

# Animated UML as a 3d-illustration for teaching OOP

Uwe Thaden and Friedrich Steimann

Learning Lab Lower Saxony (L3S), Expo Plaza 1, D-30539 Hannover  
{thaden, steimann}@learninglab.de

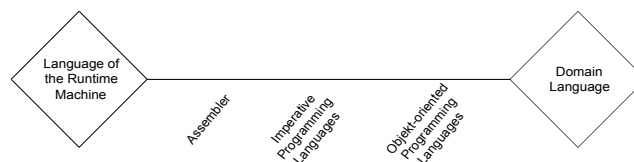
**Abstract.** The classical view of a register-based abstract machine is a barrier for understanding the execution of object-oriented programs. We visualize program execution based on the object-oriented paradigm with its objects and the message flows between them. For this we propose a visualisation for intuitive understanding of object-oriented based program executions. Three dimensional animated illustrations are created from UML diagrams. We do not visualize explaining an algorithm, but giving a tool to aid the intuitive understanding of object-oriented program execution. Our Java prototype uses an XML structure created by "Together" representing UML diagrams to transform into VRML that can be displayed in web browsers.

## 1 Introduction

Software has become and still gets more and more important in all parts of our life. Therefore the need for well educated programmers grows at the same rate. A basic premise for good software development is the ability of programmers to think in the concepts of the used programming language. Software consists of static structures as well as of dynamic sequences. This means that programmers need intuitive models of the actions in a software program. For this reason, such models must exist and need to be trained in education.

## 2 The Problem

The procedural or imperative style of programming is tight-knit with the verbal description of algorithms. Thus, the intuitive process model is a sequence of instructions to an abstract machine. Object-oriented programming emphasises a different aspect: Even though the (abstract) machine is the same as in other paradigms, the main focus is not on the processing of instructions. Important and typical for the object-oriented paradigm is the dynamic creation and destruction of objects, their links, and their interaction through exchanging messages. Normally this kind of thinking is much closer to the part of reality which should be modelled. The sequence of the instructions is distributed between object. Thus, a straight forward understanding or even an idea of the needed underlying abstract machine is not given (Figure 1). The classical idea of a single processor with a fixed number of registers and program counter is not applicable. Normally, object-oriented programming with its interacting objects is understood as much nearer to our conception of reality and consequently more intuitive than other paradigms. All the more remarkable is that a vivid idea of an abstract machine visualising the execution of an object-oriented program is still missing.



**Fig. 1.** When developing a software system, it is necessary to translate the concepts (language) of the application domain to the language of the (virtual or physical) runtime machine. With the arrival of object-oriented programming languages the gap between domain language and programming language has narrowed [11]; on the other hand the task of explaining the runtime mechanisms has become more difficult, since the gap between programming language and machine language has widened.

Of course it is possible to transfer (or map) an object-oriented program to an abstract machine (with imperative character). This will even give a visualization of the execution. But remembering the goal of decreasing the distance between program and reality, this would enlarge the gap again. As said above software developers have

to think in the concepts of the selected programming paradigm. Thus, a translation of an object-oriented program into a procedural sequence does not facilitate the step to learn the object-oriented thinking. It would be more favourable to keep the model character of an object-oriented program relating to the reality and as well (not anyhow) the program execution would be explained on a suited abstraction level.

The object-oriented programming has an intuitive nature for modelling the reality. It seems to be worth examining if this can be used to explain the program execution. To do this we take the Unified Modeling Language (UML) in our approach using its dynamic diagram types (e. g. sequence- and collaboration diagram). Such visualizations of program executions are solely static projections because of the limitations implied by the medium paper and do not stimulate the clearness.

Animated UML has two variants that should be used in education:

1. As instructional films, using separated and commented sequences to show fundamental execution of object-oriented program like instantiation, method calls etc. and
2. for visualization of (at least theoretically) any object-oriented program parts.

### **3 What is animated UML?**

By their nature, animations illustrate dynamic aspects, without excluding static parts. For instance, a class diagram doesn't lend itself to animation (unless illustrating the programming process itself). A combination of class- and object diagram can be animated if the process instantiation and linking between the objects are graphically illustrated. The links between the objects can be seen as pipes for a (potential) message exchange. With this we have the dynamic collaboration diagram that is obviously well suited for animation. In fact the most important metaphor (message exchange) is well suited for creating animations to illustrate program executions. Objects send messages by a kind of letter shoot containing references to other objects as parameters. The recipient of such a message changes its state and/or sends further messages to other objects. Even the instantiation can be illustrated with this metaphor when interpreted as initiated by message to a class. This action creates a new object; a proper metaphor for this you find below.

Even if this animation type (as a pendant to the UML diagram type) visualises a large amount of the object-oriented programming, didactical reasons push to more differentiated animation types each emphasising a special aspect of the execution. Firstly we take a look at:

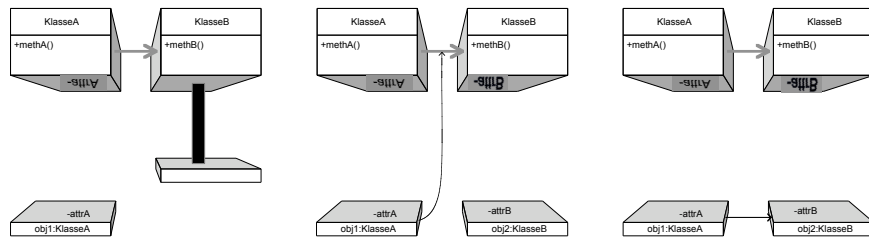
1. Structure- or better instantiation animations that illustrate the type hierarchy and the instantiation of classes to object and the links between them, and
2. interaction animations that visualize the exchange of messages (with parameters).

We want to point out here that we do not want to add to the possibilities of software modelling, but rather build an intuitive notion of the concrete program executions in object-oriented programs. Therefore we can ignore use cases and component diagrams; nevertheless they are very important for the software modelling.

#### **3.1 Instantiation animation**

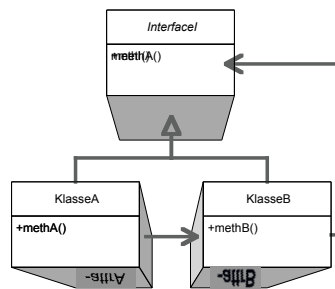
In this illustration the type hierarchy and the process of the object instantiation (creation of objects and the interconnecting links) are pictured. Since links may only be created when an association is declared on the type level, the associations are also drawn.

The German word for instantiation gives an intuitive idea for a metaphor. It implies the German word for "to stamp". This lead us to the animation illustrated in Figure 2. The bottom of the class symbol slants down like a stamp. It coins an object with its attributes on the layer where all objects are placed.



**Fig. 2.** Snapshots of an instantiation animation. First a new object gets stamped, then a link connects. While the classes remain at their positions, the object plain moves so that the objects can be stamped at their correct positions.

This stamp metaphor gives implicit a three dimensional visualization that allows us to give an appropriate illustration of the type hierarchy. The two dimensional UML has the malus of insufficient visual differentiation of first order relations (e. g. associations) and relation of second order (e. g. generalisation). Novices often are confused about that and take the generalisation as a special kind of association (comparable with aggregations). Precursors to UML therefore used a kind of Venn diagrams for the type hierarchy. The three dimensional visualization gives the possibility to do a spatial partitioning so that the super types are arranged above their sub types. The relations of both orders are in different dimensions (Figure 3). An association can extend over more than one level (meaning the connection from super- and sub class). To illustrate this the associations always connect at the side of the type symbols while generalisation resp. implements relations always connect a type top side with a type bottom side.



**Fig. 3.** Type hierarchy, the super types are ordered spatial above their sub types.

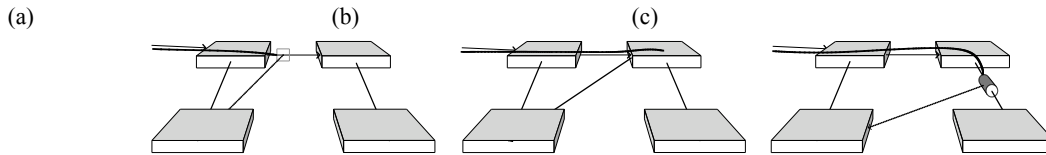
A useful side effect of illustrating class hierarchy ranging over several levels is the ability to show inheritance, e. g. the repetition of inherited associations for sub types without overloading the diagram. Depending on such an option the instantiation of objects can be understood as a single step process (with all attributes at once) or as a sequence of instantiations (the class itself and all its super classes). The latter is equivalent to the recursive invocation of the constructors of all super classes in Java.

### 3.2 Interaction animation

The idea for our interaction animations is taken from the UML interaction diagrams. Since animations are inherently dynamic the importance of the time affects in static interaction diagrams is pushed into the background. They are only relevant for tracing the temporal execution. Especially the vertical time dimension and the corresponding advantage of the sequence diagram deteriorate. If one admits that objects in sequence diagrams are arranged two dimensional as discussed in [3] the sequence diagram viewed in the temporal projection becomes a collaboration diagram (s. discussion in 5.1).

A collaboration animation consists at first of a composition of objects (e. g. emerged from an instantiation animation) and the links between them. Starting from an actor a message is sent from an object to another. Doing this the message creates an Ariadne thread. When receiving the message the object sends new messages along its links and/or acknowledges an internal execution.

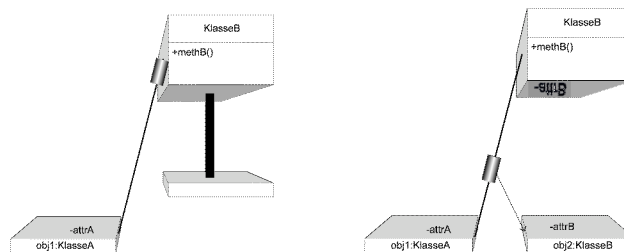
Differing from the collaboration diagram the animation can visualize sent objects (= parameters) as “moving links” (Figure 4 (a)). This parameter can be used to fix a connection between itself and the receiver (b) or passed on (c) or become a recipient itself (not shown).



**Fig. 4.** Various snapshots of a collaboration animation (s. text)

A collaboration animation as shown in Figure 4 shows no instantiations. Normally an instantiation comes along with an initialization of a new instance. Since this initialization is parameterized it is reasonable to integrate an instantiation animation into a collaboration animation. For this the class level has to be made visible and the message (causing the instantiation) has to come from an instance or an actor going to a class.

For the instantiation it is necessary that the initiating object knows the class which will be used to instantiate. But it is not reasonable to create corresponding links since all classes are accessible globally. Anyhow, it makes sense to illustrate the instantiation activated by a message sent to a class; even if e. g. the syntax of object construction in JAVA (`new <classname>([parameter list])`) is not following this clearly. Such a combination of instantiation-/initialization process as a combined instantiation-/collaboration animation is shown in Figure 5; so that the type collaboration animation subsumes the instantiation animation.



**Fig. 5.** Collaboration animation as the process of instantiation

## 4 Current status

In our first approach we use the UML modeling tool TOGETHER (Togethersoft, <http://www.togethersoft.com>) to automatically create class- and collaboration diagrams from an existing source or to create these diagrams manually. The CASE tool can export the diagrams in XMI (=XML Metadata Interchange) format. The XMI type has to be UNISYS 1.1 because only this one exports positioning information about the diagrams as well.

The XMI files represent the information about the individual model elements in a hierarchical manner. We have written a Java application that reads in the information from the generated file. Using the XML parser XERCES (Apache, <http://xml.apache.org/xerces-j/index.html>) a parse tree is built. From this tree a couple of files are generated which contain the VRML (=Virtual Reality Markup Language) code for presenting 3d diagrams (Figure 6). This includes the descriptions of the used geometric objects as well as their positioning- and movement data. Furthermore the information about the so called world system and the coloring and texturing is included.

To combine the geometric VRML primitives to an animation each model element (class, object etc.) has a corresponding Java class. These classes generate 3d representations to the respective nodes of the parse tree. More complicate is the generation of the animation. Though it is possible to define movements in VRML as well we have to distinguish between rotation and scaling initiated by changing the viewpoint of the user, and movements where the object itself changes the position. For the latter VRML offers movement paths, on which the object can change its position. For this purpose there are special scripts commands, which consist primarily of the start- and end point of the movement path and the movement speed.<sup>1</sup>

<sup>1</sup> Furthermore VRML offers various so called sensors to affect an animation, e. g. mouse clicks and object collisions.

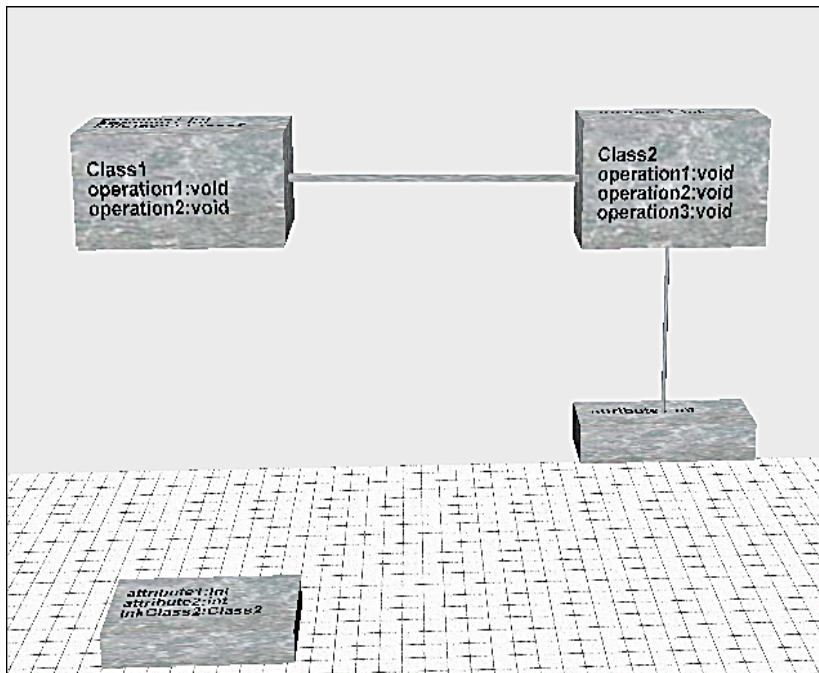
Currently, all animations in our work are generated by our Java program and cannot be controlled from outside.

## Conclusion

We presented a novel way to visualize program execution of object oriented programs. With our approach both teachers and learners of object orientation have a great opportunity to use a 3d graphical animation which emphasizes the intuitive understanding of program executions in their didactical context.

The goal is to replace the still used idea of instruction based machines by an adequate use of UML animation which, following the typical aspects of program executions in OO, combine both the clarification of OO in principle and a concrete visualization of program(-parts) written by learners helping to understand what OO-concepts mean.

With our work we present a step towards new elements for the teaching and learning of object orientation.



**Fig. 6.** Screenshot from an instantiation animation. On the bottom layer an object is already instantiated, while the second class is just moving the stamp to create an instance of its type. Instantiation animation described in 3.1.

## References

1. Beck, K.: Smalltalk best practice patterns. Prentice Hall, Upper Saddle River (1997)
2. DePaus, W., Kimelman, D., Vlissides, J.: Modeling object-oriented program execution. Proceedings of the 8<sup>th</sup> European Conference, ECOOP'94 (1994) 163-182
3. Gogolla, M., Radfelder, O., Richter, M.: Towards three-dimensional representation and animation of IML diagrams. In: France, R., Rumpe, B. (eds.): Proc. 2<sup>nd</sup> Int. Conf. Unified Modeling Language (UML'99) Springer LNCS 1723 Berlin (1999) 489-502
4. Jayaraman, B., Baltus, C.M.: Visualizing Program Execution. Proceedings of the 1996 IEEE Symposium on Visual Languages (VL'96) IEEE (1996)
5. Jerding, D.F., Stasko, J.T., Ball, T.: Visualizing interactions in program executions. 1997 Int'l Conference on Software Engineering (ICSE'97) Proc. ACM Press (1997) 360-370
6. Koike, H., Aida, M.: A bottom-up approach for visualizing program behaviour. In: 11<sup>th</sup> IEEE International Symposium on Visual Languages (1995) 91-98
7. Mellor, S.J., Tockey, S.R., Arthaud, R., Leblanc, P.: An action language for UML: proposal for a precise execution semantics. In: UML'98: Beyond the Notation Springer LNCS 1618 (1999) 307-318
8. Radfelder, O., Gogolla, M.: On better understanding UML diagrams through interactive three-dimensional visualization and animation. In: Di Gesu, V., Levialdi, S., Tarantino, L. (eds.): Proc. Advanced Visual Interfaces (AVI'2000) ACM Press (2000) 292-295

9. Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omotogbe, N.: The Architecture of a UML Virtual Machine. In: Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01) ACM Press (2001)
10. Steimann, F.: Abstract class hierarchies, factories, and stable designs. Communications of the ACM 43:4 (2000) 109-111
11. Steimann, F., Vollmer, H.: Exploiting the theoretical limitations of UML for model validation and execution. submitted to: UML 2003.
12. Züllighoven, H.: Softwareentwicklung. In: Schwabe, G., Streitz, N., Unland, R. (eds.): CSCW-Kompodium. Springer Heidelberg (2001)