Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 178 (2007) 15–22
www.elsevier.com/locate/entcs

Animation Metaphors for Object-Oriented Concepts

Jorma Sajaniemi[1,2] Pauli Byckling[3] Petri Gerdt[4]

Department of Computer Science and Statistics
University of Joensuu
Joensuu, Finland

Abstract

Program visualization and animation has traditionally been done at the level of the programming language and its implementation in a computer. However, novices do not know these concepts and visualizations that build upon programming language implementation may easily fail in helping novices to learn programming concepts. Metaphor, on the contrary, involves the presentation of a new idea in terms of a more familiar one and can facilitate active learning. This paper applies a metaphor approach to object-oriented programming by presenting new metaphors for such concepts as class, object, object instantiation, method invocation, parameter passing, object reference, and garbage collection. The use of these metaphors in introductory programming education is also discussed.

Keywords: Metaphor, object-oriented programming, program visualization, program animation.

1 Introduction

Program visualization and animation has traditionally been done at the level of the programming language and its implementation in a computer. For example, variables have been visualized as boxes (representing memory locations), and nested function calls as a stack of frames containing parameters and local variables (representing the call stack implementation in many computer architectures). In object-oriented (OO) context, animation has also been based on UML diagrams that reveal connections between objects and classes and thus represent another level, i.e., relationships between components of an individual program. We know of only one program animation system, PlanAni [20], that builds its visualization on general

[1] This work was supported by the Academy of Finland under grant number 206574
[2] Email: saja@cs.joensuu.fi
[3] Email: pbyckli@cs.joensuu.fi
[4] Email: pgerdt@cs.joensuu.fi

doi:10.1016/j.entcs.2007.01.037

programming knowledge (roles of variables) and uses metaphors to make this knowledge easier to assimilate by learners.

Novices have problems in learning the very basic OO concepts which results in misconceptions leading to either erroneous or suboptimal programming skill (see, e.g., [4,6,7]). Program visualization and animation are supposed to enhance learning and prevent misconceptions but the visualizations should be informative and at the same level as the concepts to be learned. Thus visualizations that build upon programming language implementation or uninformative graphical notation may easily fail in helping novices to learn programming concepts.

Metaphor involves the presentation of a new idea—the so called target—in terms of a more familiar and usually more concrete one, the source [3,11,17]. In contrast to analogy, metaphor is not an exact counterpart but differs from the target usually both in form and in content. Critical to the power of metaphor is that the relation between the source and the target must involve some transformation, hence people have to actively construct the relationships that comprise the metaphor [1]. Salient dissimilarities of the source and the target—in the context of salient similarities—stimulate thought and can facilitate active learning [3].

This paper applies a metaphor approach to object-oriented programming. Our ultimate goal is to provide novices with metaphors that will help them in learning basic OO concepts. For this purpose, we present new metaphors for class, object, object instantiation, method invocation, parameter passing, object reference, and garbage collection. The metaphors are designed to grasp the basic ideas of object-oriented programming; they do not rely on implementation issues or diagramming techniques designed for expert use.

The rest of this paper is organized as follows. Section 2 describes the new metaphors and explains how they can be visualized and animated in a program animator. Section 3 is a literature review that presents visualizations in current program animation systems and compares them with our ideas. Section 4 discusses possible uses of the metaphors and metaphor-based animation in introductory programming education. Finally Section 5 contains the conclusion.

2 Visualization of OO Concepts

An object encapsulates the existence, state and behavior of an entity. Its visualization should reflect these three aspects. The existence is limited by the instantiation of an object and its destruction in garbage collection. The state is manifested in the member variables, and the behavior is a result of method invocations that include the creation and destruction of local variables. The behavior of individual member and local variables can be described by roles [18,19] that already have metaphors, e.g., a dog for the role follower, a box for gatherer etc [20]. For an object, we therefore select the metaphor of a *watch panel* with class-dependent fixed "monitors" for its member variables depicted in the form of *role metaphors* (see Figure 1).

The instantiation of an object is animated by making a copy of a class-specific *blueprint* found in a *blueprint book*. The blueprint book lays normally outside the

screen and emerges only when needed for object instantiations. Each blueprint occupies its own page in the book, and class variables are located on the same page. Whereas the background color for blueprints is blue, the background color of the class variable area is white. This area becomes visible whenever an object of that class is active. Class variables are depicted with the same role metaphors as member variables.

In this context a meaningful metaphor for method invocation is a temporary *workshop* containing all parameters and local variables, and a *workbench* for the result of the invocation. Because new local variables can be created and destroyed during the invocation, the workshop must be large enough to accommodate all variables. If another method is invoked or a method is invoked recursively, a new workshop is created. Thus the number of co-existing workshops depends on the number of unfinished method invocations. To stress the fact that method invocations are associated with the object's member variables, the workshops are attached to the watch panel depicting the object. Finally, a static method is visualized as a *permanent workshop* with a concrete foundation and a strong roof.

The traditional verbal metaphor for method call is "message passing". We visualize this metaphor with an *envelope* containing actual parameters. The animation of a method call starts with the creation of the parameter envelope in the invoking workshop, the envelope then flies to the watch panel associated with the called object, a new workshop emerges, and the values in the envelope are transferred to role metaphors of the formal parameters thus giving their initial values. The empty envelope stays on the workbench and is filled with the return value when the method invocation ends. Then the finished workshop disappears and the envelope flies back to the calling workshop along a path that was created during the method call.

In Java, pointers are replaced by object references. This concept has been found to be problematic for novices and a well-designed metaphor is needed. Pointers have been traditionally depicted by arrows that are redrawn to point to a new item each time a new value is assigned to the pointer. This arrow metaphor builds on the implementation aspect: pointers are memory addresses and an assignment to a pointer means the setting of a new memory address to the pointer.

In order to avoid this implementation point of view, we suggest a *pennant* metaphor: an object reference is visualized as two pennants with the same unique identity; one pennant is attached to the object reference variable and the other to the referenced object. A null reference is visualized with two pennants lying on the ground. Assignment of a newly created object to an object reference is animated by moving the other pennant to the new object; re-assignment of an object reference is animated by moving the pennant from the old target to the new target. If two variables refer to the same object, the object has two pennants. As a consequence, an object with no pennants cannot be referenced and is subject to garbage collection, which is animated by a *garbage vehicle* that moves around and stops next to each object, i.e., watch panel, with no pennants. The finalizer is then invoked and the watch panel is finally squashed into the vehicle.

Figure 1 is a sketch of a visualization based on the above metaphors. The ani-

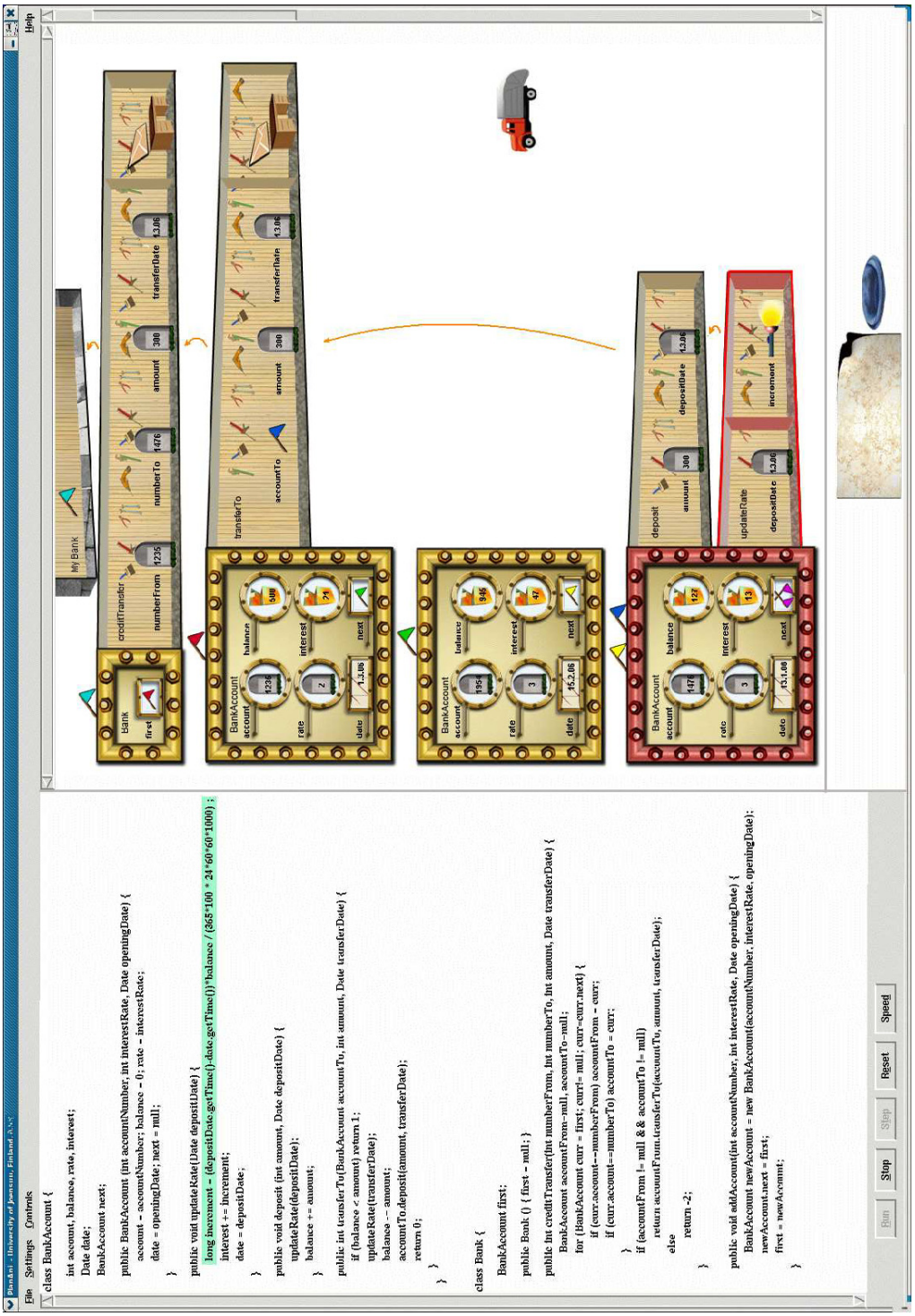

Fig. 1. A hypothetical user interface for a program animator using the OO metaphors.

mated program models a bank that consists of bank accounts. The object reference `myBank` is visualized as a pennant whose pair is attached to the `Bank` object. Both of these pennants have the same color that is different from all other pennant pairs. The individual bank accounts are implemented as a linked list and visualized as watch panels with pennants making the linkage. The `next` link in the last account is null represented with two pennants on the ground.

Member variables are depicted with role images: account number stored in the variable `account` is a fixed value; the variable `balance` that gathers the net effect of deposits and withdrawals is a gatherer etc. System defined objects that conceptually encapsulate a single attribute (e.g., String, Date) are visualized just like primitive variables. For example, the latest transaction date stored in the member variable `date` is a most-recent holder.

The currently active object (account 1476 at the bottom of the window) and its active method (`updateRate`) are enhanced with red color. The workshop for this method invocation contains the parameter `transferDate` (having the role fixed value) and the local variable `increment` (temporary). This method has been called from the method `deposit` of the same object, which was called from the method `transferTo` of the bank account 1235 (the topmost bank account), which in turn is called from the method `creditTransfer` of the single bank object—called from the static method `main`. The methods `creditTransfer` and `transferTo` return an integer; therefore they have a workbench for the preparation of the return value.

The garbage vehicle moves around but currently there are no objects with no pennants. The blueprint book is not visible at the moment.

3 Comparison with Current Visualizations

Current OO program animation systems use basic geometric figures (2D or 3D boxes, cones, arrows, etc) for visualizations. These figures make up a notation language that is new to students. Thus students have to learn simultaneously the new geometric notation language, the new OO concepts themselves, and the connections between these two worlds.

For example, GROOVE [10], Jeliot 3 [15], OGRE [14], and OOP-Anim [5] use geometric figures to represent OO constructs such as class and object, relationships between classes and objects, relationships within class hierarchies etc. On the other hand, JACOT [12], JAN [13], and JavaVis [16] use UML notations for the same purposes. Even though UML is a standard notation language, it is new to novices and must be learned in addition to the OO concepts themselves. None of these systems uses metaphors for OO concepts.

JACOT, JAN and JavaVis use UML sequence and object diagrams also to animate method calls; method instances are not visualized. GROOVE visualizes method calls similarly to our suggestion but method instances are not attached to the corresponding objects. In OOP-Anim, each object is depicted as in UML and contains both the member variables and names of all methods; a method invocation is animated by changing the color of the invoked method within the object

and the whole method is executed in a single step. Neither parameters nor local variables are represented in the visualization; moreover, several instances of the same method cannot be visualized. Jeliot 3 animates method invocations with a special method instance area that represents the implementation-based method call stack. None of these system uses anything similar to our workshop metaphor.

The above systems visualize object references with arrows or miniature pictures of the referenced object. If a new value is assigned to an object reference variable, the arrow is redrawn or the miniature picture is replaced by a new one. This is in contrast to our pennant metaphor where the referencing variable has a unique identity that does not change if the target of the reference changes.

4 Metaphors and Animation in Education

Metaphors, visualization, and animation can be used in education in several ways. For example, the level of student engagement seems to affect the outcome of learning efforts [9]. It seems obvious that a quick passive watching of an autonomously running animation does not provide improvement in learning but more laborious mental processing of the visualizations is needed.

In our earlier studies [2,21] students watched animations of short programs for about half an hour for each program: first the teacher demonstrated the animation, then the students ran the animation by themselves using data that was carefully selected by the teacher, and finally the students animated the program with their own input data. All the time, students were encouraged to proceed slowly with the animation and predict the effect of the next statement on the values of variables and other aspects of the program. This method proved to be effective but required a substantial time for the animation of each program. We therefore suggest that animation is used only for a few, carefully selected OO programs that demonstrate central aspects of OO concepts.

The selection of only a few programs to be animated makes the authoring of the animations rather easy as there is no need to animate arbitrary programs. Especially, we do not suggest that the OO metaphors and slow animation should be used as the only programming environment in an introductory programming course. Rather, we do suggest the use of common programming environments for normal programming tasks, and the use of metaphor-based animation for the few, carefully selected programs only. Each animation should be presented after the appropriate concepts have been treated in lectures. In addition, the metaphors can—and probably should—be used in static visualizations used in learning materials so that students have enough time to actively construct the relationships that comprise the metaphors. Only repeated elaboration of the concepts may result in active learning.

The effects of metaphors have been found to be positive in complex learning situations whereas their effects in simple tasks (retention or rote learning) has been found to be minimal [8]. As programming is a very complex task, one can expect positive effects. Indeed, our earlier experiments [2,21] have indicated a strong effect of metaphors and animation on students' capabilities to use variables in imperative

programming. Classes, objects, methods and other OO constructs are even much more complex concepts, and the use of these OO constructs leads to much more complex programming situations than the use of variables and simple control structures. Thus metaphors can be expected to have a positive effect not only on simple programming but on the learning of OO concepts, also.

For example, a typical novice misconception in OO concepts is to confuse objects with classes [4,7]. Our new metaphors make this distinction clear: classes are represented as blueprints in a blueprint book, and objects are concrete instances created using the appropriate blueprint. The visually strong distinction with blue-and-white blueprints for classes and brass-colored watch panels for objects makes the difference between these concepts clear. Another novice misconception is to assume that methods have some associated object. In the animation, method execution is depicted with workshops that are spatially located next to the associated object, but the static method `main` has no object next to it. Thus the difference between static and non-static methods can be easily explained in visual terms.

In summary, previous literature gives grounds to assume that metaphors and animation can be utilized to enhance learning OO programming. Of course, this hypothesis must be thoroughly studied with empirical investigations.

5 Conclusion

We have presented new metaphors for classes (blueprint book), objects (watch panel), method invocation (workshop), parameter passing (envelope), return value (workbench), object reference (pennant), and garbage collection (garbage vehicle). We have also presented visualizations for these metaphors and sketched program animation that uses the metaphors.

The visualizations and animations do not scale up to large programs but this is not a problem. We do not suggest using the visualizations in, e.g., debugging or comprehending large programs. Instead, we do suggest that in elementary programming education, roles of variables are first introduced with their role metaphors using the PlanAni program animator. When the role metaphors are familiar, the OO metaphors can be introduced by animating a few, carefully selected OO programs. With appropriate student engagement this can be expected to give students a correct mental model of the relationships between OO concepts—a mental model that builds up on a spatial representation of program execution. The visually rich visualizations of the metaphors are expected to consolidate this mental model so that even though the visualizations do not scale up on computer screen, the metaphors do scale up in novices' mental representations.

We are currently authoring animations for several Java programs. In future, we are planning to use them in introductory programming courses and to study how the metaphors and animations affect the development of novices' mental models of OO concepts.

References

[1] Alty, J. L., R. P. Knott, B. Anderson and M. Smyth, *A framework for engineering metaphor at the user interface*, Interacting with Computers **13** (2000), pp. 301–322.

[2] Byckling, P. and J. Sajaniemi, *Roles of variables and programming skills improvement*, in: *Proc. of the 37th SIGCSE Technical Symposium on Computer Science Education*, 2006, pp. 413–417.

[3] Carroll, J. M. and R. L. Mack, *Metaphor, computing systems, and active learning*, International Journal of Human-Computer Studies **51** (1999), pp. 385–403.

[4] Eckerdal, A. and M. Thuné, *Novice Java programmers' conceptions of "object" and "class", and variation theory*, in: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education ITiCSE'05* (2005), pp. 89–93.

[5] Esteves, M. and A. Mendes, *OOP-Anim, a system to support learning of basic object oriented programming concepts*, in: *CompSysTech' 2003 - International Conference on Computer Systems and Technologies*, 2003, available at http://ecet.ecs.ru.acad.bg/cst/Docs/proceedings/S4/IV-6.pdf.

[6] Fleury, A. E., *Programming in Java: Student-constructed rules*, in: *Proc. of the 31th SIGCSE Technical Symposium on CS Education*, 2000, pp. 197–201.

[7] Holland, S., R. Griffiths and M. Woodman, *Avoiding object misconceptions*, SIGCSE Bulletin **29** (1997), pp. 131–134.

[8] Hsu, Y.-C., *The effects of metaphors on novice and expert learners' performance and mental-model development*, Interacting with Computers **18** (2006), pp. 770–792.

[9] Hundhausen, C. D., S. A. Douglas and J. T. Stasko, *A meta-study of algorithm visualization effectiveness*, Journal of Visual Languages and Computing **13** (2002), pp. 259–290.

[10] Jerding, D. F. and J. T. Stasko, *Using visualization to foster object-oriented program understanding*, Technical Report GIT-GVU-94-33, Atlanta, GA, USA (1994).

[11] Lakoff, G. and M. Johnson, "Metaphors We Live by," University of Chicago Press, Chicago, 1980.

[12] Leroux, H., C. Mingins and A. Requile-Romanczuk, *JACOT: A UML-based tool for the runtime-inspection of concurrent Java programs*, in: R. Filman, M. Haupt and K. Mehner, editors, *1st Workshop on Advancing the State-of-the-Art in Run-Time Inspection*, 2003, available at http://www.st.informatik.tu-darmstadt.de/pages/workshops/ASARTI03/LerouxASARTI03.pdf.

[13] Lohr, K.-P. and A. Vratislavsky, *Jan - Java animation for program understanding*, in: *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments*, 2003, pp. 28–31.

[14] Milne, I. and G. Rowe, *Ogre: Three-dimensional program visualization for novice programmers*, Education and Information Technologies **9** (2004), pp. 219–237.

[15] Moreno, A., N. Myller, E. Sutinen and M. Ben-Ari, *Visualizing programs with Jeliot 3*, in: *Proceedings of the International Working Conference on Advanced Visual Interfaces*, 2004, pp. 373–376.

[16] Oechsle, R. and T. Schmitt, *JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java debug interface (JDI)*, in: S. Diehl, editor, *Software Visualization: International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001; Revised Papers*, Lecture Notes in Computer Science **2269**, 2002, pp. 176–190.

[17] Ortony, A., editor, "Metaphor and Thought," Cambridge University Press, 1993, second edition.

[18] Sajaniemi, J., *An empirical analysis of roles of variables in novice-level procedural programs*, in: *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)* (2002), pp. 37–39.

[19] Sajaniemi, J., *Roles of variables home page*, http://www.cs.joensuu.fi/~saja/var_roles/ (2006), (Accessed Aug. 4th, 2006).

[20] Sajaniemi, J. and M. Kuittinen, *Visualizing roles of variables in program animation*, Information Visualization **3** (2004), pp. 137–153.

[21] Sajaniemi, J. and M. Kuittinen, *An experiment on using roles of variables in teaching introductory programming*, Computer Science Education **15** (2005), pp. 59–82.