not explicated in the program text. Note that there are only a few states with significant meanings for lists, including "sorted" and "empty". But there is no specific word for the specific state of a list that looks like `[1, 4, 10, 3]`. The state "sorted" is also associated to activity: "You must not touch this part of the list any more, it is already sorted".

5. Conclusions

In this section I draw some general conclusions from the previously discussed examples and discuss educational implications.

An intuitive model based on meaningful states might help developing a program. But the programmer must be able to extract algorithmic information from the meaning of a state considering properties, activities and related states. The analysis of relevant properties of an entity in a certain state might lead to the introduction of variables or classes. Activities, which are associated to a certain state, correspond to tests, whether values are within certain ranges or not, and data processing operations.

Besides inspiring to use certain programming language constructs the anticipatory intuitive model can also be used to check the logical correctness of the emerging program. The developer makes sure that the state transitions of the model are also performed by the program.

Visualizations containing intuitive models should be presented in textbooks, classroom discussions or multimedia applications. But they also can be created by the students themselves – for instance in role plays, brick movies or story boards. The purpose can be (1) to find algorithmic ideas for a software project or (2) to get an individual understanding of a given program.

Again: An intuitive model must be simple and meaningful. That implies that automatic visualization tools like Jeliot [7] have only limited usability for program comprehension and creation, because they are incapable of abstraction. Each activity of the running program is shown on the screen and it might be exhausting to watch a tracing. It is impossible for such a system to create a meaningful whole from some lines of code. This requires imagination and is still a very human activity.

References

1. Ayres, P. and Sweller, J.: The Split-Attention Principle in Multimedia Learning. In The Cambridge Handbook of Multimedia Learning, Richard E. Mayer, Ed. Cambridge University Press, pp. 135 – 146 (2005).
2. Beck, K.: Extreme Programming Explained. Boston, Addison Wesley (1999).
3. diSessa, A. A.: Knowledge in Pieces. In Constructivism in the Computer Age. George Forman and Peter B. Pufall, Eds. Lawrence Erlbaum, Hillsdale, pp. 49–70 (1988).
4. diSessa, A. A.: Changing Minds. Computers, Learning, and Literacy. MIT Press, Cambridge, Massachusetts (2001).
5. Fischbein, E.: Intuition in Science and Mathematics. Reidel, Dordrecht Boston Lancaster Tokio (1987).
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995): Design Patterns – Elements of Resuable Object-Oriented Software. Addison Wesley, Reading, MA (1995).
7. Moreno, A. and Myller, N.: Producing an Educationally Effective and Usable Tool for Learning, The Case of the Jeliot Family. Proceedings of the International Conference on Networked e-learning for European Universities, Granada, Spain (2003).
8. Sajaniemi, J.: Visualizing Roles of Variables to Novice Programmers. Proceedings PPIG 14 (2002), Brunel University (2002).
9. Paas, F.; Renkl, A.; Sweller J.: Cognitive Load Theory and Instructional Design: Recent Developments. In Educational Psychologist, 38 (1), Lawrence Erlbaum Associates, pp. 1–4 (2003).
10. Weigend, M.: Design of web-based educational games for informatics classes – some insights from workshops with the Python Visual Sandbox. In Workshop GML – Grundfragen multimedialer Lehre, Potsdam (2006).
11. Weigend, M.: Intuitive Modelle der Informatik. Universitätsverlag Potsdam, Potsdam (2007).