Visualizing dynamics of object oriented programs with time context

Filip Grznár *
Faculty of Informatics and Information Technologies
Slovak University of Technology

Peter Kapec†
Faculty of Informatics and Information Technologies
Slovak University of Technology

Abstract

Software visualization has been in focus of researchers for several decades. Although many interesting software visualization systems have been developed very few have managed to become part of common development process. In this paper we focus on program runtime visualization and present our visualization system that interactively presents 3D visualizations visually similar to UML sequence diagrams and we show several example visualizations.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual reality

Keywords: 3D software visualization, runtime visualization

1 Introduction

Software visualization is an ongoing research area that aims at providing helpful insight into software by providing graphical views. The goal is to make software comprehension easier by replacing cumbersome analysis of textual source code by visual analysis of graphical representations of software. The graphical representations of software can greatly utilize human's visual system. By observing software visualizations, hidden relations and structures may become visible. Fast software comprehension is often required for new developers that start working on previously unknown software. For large software systems, e.g. enterprise systems, the amount of source code might be so large, that studying source code would be impractical. Although today software developers often use CASE tools and model software using UML diagrams, still the vast amount of source code makes software comprehension difficult. Understanding the static software structure is only the first step in comprehension. Understanding how the source code behaves during program runtime might be even more difficult. Even simple programs may behave very complex, which is often caused by the heterogeneous data they process.

We must also mention that software changes during the whole development process. These changes need to be tracked, especially for fixing bugs, developing new feature branches etc. For complete understanding developers have to understand also the history of these changes and their causes. In this paper we have focused on the visualization of program runtime of object oriented programs. The following Section 2 mentions related work, especially approaches for program runtime visualization. In Section 3 we present our approach for visualizing the runtime state of object oriented programs in the C# programming language. Section 4 briefly mentions technical details of our visualization system. Section 5 shows several example visualization and is followed by conclusions.

*e-mail: xgrznarf@is.stuba.sk
†e-mail:kapec@fiit.stuba.sk

SCCG 2013, May 1 – 3, 2013, Smolenice, Slovakia.

ACM 978-1-4503-2480-9/13/05 $15.00

2 Related work

A good overview of the software visualization research area can be found in [Diehl 2007], which divides software visualization into three main areas: software *structure* visualization [Teyseyre and Campo 2009][Caserta and Zendra 2011], visualization of *program runtime* and the visualization of software *evolution*.

Program runtime visualization deals with the visualization of program behavior, what instructions are executed and how the program state changes. Part of this visualization may be algorithm visualization [Brown and Sedgewick 1984],[Baecker 1998]. However, the more interesting aspect of program runtime visualization are visualizations of program state (e.g. object diagrams and data structure visualizations [Malloy and Power 2005]) and program state changes, which can be visualized in real-time or post-mortem (the information about the execution of the visualized program is stored e.g. into a file and afterwards played back) [Diehl et al. 2002].

Visualizing program runtime and program behavior of object-oriented systems was in focus of several research projects that inspired our work [Bertuli et al. 2003], [Smith and Munro 2002],[Jerding et al. 1997]. These and other projects usually visualize program runtime using 2D visualization techniques like pattern matrices, histograms, information murals or graphs, but usually do not utilize 3D visualization. The major influence for our work are UML sequence diagrams [Booch et al. 2000] and the works [Mehner and Wagner 2000],[Mehner 2002] in which classic 2D UML sequence diagrams are generated from program traces. A very similar approach for program runtime visualization has been proposed in [Šperka et al. 2010]. It uses a dynamically created 3D visualization of objects (single helix-like visualization) and message passing. However our work differs in two main features: a) we use live program visualization instead the post-mortem approach b) our visualization can also show historical context of method execution and message passing.

3 Visualizing runtime of object oriented programs

Our goal is to provide a visualization that displays the live state of a running program, the existing objects, their inter-dependencies and partially also the historical context of program execution. The proposed visualization focuses on visualizing objects (instances of different classes), threads, access to object attributes, method invocation and message passing.

In the following sections we present our visualization approach for visualizing the execution of programs implemented in the object oriented programming language C#, however the visualization itself can be adapted to other similar object oriented programming languages.

3.1 Visualization approach

Our visualization method was inspired by UML sequence diagrams [Booch et al. 2000] which show an interaction diagram between objects and the sequence of sent messages. We actually developed a 3D version of UML sequence diagrams that are composed and visualized directly during program runtime visualization.

One of the often mentioned advantages of 3D space is that there are several views to the visualization. Different views allow presenting different information about the dynamics of a program. We have developed our visualization to utilize different views to better present interesting program runtime information.

The top view shows objects in a program, their relations (which object created which) and partially the sequence of object creation. The bottom view shows threads in a program and the assignment of these threads to objects. The side view is dedicated to showing the current message, the sender and receiver of this message and message assignment to objects and threads. Important to note is that this side view displays the history of message passing, thus showing time context.

The visualization provides different views in 3D space – these views are not fixed and the user can freely navigate and explore the visualization using a virtual camera. The transition to 3D visualizations is often justified by stating that the added third dimension provides more space for visualization. However the added dimension brings new problems: exploring the visualization and navigating in 3D space is more problematic as 3D visualizations are often more visually complex.

For better usability of the proposed 3D visualization we have implemented various space and time limitations that try to reduce visual clutter. In the following sections we present key aspects of different visualization views and discuss the proposed limitations.

3.2 Visualizing objects – top view

The fundamental elements of object-oriented programming are classes and their instances (objects). In our visualization objects are displayed as hemispheres. The static part of classes is visualized similarly as a hemisphere but is differentiated by yellow color.

The Figure 1 shows the top view of a running program that consist of the main *Program* object that created instances of the *BST* class, which in turn created four *BstNode* instances. The information about which object created which other object is displayed trough oriented arrows.

The amount of objects at program runtime cannot be foreseen, thus the visualization system needs to dynamically create the visualization on the fly. We have developed an object layout algorithm that tries to express one of the most important aspects of program runtime: the sequence of object creation.

When an object creates a second object, the second object is drawn in the scene and is connected with the first object by an oriented edge. The objects create an acyclic oriented graph that is laid out horizontally (the objects may not reside in an equal plane, their vertical position can differ). Objects that belong to the same parental object are drawn near the parental object. This is suitable because the parent object usually has a reference (or pointer) to his child objects and they will probably communicate.

The object layout algorithm is based on a tree layout algorithm that takes into account node sizes with "non-zero" size (the nodes of the tree are objects displayed as hemispheres). The object layout algorithm has been designed with emphasis that:

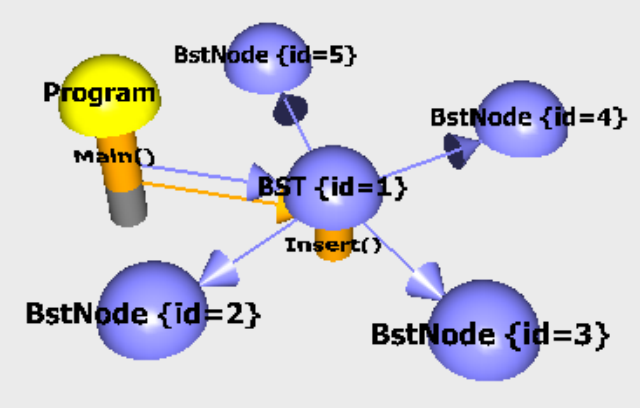


Figure 1: Program runtime with a binary tree data structure.

- the asymptotic time complexity is $O(2n)$ – thus the layout can be recalculated very fast, e.g. after addition of new nodes.
- the parameters of the algorithm (k_1 and k_2) allow setting the minimal distance between borders of nodes. The k_1 is the minimal distance between borders of connected nodes. The k_2 is the minimal distance between borders of unconnected nodes.
- the branches of the tree do not interfere.

The object tree layout algorithm is based on recursive angle calcu-

Require: $RootNode$, k_1, k_2, a, b, c
1: $WeightSubtrees(RootNode, a, b, c)$
2: $RootNode.Position = (0,0)$
3: $RootNode.Angle = 360$
4: $RootNode.Direction = (1,0)$
5: $Direction = RootNode.Direction - RootNode.Angle/2$
6: **for all** $ChildNode$ *in* $RootNode$ **do**
7: $ChildNode.Angle = RootNode.Angle * ChildNode.Weight / SumWeight$
8: **if** $ChildNode.Angle <= 180$ **then**
9: $ChildNode.d = max(RootNode.r + k_1 + ChildNode.r, (ChildNode.r + k_2/2)/sin(ChildNode.Angle/2))$
10: **else**
11: $ChildNode.d = RootNode.r + k_1 + ChildNode.r$
12: **end if**
13: **if** $ChildNode.Angle > 180$ **then**
14: $Alfa = arcsin((RootNode.r + k_2)/ChildNode.d)$
15: **if** $ChildNode.Angle > (360 - 2 * Alfa)$ **then**
16: $ChildNode.Angle = 360 - 2 * Alfa$
17: **end if**
18: **end if**
19: $ChildNode.Direction = Direction + (ChildNode.Angle/2)$
20: $Direction += ChildNode.Angle$
21: $ChildNode.Position = RootNode.Position + ChildNode.d * ChildNode.Direction$
22: **end for**
23: **for all** $ChildNode$ *in* $RootNode$ **do**
24: *repeat for* $(ChildNode, k_1, k_2)$
25: **end for**

Algorithm 1: Algorithm to calculate object positions.

lation based on subtrees. The size of the angle assigned to a subtree is proportional to the weight of the subtree. The algorithm at first calculates the weights of all subtrees using a top-down postorder search and afterwards calculates subtree angles using these weights using a bottom-up preorder search. The Algorithm 1 summarizes this approach.

We calculate the weight of a subtree using following formula:

$$w = n^a h^b r^c \quad h = 1/n \sum_{i=1}^{n} h_i \tag{1}$$

where, n is the count of nodes in a subtree, h is the average depth of a subtree, r is the average radius of nodes in a subtree, a and c are parameters with default value 1 and b is a parameter with default value -1.

3.3 Visualizing threads – bottom/side view

Object oriented programs in the C# programing language often utilize multiple threads to split computation. In our visualization we have integrated the visualization of threads directly with program objects visualization, thus showing which threads work with which objects. The threads are visualized as vertical cylinders that actually display the sequence of methods invoked upon a object. Figures 2 and 3 show eight threads running on the static object *Program* (the ninth is the *Main* method – it is attached to the *Program* hemisphere, see Section 3.4 for explanation) and one thread working on the *MyClass* object.

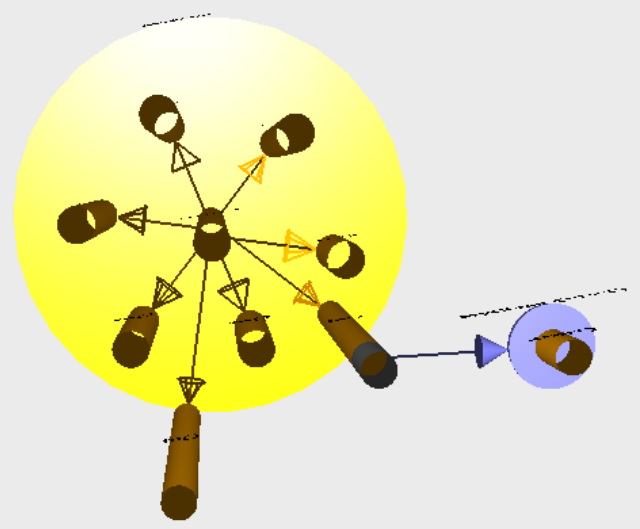


Figure 2: Threads operating on *Program* object – bottom view.

The threads, similarly as methods, are visualized as cylinders – their placement is controlled by an algorithm similar to the placement of objects mentioned above. Threads tend to communicate heavily, so the thread layout algorithm needs to address this and needs to minimize crossing of arrows that represent messages (see next section).

The thread layout algorithm calculates the *communication intensity vector* K for each thread belonging to an object. This vector K specifies the direction and intensity of thread's communication and is calculated using following formula:

$$K(t,S,R) = \sum_{s \in S} \frac{s-t}{|s-t|} + \sum_{r \in R} \frac{r-t}{|r-t|} \tag{2}$$

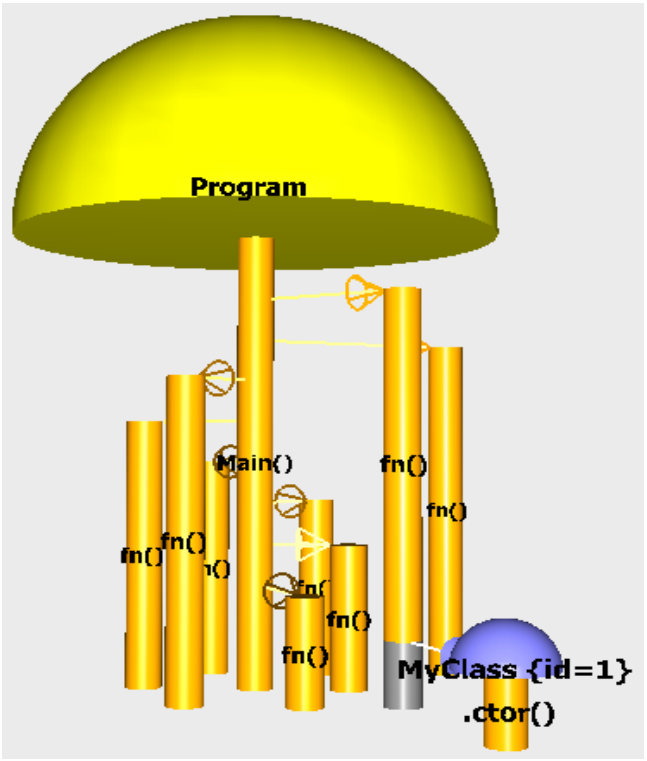


Figure 3: Threads operating on *Program* object – side view.

in which t is the current position of the thread and S is the set of positions of all objects that send messages to the thread; similarly R is the set of positions of all objects the thread sends messages to.

The *communication intensity vectors* are sorted in descending order according their length. The threads are then placed to the first not occupied position (the possible suitable positions are calculated separately) according to this sorted order.

The threads are placed below the hemisphere in concentric circles. The radius of these circles is calculated using the following recursive formula $R_i(k) = R_{i-1} + r_{max_{i-1}} + k + r_{max_i}$ where R_i is the radius of the i-th layer, r_{max_i} is the maximal radius for the thread on i-th layer.

The concrete position on the circle for each thread depends on the actual free space on the circle and the calculated communication intensity vector. We use a helper function that finds the best position on the circle depending on the amount of threads residing on the circle and the calculated communication intensity vector. The Algorithm 2 briefly illustrates this thread layout algorithm.

Require: $Object$, k {k - distance between circles}
1: **for all** $Thread$ *in* $Object$ **do**
2: $Thread.K = K(Thread.Position, Thread.S, Thread.R)$
3: $PriorityFront.Add(Thread, |Thread.K|)$
4: **end for**
5: **while** *NOT EMPTY* $PriorityFront$ **do**
6: $Thread = PriorityFront.Pop()$
7: $NewPosition = CalculateBestPossition(Object, Thread.K, k)$
8: $Thread.Position = NewPosition$
9: **end while**
Algorithm 2: Algorithm to calculate positions for threads.

3.4 Visualizing message passing – side view

The side view is dedicated to message passing and method execution visualizations. This view also provides historical information, as previously executed methods and/or threads are preserved. We see the advantages of program runtime visualization with time context, as we can see:

- the entire sequence of object creation from the first object
- the entire sequence of calls since the start of program to the current state
- recursions (e.g. the recursive *InsertNode* method call in Figure 7)
- the propagation of exceptions (e.g. the propagation of exception call *EqualKey*, shown in red, in Figure 7)
- yet unfinished methods

A message between two objects is visualized by an arrow. We have identified six types of messages (visually differentiated by color):

- creating a new (probably static) object (and optionally calling the constructor method)
- method invocation
- returning from a method
- exception
- read/write access to object's member variables
- deletion of an object

The arrow can start and end at different positions depending on what type the message was. When the message is creating a new object, the arrow ends at object's hemisphere. Arrows for other message types start from cylinders of the object that invoked the message and end at cylinders of the receiving objects.

Figure 7 illustrates visualization of these messages: messages for creating new instances are shown as blue arrows, returning from a method call as gray arrows and exceptions as red arrows. Synchronous messages are drawn with a full cone ending using yellow color as shown in Figure 4. Asynchronous messages are drawn with a outline cone as shown in Figure 5.

Figure 4: Synchronous message visualization.

Figure 5: Aynchronous message visualization.

Variables belonging to an object are visualized as a block to easily differentiate them from methods (displayed as cylinders). To distinguish read/write access to variables, we label the variables with

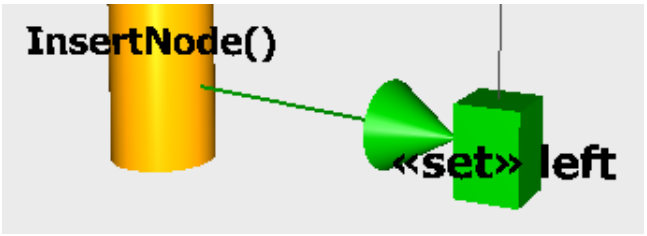


Figure 6: Visualization of variable access.

<< *get* >>, << *set* >> stereotypes. Setting a variable is illustrated in Figure 6: the method *InsertNode* writes to the variable *left*.

The execution of a method is visualized by a cylinder. As the method is executed the cylinder grows downwards by each new method call or method return. If the execution of a method and the creating of an object run together (typical for object constructors), the cylinder touches the hemisphere of an object. This is the case for the *Main* method and for the constructor method of the *EqualKey* object shown in Figure 7. The inactive part of a method (the method part that waits for completion of a another method) is shown with gray color; the active part of a method is drawn in color.

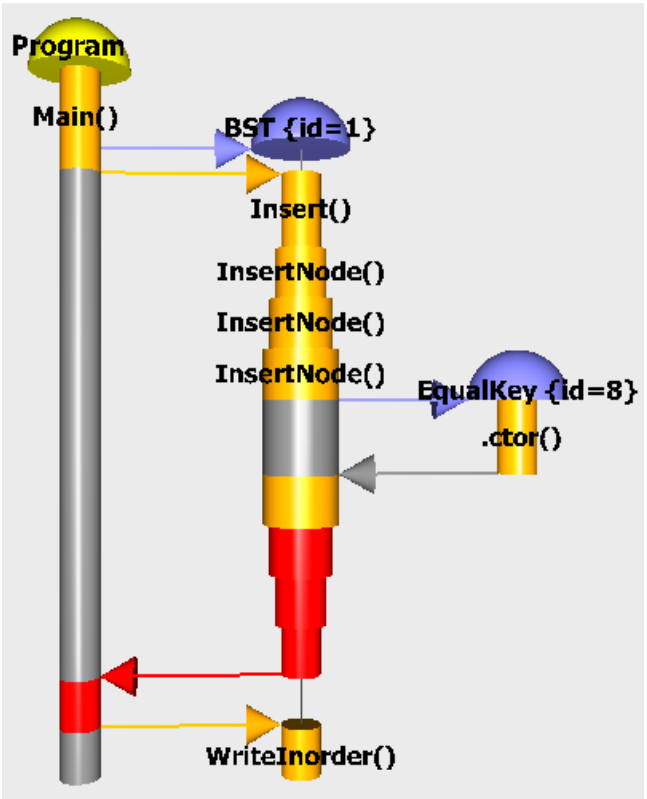


Figure 7: Several methods executed by the *BST* object. Shows recursive method calls and an exception (red arrow).

If a method calls another method of the same object, the cylinder expands and similarly shrinks when execution returns to the first method. This method wrapping is shown in Figure 7: the *Insert* method calls the *InsertNode* method (which in turn recursively calls itself). A cross section of this cylinder wrapping could show the actual method call stack of an object.

3.5 Space and time limitations

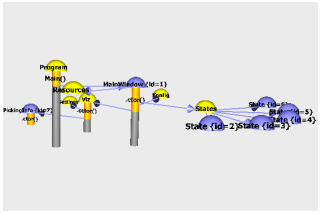
During program runtime visualization the visualization may become very large and difficult to comprehend, especially when we let the visualization system display all messaging history. Therefore, we have implemented several limitations that the user can dynamically and independently turn on and off during program runtime visualization. The space limitations try to reduce space requirements by removing or collapsing not important parts of the object tree. Time limitations reduce the amount of visible history of message passing, method execution and variable access.

The first major limitation the user can enable is to limit the call stack (limiting the vertical grow of cylinders): after enabling, all methods (displayed as cylinders) are removed immediately after they finish their execution. This can significantly reduce the vertical grow of the cylinders. The Figure 8c shows the situation in a program without call stack limitation; the Figure 8a shows the same program runtime state but with the call stack limitation enabled.

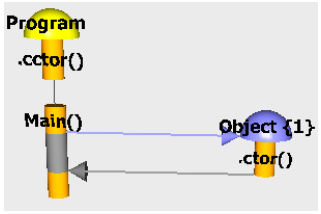
A similarly effective limitation is to disable the displaying of variable access: Figure 8d shows a program state after the main *Program* performed several read and write operations upon the *Object*; the Figure 8b shows the same reached program state but with variable access visualization disabled.

After removing the call stacks, the remaining objects are shown as hemispheres representing the objects itself. In some situations the visualized program can create a vast amount of objects and removing the call stack visualizations might not be enough to cleanup the visualization. Therefore we allow to hide all objects that are currently not active. Figure 9 shows in the top image the possible situation with many objects that have been created – this view might be useful to track object relations (which object created which instance), but might not be well suited to analyze the current method execution of the currently active object. The bottom image in Figure 9 shows directly the two communicating objects without the visual ballast of not active objects.

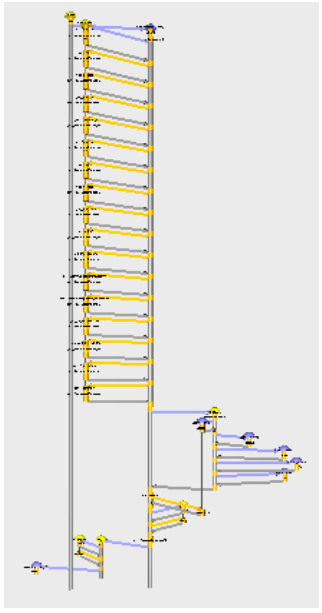
Another visual optimization is to place objects, with hidden cylinders, onto a horizontal plane that is on the same level as the "parent" object that created these objects, see Figure 10 bottom image. This may come handy when an object intensively creates new object in sequence. However, the default single helix visualization shown in Figure 10, top image, clearly shows the time context: the top objects where created first, the bottom as last.



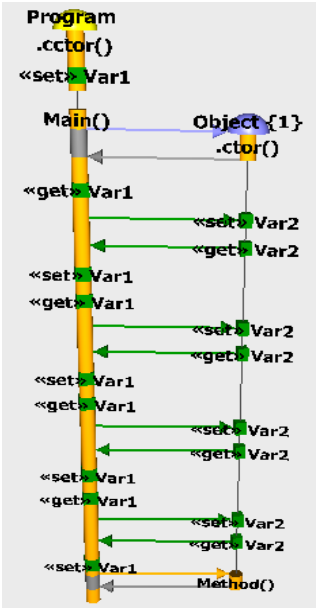
(a) Limiting the call stack visualization.



(b) Showing variable access disabled.



(c) Call stack limitation disabled.



(d) Showing variable access enabled.

Figure 8

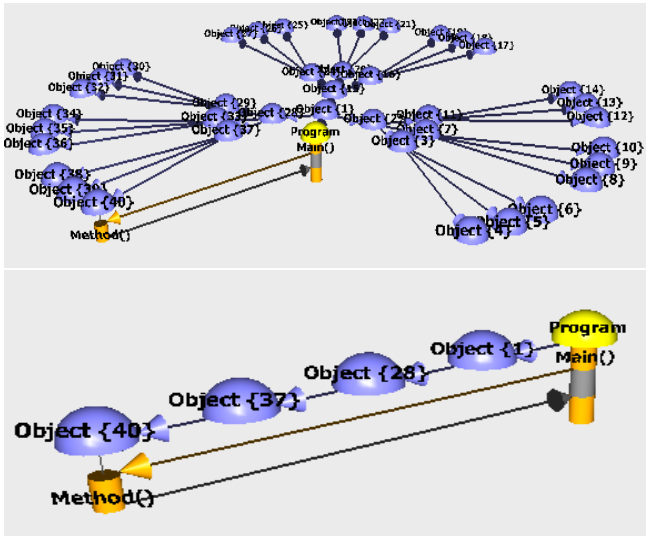


Figure 9: Hiding empty objects (top) disabled (bottom) enabled.

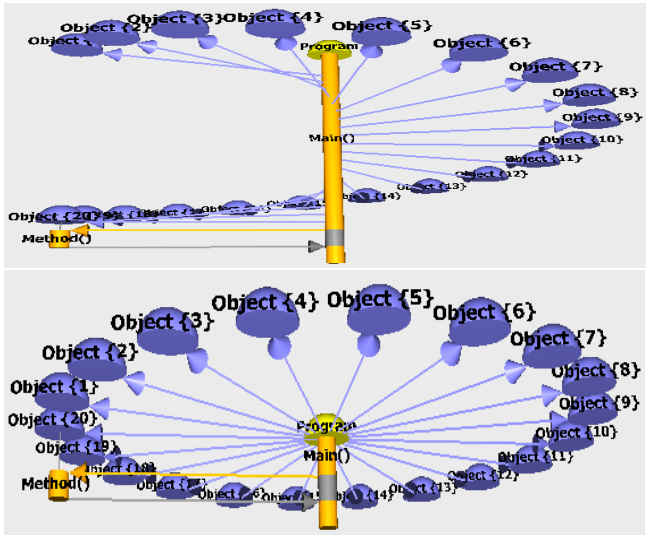


Figure 10: Elevating empty objects (top) disabled (bottom) enabled.

4 Technical details of the visualization system

Our visualization system was developed with modularization in mind. The main visualization program is independent from the visualized program. Figure 11 illustrates architecture of our system.

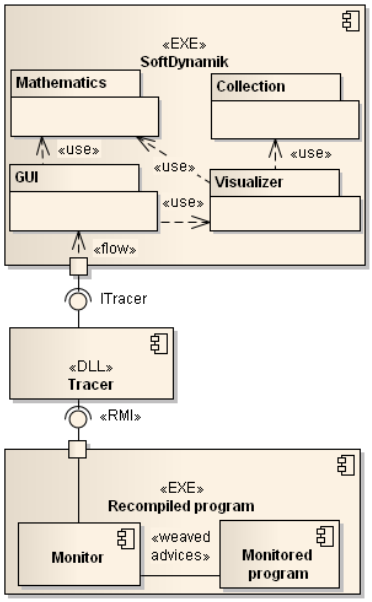


Figure 11: Architecture of our visualization system.

The visualization program, called *SoftDynamik*, is responsible for the whole runtime program visualization. The source code of the *Monitored program* is enriched with necessary functionality (using aspect oriented programming) so it can communicate with our *Monitor* component that is responsible for capturing and monitoring all the relevant program runtime information needed for visualization.

The *Monitored program* and the *Monitor* components are compiled into executable form. This executable then uses the *Tracer* dynamic-link library to communicate with the visualization program *SoftDynamik*. The addition of the *Tracer* allows in the future to implement other monitoring components for other programming languages without the need to modify the visualization program. The *Tracer* specifies an *ITrace* interface that all future monitoring components need to implement. The visualized program uses remote method invocation (RMI) to communicate with the *Tracer*.

The visualization system is capable to visualize any programs written in the C# programming language. The source code is compiled and executed – the visualization happens directly at program runtime (as opposed to post-mortem visualizations using previously saved information about program execution). This actually allows to visually debug the running program by providing detailed information about the visualized objects. The visualization needs to be significantly slower as the possible program execution speed to allow seeing what is actually happening. This of course slows down the program execution speed of the visualized program, but the program is still interactively operational. For convenience we provide a user controlled update timer that can change the speed of the visualization and execution of the visualized program.

5 Example visualizations

In this section we demonstrate the features of our visualization system on several examples. In the previous sections the Figures 1,7 show the visualization of a program manipulating a binary search tree. We have visualized the common *observer* pattern. The Figure 12 shows how the static method *Main* of the *Program* class created the *Subject* object as first, followed by the creation of two *Observer* objects.

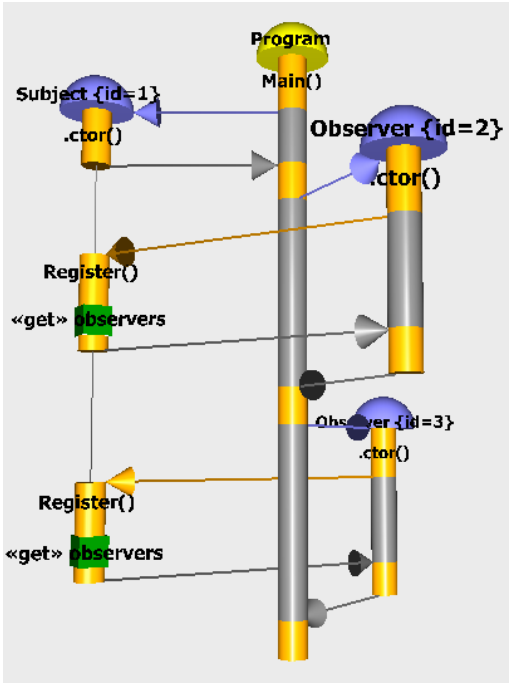


Figure 12: Observer pattern – *Observer*s register.

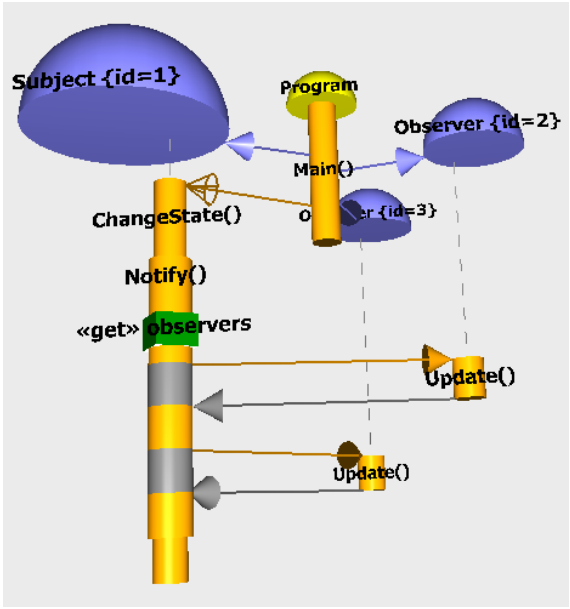


Figure 13: Observer pattern – the *Subject* notifies the *Observer*s.

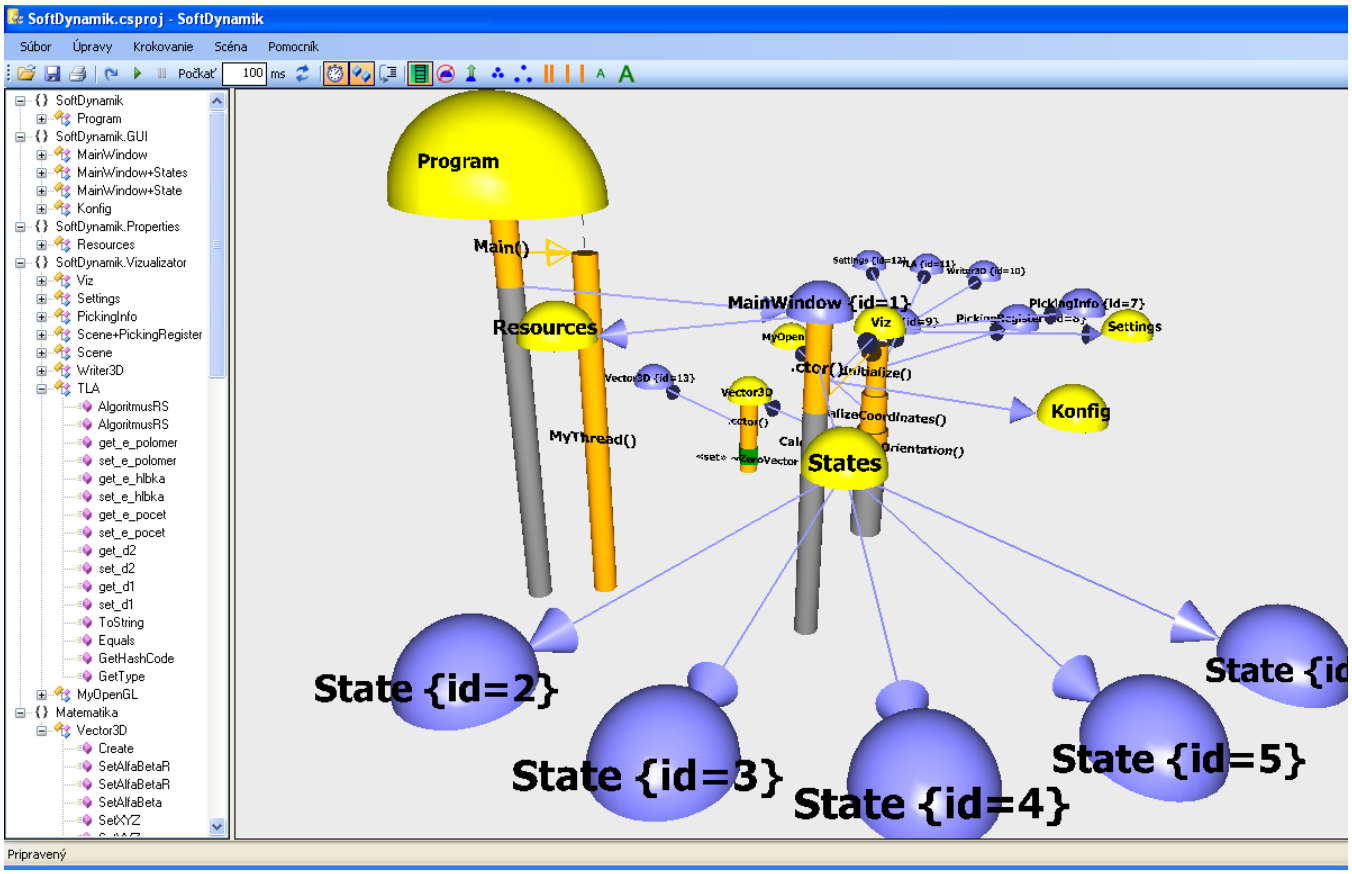


Figure 14: Our visualization system *SoftDynamik* visualizing itself. The left tree view shows classes from *SoftDynamik*'s source code.

Both *Observer* objects, in their constructors, call *Subject*'s *Register* method, which in turn accessed and updated the *observers* variable. This clearly illustrates the typical behavior of the *observer* pattern. The second typical behavior of the *observer* pattern is *notification* as shown in Figure13. The *Program*'s *Main* method asynchronously invokes the *ChangeState* method of the *Subject* that calls the *Notify* method, which in turn invokes the two *Update* methods of the *Observer*s. The Figure 13 shows that the *ChangeState* and *Update* methods run as individual threads (because their cylinders are not directly attached to hemispheres). After the asynchronous *ChangeState* call the *Main* method continues and finishes before the threads in the *Subject* and *Observer*s finish. The Figure 13 thus clearly illustrates asynchronous message passing and multiple treads.

To illustrate the capabilities of our visualization system we have utilized it to visualize itself. The Figure 14 shows the visualization of our *SoftDynamik* visualization system. The left tree-view displays all the namespaces, classes and their attributes and methods, of the visualized program. The visualization does not show all these classes, but only the currently active objects of the visualization systems that is executed in a separate process.

6 Conclusions

In this paper we have presented our program runtime visualization system. The visualization tries to provide the expressiveness of UML sequence diagrams extended to 3D. The visualization is done on-the-fly directly as the visualized program is executing, which allows visual debugging. The most interesting feature is the showing of historical context, because we can observe the sequence of method passing and also method execution. We showed several practical visualizations of real-world scenarios (e.g. the observer pattern) and also the capability of the visualization system to handle larger programs (visualizing runtime of the visualization system). Future work will be directed to end-user evaluation in the learning process, where we see a big potential for illustrating program runtime visually to students of programming courses.

Acknowledgements
This work was supported by the KEGA grant 068UK-4/2011: Integration of visual information studies and creation of comprehensive multimedia study materials.

References

BAECKER, R. 1998. Sorting out sorting: A case study of software visualization for teaching computer science. *Software Visualization: Programming as a Multimedia Experience 1*, 369–381.

BERTULI, R., DUCASSE, S., AND LANZA, M. 2003. Run-time information visualization for understanding object-oriented systems. In *International Workshop on Object-Oriented Reengineering*.

BOOCH, G., JACOBSON, I., AND RUMBAUGH, J. 2000. Omg unified modeling language specification. *Object Management Group ed: Object Management Group*, 1034.

BROWN, M. H., AND SEDGEWICK, R. 1984. *A system for algorithm animation*, vol. 18. ACM.

CASERTA, P., AND ZENDRA, O. 2011. Visualization of the static aspects of software: a survey. *Visualization and Computer Graphics, IEEE Transactions on 17*, 7, 913–933.

DIEHL, S., GÖRG, C., AND KERREN, A. 2002. Animating algorithms live and post mortem. *Software Visualization*, 46–57.

DIEHL, S. 2007. *Software Visualization*. Springer London, Limited.

JERDING, D. F., STASKO, J. T., AND BALL, T. 1997. Visualizing interactions in program executions. In *Proceedings of the 19th international conference on Software engineering*, ACM, 360–370.

MALLOY, B. A., AND POWER, J. F. 2005. Using a molecular metaphor to facilitate comprehension of 3d object diagrams. In *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, IEEE, 233–240.

MEHNER, K., AND WAGNER, A. 2000. Visualizing the synchronization of java-threads with uml. In *Visual Languages, 2000. Proceedings. 2000 IEEE International Symposium on*, IEEE, 199–206.

MEHNER, K. 2002. Javis: A uml-based visualization and debugging environment for concurrent java programs. *Software Visualization*, 643–646.

SMITH, M. P., AND MUNRO, M. 2002. Runtime visualisation of object oriented software. In *Visualizing Software for Understanding and Analysis, 2002. Proceedings. First International Workshop on*, IEEE, 81–89.

TEYSEYRE, A. R., AND CAMPO, M. R. 2009. An overview of 3d software visualization. *Visualization and Computer Graphics, IEEE Transactions on 15*, 1, 87–105.

ŠPERKA, M., KAPEC, P., AND RUTTKAY-NEDECKÝ, I. 2010. Exploring and understanding software behaviour using interactive 3d visualization. In *ICETA'2010: 8th International Conference on Emerging eLearning Technologies and Applications*, elfa, s.r.o., 281–287.