Do Illustrations help Understanding Algorithms?

Michael Weigend, *mw@creative-informatics.de*
Holzkamp-Gesamtschule Witten, Willy-Brandt-Str. 2, 58252 Witten, Germany

Abstract
This paper discusses the discovery learning of algorithms. It presents the results of a study performed with 51 high school students from grade 11 and 12. The participants had to understand (discover), apply and verbalize algorithms for decoding sequences that represent digital images. In the learning phase they had to interpret an example and five metaphoric illustrations of the decoding procedure. In the production phase they had to verbalize the learned algorithm. The goal of the study was to explore the influence of visualizations on algorithm understanding.

Keywords
Informatics education algorithm, scaffold, metaphor, digital image.

INTRODUCTION
How can we teach programming? In introductory programming classes at a university it usually works like this: The students learn some syntax and semantics of a programming language and then apply recently learned language elements, for program development. For example, the lecturer explains the pattern of loops and then gives examples (e.g Grimson & Guttag, 2008). This is direct instruction and a deductive dramaturgy, moving from abstract principles to concrete applications.

According to the constructivist learning model, fundamental concepts should not just be told but should be discovered by the students themselves. Learning is seen as active construction of knowledge.

Exploring versus constructing
Seymour Papert suggests students should be allowed to construct interesting artifacts within a LOGO environment. During the development process they discover – accidently – programming concepts ("powerful ideas") on their own, ideally without any external intervention. This is basically the idea of constructionism (Papert, 2000). The motto is learning by designing. In a constructive project the primary goal (at least from the student's perspective) is to create some attractive artifact (say a new computer game), which is shared with the community at the end. Sharing - even worldwide publishing - is an important facet of modern constructionism. Scratch (http://www.scratch.mit.edu) – the well known visual programming environment in the tradition of LOGO – makes publishing easy by offering a special button for uploading the project to the Scratch website. All Scratch artifacts can be judged or used as a source of inspiration by everybody in the world (Monroy-Hernández & Resnick 2008).

Traditional discovery learning focuses on exploring a domain. This approach is slightly different from constructionism:

- The explicit goal of the learner's discovery activities is to gain knowledge about the explored domain. The children observe, experiment and document. They do not invent and create in the first instance.
- In contrast to the constructionist idea of creating persisting things, discovery learning emphasizes the moment and real life experience. The idea of "hands on" and "outward bound" is not only to give the students some freedom to discover scientific rules and principles, but also to provide rich sensory and emotional experiences in unusual and exciting environments. It is dangerous to

walk into a rook searching for animals. You might slip and get wet. Possibly you have overcome revulsion and fear before you touch a snail or a worm. In addition to individual learning, the participants also share an exciting social experience.

Exploring programs

In computer science, exploring activities may be related (a) to computer programs or (b) to observed activity in some real domain. Let me consider program exploration first. People read programs and try to find out how they work for several reasons:

- To localize a logical error.
- To improve a program, make it more efficient or add features.
- To steal an idea and use it in a different project.

The latter is part of the educational concept of Scratch: Students read Scratch scripts and try to understand them because they want to create something similar and need to know how to do it. In some way exploring a program is similar to scientific discovery activity. The programmer develops hypotheses and conducts experiments to prove or falsify them.

A fundamental difference to natural science is that the domain of informatics is not physical. Well, there is computer hardware you can see and touch. But the informatics is hidden inside. Basically it is a world of abstract ideas that is there to discover. For example, opening a digital camera and observing its physical parts in action does not help our understanding of LZW-compression and JPEG.

Block oriented programming languages like Scratch and Starlogo TNG (Klopfer) can be considered as an attempt to add sensuality to program text. You can touch a Scratch block and move it on the screen. Sometimes the blocks even make a noise when you click on them.

Blocks carry pieces of program code. So far Scratch is a textual programming language. But additionally Scratch blocks have colour (indicating a category of commands) and they have a geometrical form giving hints which other blocks they can be combined with. For instance the block “When green flag is clicked” must be at the beginning of a script. No other block will stick on top of this block. Since syntax errors cannot occur and the possibilities of putting blocks together are reduced, Scratch supports individual experimentation and discovery.

Further attempts to connect formal program text to sensuality are program visualization tools. Jelliot visualizes (otherwise hidden) changes of state that take place during a program run (Moreno & Miller, 2003). Note that these approaches focus on understanding the mechanics of program execution in the first instance, not on algorithmic ideas.

Discovering Algorithms

A program represents all details of an algorithm. It is impossible to memorize a whole program written in a formal programming language. But it is possible to understand and memorize the basic *idea* of an algorithm. Algorithms may be more abstract, focus on the most important facets and ignore details. Much of the informatics knowledge members of a modern society are supposed to be familiar with is found in algorithmic ideas:

- “What is the idea of data compression?”
- “How are digital images transported from A to B?”

Let me sketch three approaches for discovering algorithms in the class room:

(1) Observe a real life activity and discover an algorithm. The activity should be something mysterious which bears a secret (that is there to discover), like a magical trick, say a card trick. Students watch the magician manipulate the cards several times, find a common pattern of activity and copy the trick.

(2) Observe traces or example results of some activity and discover an algorithm. A typical example of this type is a series of pictures or numbers that continue in a progression (like the Fibonacci numbers or Koch snowflakes). In the study presented in the next section the participants had to discover an encoding/ decoding algorithm for digital images by studying a given example (code sequence and corresponding image).

(3) Interact with a multimedia system and discover an underlying algorithm. Figure 1 shows two screenshots from an online editor for black and white pictures. When you click on a square, which does not already have a color, all squares from the last black or white pixel to the selected square are colored. The color is alternating black and white. On the right hand side you see a series of number representing the picture. 23 represents the first 23 white pixels, 4 represents the following 4 black pixels and so on. The user must understand the mechanics of the system (including the encoding algorithm), when she or he wants to design a picture.

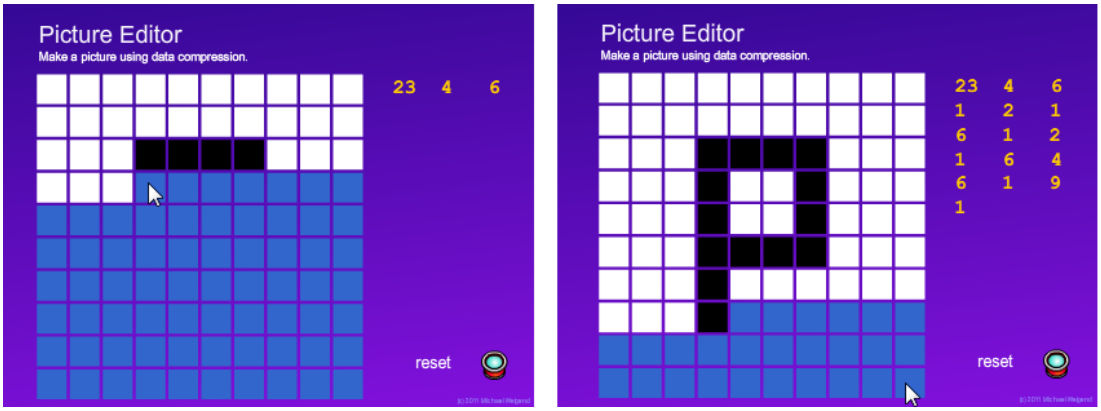

Figure 1: Screenshot from an interactive picture editor,
http:// www.c-park.org/informatics

Visual Scaffolds

The activities I have just mentioned are called “pure discovery”, since there is no external help to solve the problem and discover the algorithm. Pure discovery might be a real interesting challenge to students. But on the other hand there exists evidence that completely free exploration does not necessarily lead to significant knowledge growth (Meyer 2004), and discovery learning is less efficient than direct instruction (Klahr & Nigam 2004). An analysis of 536 Scratch projects developed by young people in American Computer Clubhouses, showed that “variables, random numbers or Boolean logic were only used in few projects; these are concepts that are not easily discovered on one’s own.” (Kafai et al. 2009, p. 140)

It seems that sometimes learners need external support, when discovering new knowledge. Wood, Bruner & Ross (1976) were the first to use the term “scaffolding” to describe tutorial assistance for problem solving. “This scaffolding consists essentially of ‘controlling’ those elements of the task that are initially beyond the learner’s capacity thus permitting him to concentrate upon and complete only those

elements that are within his range of competence." (Wood, Bruner & Ross, 1976, p.90) Scaffolding comprises many different activities, which can be divided into two categories:

(1) Process-oriented interventions, like giving hints about where to find additional information (see for instance: de Jong, 1998, 2006).

(2) Content-oriented interventions, like demonstrating or explaining a part of the solution, which has turned out to be too difficult.

The problem of content-oriented interventions is that they might destroy the excitement of serendipity. The learners might lose the sense of discovering knowledge on their own. One possible way of scaffolding whilst still keeping the task interesting and challenging, is to provide images with a metaphoric description of the algorithm.

In cognitive linguistics, metaphors are considered as cross-domain mappings, projecting the structure of a (familiar) source domain onto a target domain (Lakoff & Nuñez 1997). In elementary maths education so called "grounding metaphors" are used for teaching and learning arithmetic. Children may play with collections of beads, which represent numbers ("arithmetic is object collection"). Putting together two collections means adding the corresponding numbers. In education, metaphors are used not just as figures of speech to make a text more interesting but to facilitate comprehension. Note that a metaphor is only of use if the source concept is intuitive. In this case the learner can apply already existing (intuitive) knowledge to new domains (see Weigend, 2007).

In informatics, metaphors are more than a vehicle for learning. They are used all the time during the development and documentation of software systems. Some metaphors (like data storage, data flow, or loop) are part of professional language and even formal languages.

In each project, developers invent new metaphors to explain structures and describe activity patterns on different levels of abstraction. Project metaphors in Extreme Programming (Beck) or Design Patterns (Gamma et al, 1995) are examples of high level metaphors. Phrases in comment blocks such as "this function accepts ... and it returns ..." explain activity on a lower level, elaborating the details of a software system. Part of the work of a programmer is to read and understand metaphors, create new metaphors and use them for creating new algorithms.

Back to metaphoric illustrations as visual scaffolds. In a social situation, when a group tries to discover an algorithm from some traces of activity, an illustration might have two kinds of impact on the exploration process:

(1) It helps the individual recipient to understand the algorithm directly (by interpreting the metaphor).

(2) It helps indirectly by provoking discussions about its meaning among the group members. Talking induces verbalizing and explicating intellectual content which would remain implicit otherwise. This is very important, when people are supposed to write programs.

A metaphoric visualization may not comprise a whole algorithm but focus on one facet. This facet may be the crucial hint which opens the way to understanding.

The study I am going to present in the remainder of this paper focuses on the question: What kind of metaphoric illustrations are helpful for understanding algorithms in the field of image encoding?

DESIGN OF THE STUDY

In winter 2011, I presented a couple of workshops for a total of 51 students (grade 11 and 12, average age 17.4 years, 17 boys, 34 girls) at a German high school. Most of the participants (34) had never taken CS classes.

Each workshop started with a short discussion about how images are stored in a computer memory and how they are transported. To make it as simple as possible we focused on square black-and-white images with 10 by 10 pixels. A simple technique is to represent each pixel by the value 1 (black) or 0 (white). Obviously this representation requires a lot of storage. Considering simple images like a completely black square, the question arises whether there are smarter representations that need less space.

In the next phase the group was divided in two subgroups. Each subgroup was given a working sheet like in figure 2. The task was to figure out how to decode a series of numbers, representing a digital image. The worksheets consisted of three parts:

- An example of a code sequence and the corresponding digital image consisting of 4 x 4 squares.
- Five pictures illustrating the construction of an image from the code sequence
- Instructions.

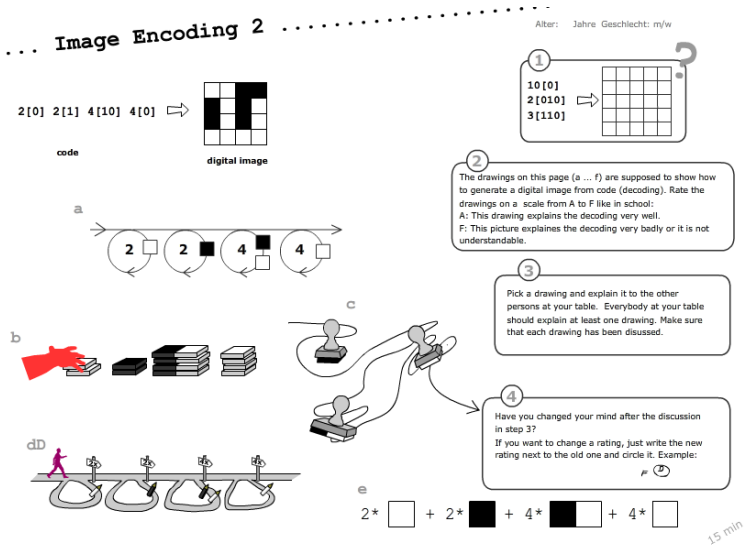

Figure 2: Worksheet for phase one.

The two subgroups were assigned different encoding/decoding functions. The following pseudocode listings (Python style) describe the ideas of the algorithms (examples are in figure 3).

```
decoding_1 (s):
  color = false
  for all x in s:
    if color: print x times a black square
```

```
        else: print x times a white square
        color = not color

    decoding_2 (s):
      for all (x, pattern) in s:
        repeat x times:
          for y in pattern:
            if y == 0: print white square
            else: print black square
```

The two groups were separated; the tables they were sitting at were placed in different areas of the classroom in order to prevent communication between them. The first algorithm is used in the activity “Colour by Number” in “Computer Science Unplugged” (Bell et al. 2005). A sequence of numbers represents alternating the white and black pixels of a picture. The second algorithm can be seen as one facet of the Lempel-Ziv-Welch- algorithm for data compression. It represents the idea that a string (of Pixels) should be regarded as a sequence of patterns.

The students had to perform four steps. In the first step they had to figure out the decoding algorithm by analysing the example and the illustrations, and then decode a different sequence applying this algorithm.

In the second step they were asked to rate the illustrations on a scale from A (very good) to F (very bad). In step three they had to explain the drawings to each other. The pictures were meant to serve as scaffolds, and probably the students used at least some of them, when they figured out the decoding procedure.

Step 3 just guaranteed (to some extent) that the participants elaborated all five illustrations. In step four they had to rate the illustrations again.

In phase 2, the students wrote a decoding algorithm describing how to construct a digital image from a string. They got a worksheet with instructions and some graphical elements structuring the process (figure 2). Additionally they were given a card with some language hints. The algorithm was supposed to be understandable to everybody. The students created a digital image and encoded it, applying the algorithm. They divided the working sheet in two parts. The right hand part (algorithm sheet) was used in phase 3.

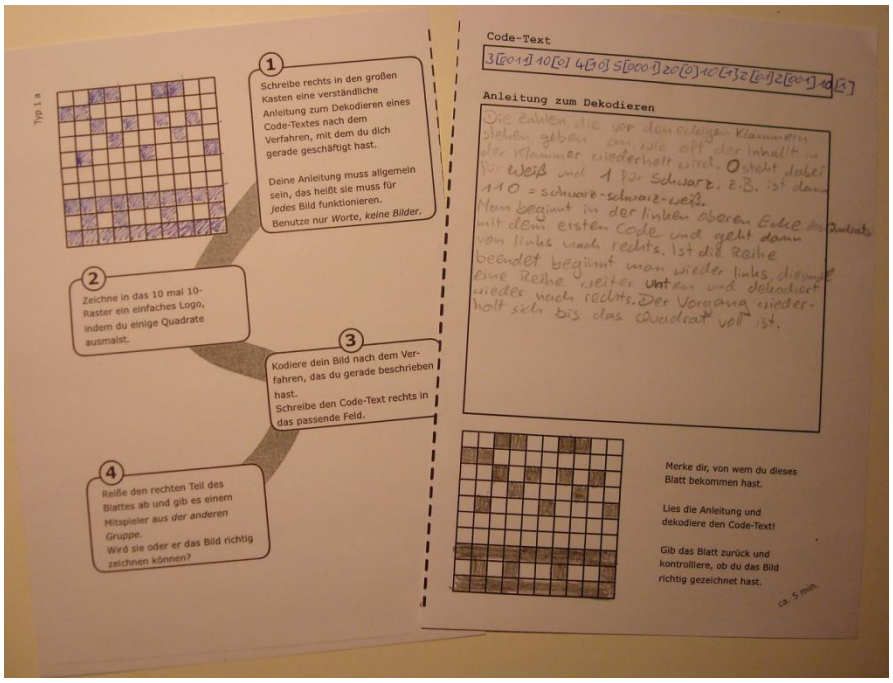

Figure 3: The two parts of a worksheet after phase 3. On the right hand side there is a code sequence on top, in the middle a decoding algorithm and below a class mate's (correct) solution, who had interpreted the algorithm.

In phase 3, students from group A exchanged algorithm sheets with students from group B. So when Lisa got a sheet from Tom, she did not already know the algorithm, and was forced to read and understand Tom's text in order to decode the code sequence. The page contained a code sequence on top, an algorithm in the middle and a grid which the recipient could use for reconstructing the image while interpreting the algorithm. After decoding, they had the opportunity to compare their results with the original images and discuss possible problems.

RESULTS

Figure 3 shows the example decodings and illustrations from both worksheets the two groups of students used in phase 1. The illustrations contain metaphors for certain aspects of the algorithm, which - of cause - need not necessarily be recognized by the recipients and which may be verbalized in different ways. However, when a student rates an illustration highly, it should provide her or him with *some* meaning.

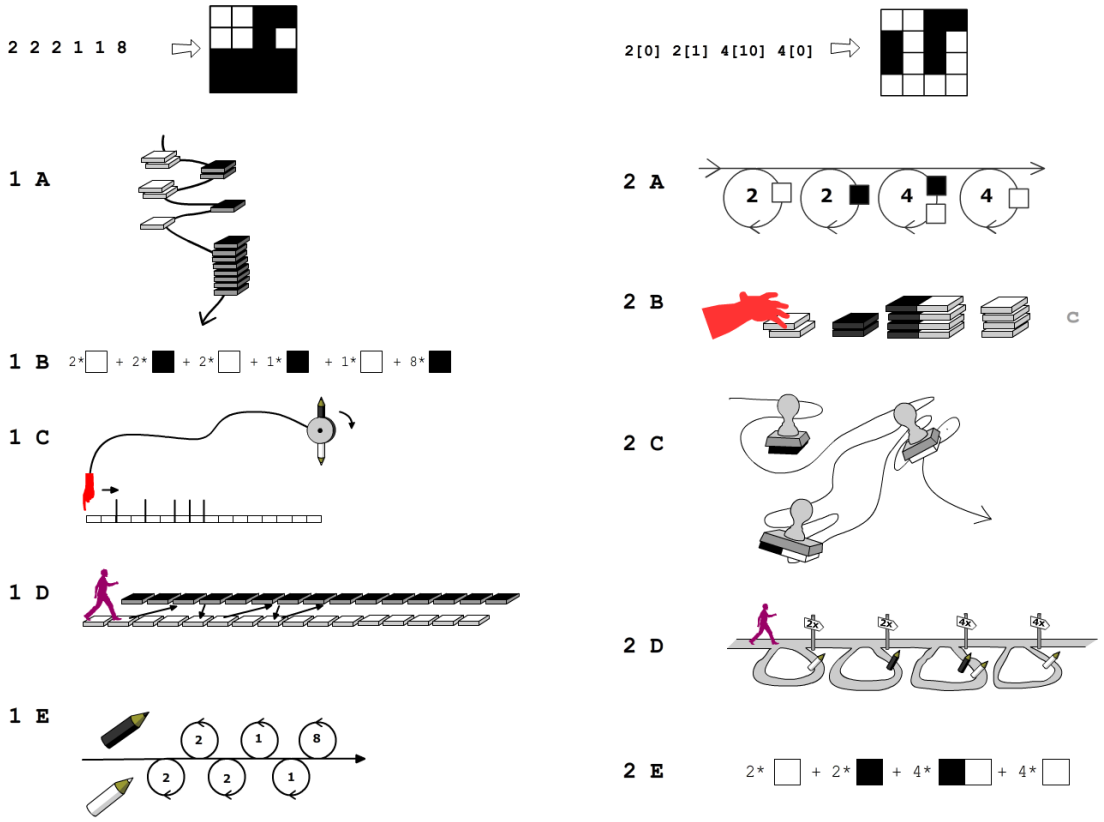

Figure 4: Example decodings (top) and metaphoric illustrations of the decoding algorithm.

Some metaphors are quite general and can be applied to both algorithms, including the following:

(1) A digital image is a “sum” of “products” (1B and 2E).
(2) Constructing a digital image is laying different types of tiles (1A, 1B, 2B, 2C, 2E).
(3) Constructing a digital image is colouring a grid with a pencil (1C, 1E and 2D).
(4) Performing activities in a certain order is moving along a path (1A, 1C, 1D, 1E, 2A, 2C, 2D).
(5) Repetition is going through a loop several times (2A, 2D).

Other metaphors are quite specific:

(6) Changing the colour is turning a two-colour-pencil (1C).
(7) Changing the current colour for pixels is changing the way (1D).
(8) Using alternating colours is walking zigzag (1A)
(9) Evaluating an expressing in brackets is constructing a sequence of black and white tiles (2b, 2E).
(10) Generating a sequence of black and white squares is stamping (2C).

Each picture is a holistic entity, some kind of consistent story, containing several metaphors. This means that different parts of the picture can be associated to different parts of an algorithm. For example pictures 1d and 2d use the metaphor “executing a sequence of commands is moving along a path”, “Repetition is moving in a loop” and “Constructing a digital image is colouring a grid with a pencil.”

Illustration	1 (A)	2 (B)	3 (C)	4 (D)	5 (E)	6 (F)	Avg. (SD)
Walk and lay tiles 1 A	5	17	2	0	1	0	2.00 (0.82)
Sum of products 1 B	14	9	1	1	0	0	1.56 (0.77)
Two-colour-pencil 1C	0	3	4	7	3	8	4.36 (1.41)
Walk on tiles 1D	1	3	6	7	7	1	3.76 (1.23)
Pencils and loops 1E	1	8	8	5	2	1	3.08 (1.19)
Tiles and loops 2A	4	11	6	3	0	2	2.62 (1.33)
Stacks of tiles 2B	8	6	3	5	4	0	2.65 (1.50)
Stamps 2C	0	0	2	0	4	18	5.58 (0.88)
Loops and pencils 2D	4	6	10	2	3	1	2.88 (1.34)
Sum of products 2E	13	10	1	0	0	0	1.50 (0.59)
Sum of all ratings	50	73	43	30	24	31	

Table 1: Rating of illustrations on a scale from 1 (A) to 6 (F).

The students had to discuss, judge and rate the images. Table 1 shows the results (final rating after the discussion). The students clearly preferred the two structure-oriented illustrations 1B and 2E, which describe the code sequence as a mathematical term. Note that the operator * has a non-arithmetical meaning here. It is an extension of concatenation. Similar to Python the term

n * x

indicates that the pixel sequence *x* has to be repeated *n* times. The students had no Python experience. They interpreted the illustration intuitively. I call these illustrations structure-oriented because they contain no explicit visualisation of activity. By contrast, pictures 1D and 2D show a walking person, clearly indicating sequential activity.

Of interest are those images whose ratings have large standard deviations (SD). Some students found them useful and some did not. Consider image 1C showing a two-colour-pencil and a hand. The idea is that the hand moves along a path of pins. When touching a pin the pencil switches the colour. This metaphor represents an essential idea of the algorithm. A technical implementation of the algorithm needs a (Boolean) variable storing the current pixel colour. After reading the next number of the code sequence the value of this variable changes. This image received a poor average rating. Yet three students obviously got the idea and rated it a 2 (B). Such diversity of knowledge provokes debate, since at least *some* students can explain the meaning of a metaphor to the others. Additionally, different opinions on the correctness and usability of illustrations are a social motivation for debate.

Did the illustrations influence the algorithm construction in phase 2? In fact not a single student wrote an algorithm with nested loops as in the two pseudocode listings in the previous section. However, 22 out of 51 students managed to verbalize the algorithm in a way that was understandable and executable to the other students. Probably the algorithms could be understood because the readers used additional knowledge, which they had obtained in phase 1. They already knew that the code sequence represented a picture that was to be reconstructed in a 100-pixel-grid.

Relatively few students used the example phrases, which were given to them as an aid (see table 2).

Algorithm 1, n = 25	
Phrase	Number of students
If ... then ...	3
Put your finger on the first number ...	6
Put your finger on the next number ...	2
This number I call x ...	4
Do ... x times ...	2
while ... do ...	0
Algorithm 2, n = 26	
An item consists of a number and an expression inside brackets.	3
Put your finger on the first item, ...	1
Put your finger on the next item ...	2
I call the number before the brackets x ...	1
Inside the brackets is ...	10
Do ... x times ...	0
While ... do ...	0

Table 2: Usage of given phrases for articulating a natural language algorithm.

In the case of algorithm 2 students tended to describe the *meaning* of the code sequence instead of describing how to process it. Again – as in the rating of the illustrations – they preferred structure over activity. Another point is the avoidance of names (variables) for elements of the code sequence. To write a proper iteration you need a name for the current element. But this seems to be a very difficult concept.

Property	Number A1, n=25	Number A2, n=26	Total, n= 51
Some kind of explicit naming (x)	6	2	8
The structure of the code sequence is explained	1	18	19
Repetition is implied (for example “and so on”)	14	4	18
Algorithm is understandable	11	11	22

Table 3: Some properties of natural language decoding algorithms. The second column refers to algorithm 1 and the third to algorithm 2.

CONCLUSION

Illustrations can serve as door openers and as an aid to comprehending algorithms. In this study most students rated at least one drawing with the best mark, indicating that it was of help, when they had to figure out the decoding algorithm. Illustrations may facilitate the understanding directly. Additionally they may serve as a reason for discussion in a peer group.

It seems that different people need different kinds of help. A drawing that is incomprehensible to Tom might immediately open Jenny’s eyes and make her understand how the algorithm works. This diversity in human perception and thinking is an opportunity for cooperative learning. Students discover that they can profit from the differences of others.

REFERENCES

Bell, T. Fellows M., Witten, I.H. (2005): Computer Science Unplugged. Online available (access 2/12/2011): http://csunplugged.com

Gallenbacher, J.(2006): Abenteuer Informatik [Adventure Computer Science]. IT zum Anfassen. Spektrum Akademischer Verlag.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1995): Design Patterns−Elements of Resuable Object-Oriented Software. Addison Wesley, Reading.

Grimson, E. & Guttag, J. (2008): Introduction to Computer Science and Programming. MITOPENCOURSEWARE, Massachusetts Institute of Technology. Available on YouTube, http://www.youtube.com/watch?v=k6U-i4gXkLM.

de Jong, T. & van Joolingen, W. R. (2006) Scaffolds for Scientific Discovery Learning. In Handling Complexity in Learning Environments, Theorie and Research, J. Elen und R. E. Clark (eds.), Emerald Group Publishing, 107−128.

de Jong, T (1998) Scientific Discovery Learning with Computer Simulations of Conceptual Domains. In REVIEW OF EDUCATIONAL RESEARCH vol. 68 no. 2 179−201.

Kafai, Y.; Peppler, K. A.; Chapman, R. N. (2009): The Computer Clubhouse. Constructionism and Creativity in Youth Communities. Teachers College, Columbia University, New York and London.

Klahr, D. & Nigam, M.: The equivalence of learning paths in early science instruction: effects of direct instruction and discovery learning. In: Psychological Science October 1, 2004 vol. 15 no. 10, 661−667.

Meyer, R. E. (2004) Should There Be a Three-Strikes Rule Against Pure Discovery Learning? The Case for Guided Methods of Instruction. American Psychologist, 59, 1, 14—19.

Monroy-Hernández, A. & Resnick, M.(2008): Empowering kids to create and share programmable media. In Interactions, Vol. 15, No. 2, 50-53.

Moreno, A. & Myller, N. (2003): Producing an Educationally Effective and Usable Tool for Learning, The Case of the Jeliot Family. In Proceedings of the International Conference on Networked e-learning for European Universities, Granada, Spain.

Papert, S. (2000) What's the big idea? Toward a pedagogy of idea power. IBM SYSTEMS JOURNAL, Vol. 39, 720−729.

Weigend, M. (2007) Intuitive Modelle der Informatik [Intuitive Models of Computer Science], Universitätsverlag Potsdam.

Wood, D.; Bruner, J. S.; Ross, G. (1976): The Role of Tutoring in Problem Solving. In Journal of Child Psychology and Psychiatry, Vol. 17, 89−100.

Biography

Michael Weigend studied Computer Science, Chemistry and Education at the Universities of Bochum and Hagen and received a PhD in Computer Science from the University of Potsdam. He is a teacher at a secondary school in Witten, Germany and teaches Didactics of CS at the FernUniversität Hagen, Germany. He has published several books on computer programming, web development and visual modeling.