Reifying Abstract Programming Concepts through Visualization

Peter Bellström, Claes Thorén
Department of Information Systems
Karlstad University
651 88 Karlstad, Sweden
{Peter.Bellstrom, Claes.Thoren}@kau.se

ABSTRACT: *Beginner's computer programming is a course that constitutes an important part of any Information Technology (IT) related university program. For the majority of students, this particular course either comes across as a near-impossible trial by fire or as the moderately difficult challenge it should be. We relate this dichotomy to mathematical literacy and propose that information systems students and computer science students require two diametrically opposed strategies of teaching. In this context, we present three aspects in to the field of reification through visualization: a study on the benefits of visualizing using the bubble sort algorithm, a design for introducing visualization into beginners programming courses and lastly a taxonomy for programming tools which will aid in gauging the appropriateness of each tool mapped against a particular student type. Results of the study show that visualization has a positive effect on information systems students.*

Keywords: Visualization, Programming, Reification

Received: 15 March 2010, Revised 18 April 2010, Accepted 29 April 2010

1. Introduction

Beginner's computer programming is a course that constitutes an important part of any IT-related university program. For the majority of students, this particular course either comes across as a near-impossible trial by fire or as the moderately difficult challenge it should be. While some students manage the course with relative ease, a sizeable portion struggle and fail (Bennedsen & Caspersen, 2007; Lewis & Olson, 1987; Robins et al., 2003). A major reason for this struggle is that different IT students come from different academic backgrounds, particularly with regards to mathematical literacy and abstract thinking. Computer science students, for instance, tend to have scientific math backgrounds and are better prepared for problem solving and abstract thinking. Information Systems students on the other hand, are closer to the social science side of the spectrum with only basic mathematical skills, and are therefore often less proficient in problem solving and abstract thinking (see for instance Hvorecký (1990)). These two groups require two diametrically opposed strategies of teaching, particularly when teaching programming. Oftentimes, however, teaching is conducted the same way with both groups, favouring those of a mathematical background.

The traditional way of teaching programming starts with simple syntax descriptions and gradually moves on to larger examples, ending with contextualized, real-world applications with interface functionality. When adhering to this kind of teaching strategy, the problems are mathematical in nature to begin with. The students practice adding and subtracting and are shown how to assign values to variables and pass those variables to functions in order to receive a return value. These are the common basic beginner's skills that are required before you may move on to "proper" applications with purposes that are relatable to the real world.

In other words, the students practice syntax before even realizing what the commands can be used for in a broader context, not uncommonly relying on superficial formulaic memorizing rather than logical understanding. Approximately midway

through the course, the students might engage in their first real-life application example where they are allowed to see how the small commands are assembled to form greater wholes that in turn fulfil a real purpose, allowing the logic (the reasoning behind programming) to reveal itself. While this "abstract-algorithms-first-then-real-examples" strategy might work well for computer science students who are used to abstract mathematical thinking, it does not work as well for information systems students who need to understand why before how. In a worst case scenario, the students never reach the point of logic realization, but simply give up, taking the direct route between syntax and application, effectively bypassing logic (the reasoning), in which case the programming knowledge is superficial and memorized.

In this article, we argue that the reverse process is preferable for students who enter these courses equipped with less abstract problem solving skills. The visual strategy we promote starts with something tangible, something concrete, successively breaking it down into separate commands in order to show how the commands constitute the whole, and what part they play together. Visualization – seeing the end result before going into details – gives the students something they can relate to. Being able to relate to your work is most important when attempting to reach those students that struggle with programming. This inversion of traditional ways of teaching programming is shown in the knowledge triangle:

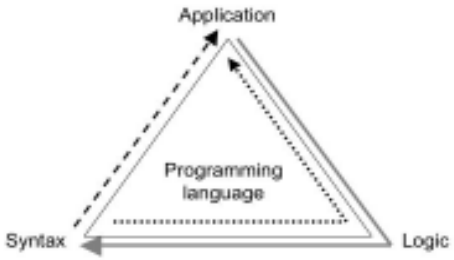

Figure 1**.** Sequence of learning programming (Bellström & Thorén, 2009, p. 93)

The triangle designates the three types of knowledge that have to be passed through sequentially when learning how to program. In the traditional sense, a student would start with syntax, moving on to logic and ending with practical application. We advocate the exact opposite, starting with application, and reaching syntax comprehension through logic. As we mentioned previously, there is also a third undesirable method, and that is to go directly from syntax to application without passing the logic stage, which commits everything to memory without prior reasoning. The three distinct, major knowledge types therefore are application, logic and syntax. In our approach, application represents the "real" example, or the practical end purpose. Logic represents an understanding of how this end purpose is to be achieved, and syntax represents the programming commands used to construct the solution.

This article is structured as follows: in section two we address other initiatives on visualization and in section three we present three aspects in the field of reification through visualization: a study on the benefits of visualizing using the bubble sort algorithm, a design for introducing visualization into beginners programming courses and lastly a taxonomy for programming tools, which will aid in gauging the appropriateness of each tool mapped against a particular student type. Finally, in section four we present a summary along with our conclusions.

2. Previous Research on Visualization

There are several approaches that focus on bridging the gap between students and the process of learning programming. In the various programming tools that we introduce, each tool represents an attempt to bring some aspect of visualization into the programming situation. For the purposes of this article, it is useful to briefly outline how they differ from each other:.

The programming tool Scratch "is a networked, media-rich programming environment" (Maloney et al., 2004, p. 104), which was designed for youths between 10-18 from economically disadvantaged areas with culturally diverse communities. Resnick et al. (2009) also point out that "[a]s Scratchers share interactive projects, they learn important mathematical and computational concepts, as well as how to think creatively, reason systematically and work collaboratively […]" (p. 60). The tool uses a building block metaphor to show how chunks of code are assembled to create a whole. When using Scratch, the users (programmers) build programs by putting graphical building blocks together. Scratch also helps the programmer because it visualizes which building blocks fit together and which ones do not.

A second approach is object-first, which is realized in the programming tools *BlueJ* (Kölling, 2008; Kölling et al., 2003) and *Greenfoot* (Henriksen & Kölling, 2004; Kölling & Henriksen, 2005). The purpose of the object-first approach is to focus conceptually on objects and object classes, disregarding certain practical details such as main functions. BlueJ is conceptually not as elaborate as Scratch, offering more or less a stripped-down Unified Modeling Language (OMG, 2009) class diagram notation language to visualize abstract classes and objects. Greenfoot takes this one step further: in Kölling & Henriksen (2005), the authors describe the Greenfoot environment as "an interactive object world [where Greenfoot] provides a framework and environment to create interactive, simulation-like applications in a two-dimensional plane" (p. 60). A third object-first environment is Alice (Cooper et al., 2003a, 2003b). In Alice, the programmer creates 3D animations. In doing so, the environment visualizes objects and their behaviours in the 3D world. The programmer can use simple "drag-and-drop" actions to make a program and therefore does not have to struggle with syntax (Cooper et al., 2003b).

An important difference between these previous approaches and our approach presented in this article is that Scratch, BlueJ, Greenfoot and Alice are all Integrated Development Environments (IDE). BlueJ and Greenfoot are furthermore geared specifically toward the Java programming language and Scratch and Alice use their own 'language'. Our approach exists independently of language and environment. In other words, the IDE approach affects the entire teaching and learning situation, while our approach complements it. This is an important distinction because what we want to achieve with our approach is simplification, rather than complication. The benefits of being a complement rather than a full environment is exemplified by Leung (2006) where the author states that "[a]lthough equipped with high-level tools, such as IDE and libraries, students often feel more overwhelmed than empowered" (p. 150).

Other ways of visualizing that have been implemented in the past include gaming and games, both in the sense of constructing games (Bayliss & Strout, 2006; Chamillard, 2006; Sung, 2009) as well as playing them (Eagle & Barnes, 2008).

Finally, several algorithmic animation systems and algorithmic animation languages have been developed. This is also a rather large research area in visualization. The interested reader can therefore read more about these types of systems and languages in Price et al. (1993) and Karavirta et al. (2010).

3. Three Aspects in the Field of Reification through Visualization

In this section, we present three contributions to the field of reification through visualization:

In the first aspect, we present an initial pilot study, where we examined how visualizing an abstract, mathematical programming algorithm enhances the understanding of the source code, and in a broader sense the programming language itself (Bellström & Thorén, 2009). Serving as our example, the bubble sort algorithm is a simple sorting algorithm that is often used in introductory programming courses. The algorithm works by gradually stepping through an array of values, comparing each value pair, and potentially swapping them so that the lower value is always to the left of the greater value. As values are sorted, they progressively rise, or "bubble" to their proper location in the array with each sweep of the surrounding 'while' loop. The array is checked several times until the array is completely sorted. The sorting algorithm manages, in its simplicity, to use several central commands and put them to use, which is why it is popular in introductory courses. Sequence, selection and iteration are all included. In the version of the bubble sort algorithm that we used (Figure 2), focus is put on simplicity and

```
int outerCounter = 0, innerCounter = 0, tempValue = 0;
int myArray[]={10,7,8,1,5,9,2,4,3,6};

for(outerCounter = 0; outerCounter < 10; outerCounter++){

    for(innerCounter = 0; innerCounter < 9; innerCounter++) {

        if(myArray[innerCounter] > myArray[innerCounter + 1]){

            tempValue = myArray[innerCounter + 1];
            myArray[innerCounter + 1] = myArray[innerCounter];
            myArray[innerCounter] = tempValue;
        }
    }
}
```

Figure 2. The Bubble Sort Algorithm

understandability rather than speed and optimization. The algorithm is therefore not optimized and could be solved using different iteration types. In Figure 2, we show one possible implementation of the bubble sort algorithm using the Java (Java, 2010) programming language. In the code, myArray is the vector that is being sorted using two "for" loops. It should be noted that even though the vector might be sorted before all conditions are met, all iterations are still made.

Making the abstract concrete, or reifying abstract programming code into something "real", is a powerful way of introducing information systems students to the subject, and this concept explains quite accurately what we wanted to achieve with our pilot study. In the pilot study, the students were shown a visual, stick figure animation of the bubble sort algorithm. Before and after having been shown the visualization, they examined the source code and were asked two questions on whether the animation added to their understanding or not. The results showed that employing a visualization that uses some kind of metaphor that attaches itself to real-world concepts has the potential to compensate for a lack of prior mathematical experience.

In the second aspect, we present a design for introducing visualization into beginners programming courses. The aim with the proposed design is to urge students to study towards a deep, holistic learning which is particularly important when learning programming; however not all students have that approach (Booth, 1992; Kilbrink, 2008; Segolsson, 2006). As addressed above in the section "previous research", there are several approaches to visualization. There are many kinds of visualization techniques, and the particular type of visualization employed in this contribution is animation.

In the third aspect we present a classification of visual programming environments. Attempting to visualize or attach a metaphor to facilitate learning is not new. Programming tools such as Alice, Scratch, GreenFoot and BlueJ all use various degrees of visual aids and Graphical User Interface (GUI) functions to make programming more intuitive. These applications all have their individual strengths and weaknesses, as we will show using our own taxonomy. If using visualization and practical application is a good beginning, the tools achieve the end with varied degrees of success. To show this, we position the tools along two dimensions. The "granularity" dimension represents the size of the programming components needed, from the smallest (individual command syntax) to the largest (sections of code that are visual representations). Thus, tools that require detailed programming syntax knowledge gravitate towards the high rating, and tools that have larger building blocks rate more towards low granularity. The second dimension represents the degree of visualization. In this case, visualization refers to a metaphoric visual representation that attaches to some element in a real (or imagined) world. Whether it is parts of a jigsaw puzzle or Lego building blocks, or controlling the movements of a figure skater, it does not matter; the important thing is that it is relatable. A tool that rates high on the visualization axis has a sophisticated, almost narrative style using a real-world example. A very logical tool from an information systems point of view would therefore rate high on visualization and low on granularity. No details in command syntax are required, only larger building blocks and a relatable metaphor. In so doing, we have achieved the kind of reification that we refer to and advocate in this article.

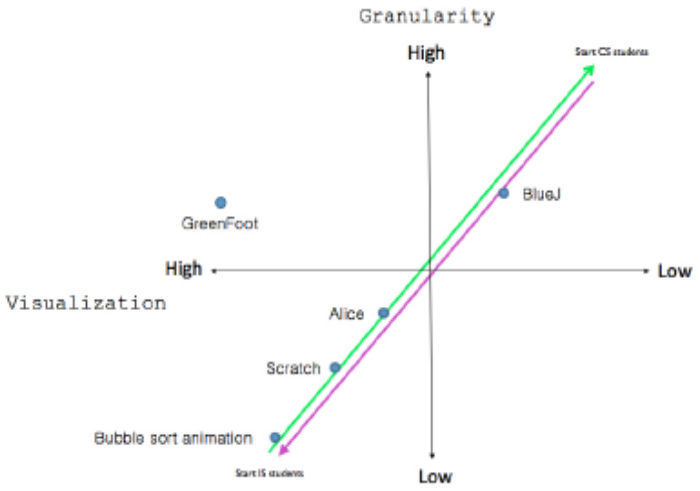

Figure 3. A taxonomy of visual programming environments (adaped and modified from Bellström & Thorén (2010, p. 135))

3.1 Visualizing the Bubble Sort Algorithm

In the pilot study, which was initially presented in Bellström and Thorén (2009), the authors examined how visualizing an abstract, mathematical programming algorithm enhances the understanding of the source code, and in a broader sense the

programming language itself. Making the abstract concrete, or reifying abstract programming code is a powerful way of introducing information systems students to the subject. Results showed an increased understanding of abstract programmingconcepts. Five students taking an intermediate programming course were shown a visual, stick figure animation of the bubble sort algorithm. Figs. 4 through 9 show a slightly improved version of the visualized Bubble Sort algorithm focusing on how “35” and “4” switch place.

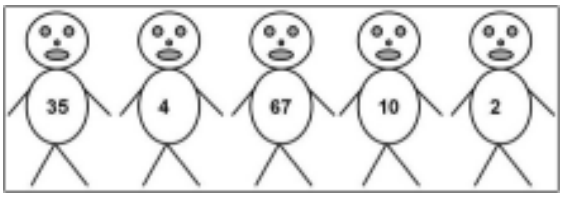

Figure 4. Start up position of stick figures (values) to be sorted

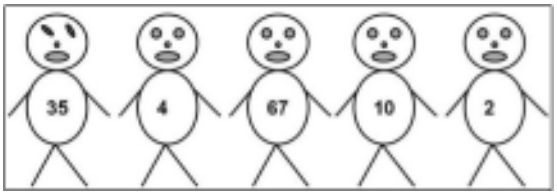

Figure 5. The stick figure to the left is checking the stick figure to the right (its value)

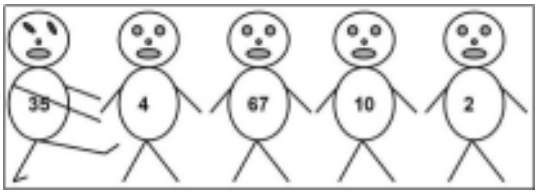

Figure 6. The stick figure to the left (has a higher value) kicks the stick figure to the right

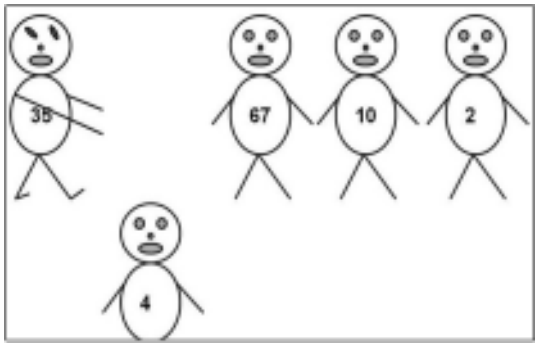

Figure 7. The stick figure to the right is kicked into repositioning

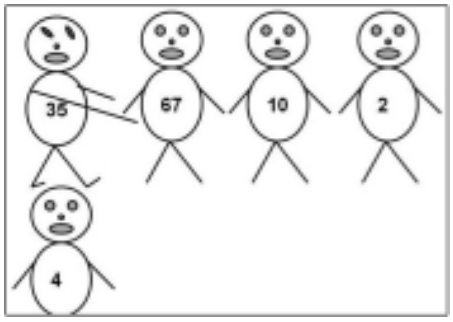

Figure 8. The stick figure to the left takes one step forward

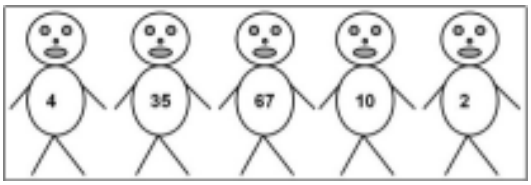

Figure 9. The stick figure that was kicked takes one step back

Figures 4 to 9 should be interpreted as follows: Figure 4 holds the sequence of stick figures (values) to be sorted.In Figure 5, the stick figure to the left (value 35) checks whether the stick figure to the right (value 4) has a lower value on the chest and if so kicks the stick figure to the right (Figure 6)". In Figure 7, the stick figure (value 4) has been kicked down to another position and in Figure 8 the stick figure to the left (value 35) takes one step forward. Finally, in Figure 9 the stick figure that was kicked moves up to the correct position of the other stick figure (value 35). Now "35" and "4" have switched places with each other.

Prior to showing the visualization, the source code had been shown and discussed briefly. After the students had observed the animation, they were asked two open-ended questions:

1. Did the stick figures with numbers add to your understanding of programming? What? How? Motivate!

2. Could you yourself explain bubble sorting to someone with help from these examples (source code and/or visualizations with stick figures)? Which one of the two, or both? Motivate!

The results showed an increased understanding of abstract programming concepts, and furthermore showed that there is potential in reversing the sequence of learning starting with a tangible, visualized application and ending with syntax. Thus, employing a visualization that uses some kind of metaphor that attaches itself to real-world concepts might compensate for a lack of prior mathematical experience.

3.2 A Design for Introducing Visualization into Beginners Programming Course

Reifying abstract programming concepts through visualization should be addressed as one promising path to follow to make abstract programming concepts more concrete for students enrolled in beginners programming courses. According to Sung (2009), the Computer Science education community has dealt with this issue for some time now and has had a sound understanding of how to integrate visualization into Computer Science courses. In this section, we propose a design and approach to achieve a rewarding integration. The described and proposed approach given below is an extension of the design proposed in Bellström & Thorén (2010). While describing our approach, we again address the bubble sort algorithm and use it as a concrete example on how to apply and use the design. The design follows a classical three-phase approach starting with a preparation phase followed by a realization phase and ending with a follow-up phase. This split into three phases, and the phases themselves, is inspired by the work presented in Eriksson et al. (1988) and Bellström & Carlsson (2004). As will be addressed in the next paragraph, each step might actually include several tasks (sub-steps) but still, the design can be generalized into the three addressed phases.

The preparation phase includes three tasks. It starts with a short introduction to the application or problem that is addressed in the visualization (cf. Application in Figure 1). The visualization is then shown to the students (cf. Logic in Figure 1). After having been shown the visualization, the students should reflect upon what the visualization actually showed. Making some notes could aid in the process of understanding how to actually solve the application or problem and to get a deeper and more holistic understanding of the application or problem and its solution. A deep/holistic learning approach to learning has been mentioned as particularly important when learning programming; however not all students have that approach (Booth, 1992; Kilbrink, 2008; Segolsson, 2006).

The realization phase includes two tasks: technical reflection and implementation. In the technical reflection task, students should reflect upon what instructions they need to solve the problem and how the instructions should be organized. Using a modelling language, such as the Unified Modeling Language (Object Management Group, 2007), could facilitate this task. However, it should be noted that learning suitable Unified Modeling Language diagrams could be a threshold on its own. On the other hand, some type of pseudo code helping in structuring the solution could instead be one way to follow. In the implementation task, students implement their solution to the problem or application (cf. Syntax in Figure 1).

The follow-up phase includes only one task: to compare and reflect upon the stated problem (cf. Application in Figure 1) and visualization (cf. Logic in Figure 1) with the actual outcome of the implemented solution. In doing this, the students come full circle. In other words, the students should once again be allowed to reflect upon what they have actually done.

Let us again address the bubble sort algorithm as an illustrating example. Applying the proposed approach, students are first told that the problem (or the practical application) is sorting a sequence of numbers in ascending order. Students are then shown the animation (for instance the example containing the stick figures). Afterwards, the students are told to reflect upon what they saw in the animation (preparation phase). The reflection is followed by a technical reflection where the students figure out what instructions, such as iterations, selections etcetera need to be used and in what order (see Figures 4-9). The students are then told to solve the problem and implement a solution in the chosen programming language (realization phase). Finally, students are asked to reflect and compare what they implemented with both reflections from phase one and phase two (follow-up phase).

3.3 A Taxonomy of Programming Tools

The two types of students we have discussed earlier, literally require diametrically opposed approaches when learning programming, even if the desired end result is the same. Since programming environments are designed to smooth the learning threshold, it is useful to examine what kind of approach (of the two mentioned earlier) each programming environment caters to so that we may understand what kind of student each programming environment is optimized for.

We examined some of the common software environments and placed them in a diagram that highlights two characteristics of each environment that represent a trajectory from one kind of starting knowledge to the other. If we assume that Information Systems students are visual, and Computer Science students are mathematical to begin with, they both need each other's knowledge to completely master programming. In other words, IS students should move towards the starting point of CS students, and CS students should move towards the starting point of IS students.

Thus each positioned programming environment in the diagram represents a starting point for the distance of knowledge travelling necessary to achieve practical application, which is represented by low granularity (referring to a low level of detailed programming knowledge required) and high visualization in the upper right corner. The knowledge travelling takes place across the diagram showing the transition from one type of knowledge to another. Students of information systems should start their programming journey in the bottom left corner, where visualization is high, and granularity is low (ie a low level of detailed knowledge of syntax is required), and progress diagonally across the graph towards the top right corner, reaching the minute details of programming syntax last. Computer science students do the exact opposite – they start (as tradition would have it) in the top right corner with the mathematic operations of programming commands, and move diagonally across towards the bottom left corner, ending with a visual representation of their achievements.

In other words, the type of knowledge required to complete each journey does not differ between the different student types, but they are entered into according to a different succession. Information systems students begin with a solid visual representation of the task at hand, be it a design concept or a working prototype. The prototype is then broken down into components which each represent specific, isolated functions in the overall program. At this point, it should be easy to understand how each component has to relate to the other components in order for the program to perform according to specifications. Each component is then broken down further into small operations, and finally each operation into programming syntax. The focus is functionality and practical application where each mathematical, programmable operation is contextualized in the world before it is taught and created. In other words, the mathematics of the programming are not taught as mathematics, they are instead made sense of in a broader context.

The purpose of the taxonomy, therefore, is to gauge the appropriateness of each tool relative to each student type. The closer a tool resides to each student grouping's corner starting point, the better suited it is, because the whole distance needs to be travelled from beginning to end, in order to complete the programming journey.

4. Summary and Conclusion

We have presented in this article a framework for teaching how to program that contains two strategies: one strategy that is suitable for those students that are of a mathematical inclination, and another that is suitable for those who are not. It should be made clear that this is not a binary choice, but rather a gradient scale where one extreme is syntax-only and the other

extreme is simply pictures of the world. In other words, the purpose of the taxonomy of visualization is to gauge any given teaching strategy simply by positioning it according to the two dimensions. Depending on where on the granularity/ visualization scale a particular strategy belongs, the result will offer an indication on what group of students – mathematically inclined or not - the teaching situation is optimized for. Should the situation rate high on visualization and low on granularity, the strategy starts with application moving towards syntax, and if it rates low on visualization and high on granularity, it is a mathematical, syntax-based approach suitable for mathematical problem solving.

By positioning the four programming tools (Alice, BlueJ, GreenFoot and Scratch) along the dimensions, we also provide a classification of those tools and in so doing show which student group they are appropriate for, and to what degree they are appropriate.

The granularity/visualization axis, therefore, are a tool that is helpful when deciding which group any particular teaching strategy belongs to. This kind of strategy classification is useful because it helps to predict how well the students receive the material, and in turn perhaps fewer students will fail introductory programming courses. In particular, distance education students should benefit from this classification because they do not have the personal interaction between teacher and students that a normal teaching situation offers.

In this article, we have also presented a study on visualizing the bubble sort algorithm and a design for introducing visualization into beginners programming courses. Results of the study on visualizing the bubble sort algorithm indicate that visualization is indeed a powerful tool for aiding information systems students. In the presented design for introducing visualization into beginners programming courses, we once again focused on comprehension and how to encourage and drive students to study towards a deep/holistic learning which is very important in connection to learning how to program.

References

[1] Alice (2010). What is Alice?. [Electronic]. Available: http://www.alice.org/index.php?page=what_is_alice/what_is_alice [20100611].

[2] Bayliss, J.D., Strout, S. (2006). Games as a 'Flavor' of CS1. *In: SIGCSE'06*, pages 500-504.

[3] Bellström, P., Carlsson, S. (2004). Towards an Understanding of the Meaning and the Contents of a Database through Design and Reconstruction. *In:* Proceedings of the 13th International Conference on Information Systems Development Advances in Theory, Practice and Education, p. 283-223.

[4] Bellström, P., Thorén, C. (2009). Learning How to Program through Visualization: A Pilot Study on the Bubble Sort Algorithm. *In:* Proceedings of the 2nd International Conference on the Applications of Digital Information and Web Technologies, IEEE, p. 90-94.

[5] Bellström, P., Thorén, C. (2010). On the Importance of Visualizing in Programming Education. *In:* Proceedings of the 12th International Conference on Enterprise Information Systems V. 5 Human Computer Interaction, p. 131-136.

[6] BlueJ (2010). *What is BlueJ?*. [Electronic]. Available: http://www.bluej.org/about/what.html [20100611]

[7] Booth, S. (1992). Learning to Program: A Phenomenographic Perspective. PhD Thesis, Göterborgs universitet, Acta.

[8] Chamillard, A.T. (2006). Introductory Game Creation: No Programming Required. *In SIGCSE'06*, p. 515-519.

[9] Cooper, S., Dann, W., Pausch, R. (2003a). Teaching Objects-first In Introductory Computer Science. *In SIGCSE´03*, p. 191-195.

[10] Cooper, S., Dann, W., Pausch, R. (2003b). Using Animated 3D Graphics to Prepare Novices for CS1. *Computer Science Education* 13 (1) 3-30.

[11] Cooper, S., Dann, W., Pausch, R. (2009). *Learning to Program with Alice*. Pearson Education.
Eagle, M., & Barnes, T. (2008). Wu's Castle: Teaching Arrays and Loops in a Game. *In: ITiCSE'08*, p. 245-249.

[12] Eriksson I.V., Hellman, R. & Nurminen, M.L. (1988). A Method for Supporting Users´ Comprehensive Learning. *Education & Computing The International Journal.* 4 (4) 252-264.

[13] Greenfoot (2010). What is Greenfoot?. [Electronic] Available: http://www.greenfoot.org/about/whatis.html [20100611]

[14] Henriksen, P. & Kölling, M. (2004). Greenfoot: Combining Object Visualization with Interaction. *In:* OOPSLA'04, p. 73-82.

[15] Hvorecký, J. (1990). On the Connection Between Programming and Mathematics. *SIGCSE BULLETIN* 22 (4) 53-54.

[16] Java (2010). *java.com: Java + You*. [Electronic] Available: http://www.java.com/en/ [20101126].

[17] Karavirtaa, V., Korhonena, A., Malmia, L., Naps, T. (2010). A Comprehensive Taxonomy of Algorithm Animation Languages. *Journal of Visual Languages & Computing* 21 (1) 1-22.

[18] Kilbrink N. (2008). *Legorobotar i skolan Elevers uppfattningar av lärandeobjekt och problemlösningsstrategier*. Licentiate Thesis, Karlstad University Studies.

[19] Kölling, M., Henriksen, P. (2005). Game Programming in Introductory Courses with Direct State Manipulation. *In ITiCSE'05*, p. 59-63.

[20] Kölling, M. (2008). Using BlueJ to Introduce Programming. *In:* Reflections on the Teaching of Programming Methods and Implementations, Springer, Berlin, p. 98-115.

[21] Kölling, M. (2009). Introduction to Programming with Greenfoot Object-Oriented Programming in Java with Games and Simulations. Pearson Education.

[22] Kölling, M., Quig, B., Patterson, A., Rosenberg, J. (2003). The BlueJ System and its Pedagogy, *Journal of Computer Science Education.* 13 (4) 249-268.

[23] Leung, S.T. (2006). Integrating Visualization to Make Programming Concepts Concrete – Dot Net Style, *In:* SIGITE'06, p. 149-156.

[24] Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., Resnick, M. (2004). Scratch: A Sneak Preview. *In:* Second International Conference on Creating, Connecting and Collaborating through Computing, p. 104-109.

[25] Object Management Group (2007). *OMG Unified Modeling Language (OMG UML), Superstructure*. [Electronic]. Available: http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF [20100612].

[26] Price, B.A., Baecker R.M., Small, I.S. (1993). A Principled Taxonomy of Software Visualization, *Journal of Visual Languages and Computing* 4 (3) 211–266.

[27] Resnick, M., Maloney, J., Monroy-Hernandes, A., Rusk, N., Eastmond, E., Brennan, K., Miller, A., Rosenbaum, E., Silver, J., Silverman, B., Kafai, Y. (2009). Scratch: Programming for All. *Communications of the ACM*, 52 (11) 60-67.

[28] Scratch (2009). *Reference Guide version 1.4*. [Electronic]. Available: http://info.scratch.mit.edu/Support/Reference_Guide_1.4 [20100611].

[29] Segolsson, M. (2006). *Programmeringens intentionala object Nio elevers uppfattningar av programmering*. Licentiate Thesis, Karlstad University Studies.

[30] Sung, K. (2009). Computer Games and Traditional CS Courses. *Communications of the ACM* 52 (12) 74-78.

Authors biographies

Peter Bellström received his Ph.D degree in Information Systems from Karlstad University, Karlstad, Sweden, in Spring 2010. He is now working as an assistant professor at the department of Information Systems at Karlstad University and as a Director of Studies for the study programme in Web and Multimedia. His research interests include conceptual modeling, schema integration and teaching and learning techniques for programming courses. This includes integration of static schemata, integration of dynamic schemata, Problem-Based Learning (PBL), visualizing abstract programming concepts and combining visual programming environments, developed for teaching and learning programming, with game tasks.

Claes Thorén is a doctoral candidate at the department of Information Systems at Karlstad University in Sweden, and is attached to The Swedish Research School of Management and IT (MIT). Claes holds bachelor degrees in Information Systems (1998) and English Literature (2005) as well as a Master of Arts with distinction in Cultural Studies from University of East London (2007).His main research interests concern how newspaper organizations cope with technological innovation and social media. Other interests include visual culture, digital games theory, cultural theory and management studies.