Animation Metaphors for Object-Oriented Concepts

J. Sajaniemi, P. Byckling, P. Gerdt

Department of Computer Science, University of Joensuu, P.O.Box 111, FI-80101 Joensuu, Finland

{saja|pbyckli|pgerdt}@cs.joensuu.fi

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 (Sajaniemi and Kuittinen, 2004), that builds its visualization on general 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., Eckerdal and Thuné (2005); Fleury (2000); Holland et al. (1997)). Program visualization is supposed to enhance learning and prevent misconceptions but the visualizations should be at the same level as the concepts to be learned. Thus visualizations that build upon programming language implementation may easily fail in helping novices to learn programming concepts.

Metaphor involves the presentation of a new idea in terms of a more familiar one (Carroll and Mack, 1999). In contrast to analogy, metaphor is not an exact counterpart but differs from the idea usually both in form and in content. The similarities and differences between the two ideas stimulate thought and can facilitate active learning. 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 constructs. For this purpose, we present new metaphors for such concepts as object, object instantiation, method invocation, parameter passing, and object reference. 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.

Section 2 describes the new metaphors and explains how they can be visualized and animated in a program animator. Section 3 discusses visualizations in current program animation systems and compares them with our ideas. Finally Section 4 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 (Sajaniemi, 2002) that already have metaphors, e.g., a dog for the role follower, a box for gatherer etc (Sajaniemi and Kuittinen, 2004). For an object, we therefore suggest the metaphor of a watch panel with class-dependent fixed “monitors” for its member variables depicted in the form of role metaphors.

The instantiation of an object is animated by making a copy of a class-specific blueprint found in a blueprint book that is created during the processing of class declarations. 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 either flexible or large enough to accommodate all variables. If another method of the
same object 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 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 tradition-
ally 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;
assignment to 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 animated
program models a bank that consists of three 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 active object (account 1476) and its active method (updateRate) are enhanced with
red color. The workshop for this method invocation contains the parameter transferDate
Figure 1: A hypothetical user interface for a program animator using the OO metaphors.
(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, 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 (Jerding and Stasko, 1994), Jeliot 3 (Moreno et al., 2004), OGRE (Milne and Rowe, 2004), and OOP-Anim (Esteves and Mendes, 2003) 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 (Leroux et al., 2003), JAN (Lohr and Vratislavsky, 2003), and JavaVis (Oechsle and Schmitt, 2002) 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. 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 Conclusion

We have presented new metaphors for classes (blueprint book), objects (watch panel), method invocation (workshop), parameter passing (envelope), return value (workbench) and object reference (pennant). We have also presented visualizations for these metaphors and 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 will give students a correct mental model of the relationships between OO concepts—a mental model that builds up on a spatial representation for 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.
In future, we are planning to add the new metaphors into the PlanAni program animation environment and study their effects on novices’ mental models of OO concepts.

Acknowledgments
This work was supported by the Academy of Finland under grant number 206574.

References


