Online-Configuration of Software Visualizations with Vizz3D

Thomas Panas Rüdiger Lincke Welf Löwe

Software Technology Group
MSI, University of Växjö, Sweden
{Thomas.Panas|Rudiger.Lincke|Welf.Lowe}@msi.vxu.se

Abstract

A software visualization is defined by an abstract software model, views on this model and a mapping between them. For creating new visualizations, we online-configure views and their mappings instead of hand-coding them. In this paper, we introduce an architecture allowing such an online configuration and, as a proof of concept, a framework implementing this architecture. In several examples, we demonstrate generality and flexibility of our approach.

1 Introduction

Software visualization is a goal-directed task. On different levels of abstraction, it supports humans in e.g. learning, understanding, reengineering, or communicating software. Depending on the goal, the abstraction level, and even the humans involved, different software visualizations are appropriate. The large number of possible variations makes it expensive to build predefined tools for each task. Instead, the suitable visualization ought to be *developed* on demand. A tool or framework supporting this idea may have a set of predefined appropriate visualizations. However, more important, it must be easy to configure and extend.

Separation of concerns and the reuse of predefined components implementing these concerns increase the efficiency of software development in general. Two concerns for software visualizations are obviously the abstract *model* of the software to be visualized and the actual *view* on this model. The former needs to capture the information required for the software visualization task. The latter needs to create the image of that information for the humans involved.

Separating the model from the view allows reusing and recombining models (and analyses creating them) and views thereof. To create a new suitable software visualization, a *mapping* between model and view must be defined. This mapping connects two worlds, the *domain of software analysis* and the *domain of information visualization*.

Within the domain of information visualization, different *images* can illustrate the same information. Again, there is no single image suitable for a certain view. Instead, the image needs to be adapted to the humans involved. Hence, we also distinguish a configurable *scene* of a view that finally represents an image from the view itself.

Distinguishing different scenes for a view is not only increasing the flexibility, it also increases reusability of certain concerns and components: the same model, view and model-to-view mapping can be reused in different information visualization tools. In fact, each tool defines its own scene data structure. This structure is easy to construct from a view by a one-to-one mapping. Figure 1 summarizes the relations between models, views, and scenes. Hence, we have a model-view-scene controller architecture that is online-configurable, i.e. we can map model-to-view and view-to-scene online and do not hard-code the mapping. This refines the general model-view controller architecture, which corresponds, in our terms, to a model-scene controller.

The contribution of this paper is an architecture allowing to online *configure* the mappings between model, view and scene instead of programming them. Altogether, this enables to interactively develop software visualizations in order to get the appropriate one. We have implemented this architecture in a framework called Vizz3D serving as a proof of concept.

To be able to configure a mapping between the model for software visualization and the view on that model, we need to study both data structures, model and view, which we do in Section 2 and Section 3, respectively. Then, in Section 4, we define the actual configurable mapping between them. A view is further mapped to a scene, as discussed in Section 5. Examples are presented in Section 6. In Section 7, we describe our approach of online-configuring visualizations with the Vizz3D framework. Section 8 discusses related work and Section 9 concludes the paper.

2 Models for Software Visualization

As discussed before, depending on the software visualization goal, different models of the software to be visualized are appropriate. Since these goals are quite different, data types of possible models appear rather heterogeneous, too. A call graph data type for visualizing a software architecture, e.g., is quite different from a data type that captures a model for animating a sorting algorithm.

For creating a configurable online mapping between model and view, it is, however, necessary to find a suitable data type, capturing all possible models (or at least a relevantly large number thereof). We are able to define such a data type when we assume that matter of software visualizations are: (i) program structures and program (structure) changes over a number of development cycles, and (ii) program behavior.

Program structures (i), in general, can be captured by entities and relations between them. Their changes over a number of development cycles add just new relations, as e.g. ”evolves-to” or ”developed-by”, to the picture. Program behavior (ii) is a change of program state over time. State is described as objects and their mutual references and attribute values, i.e. again entities and relations. State transitions add just a new temporal relation.

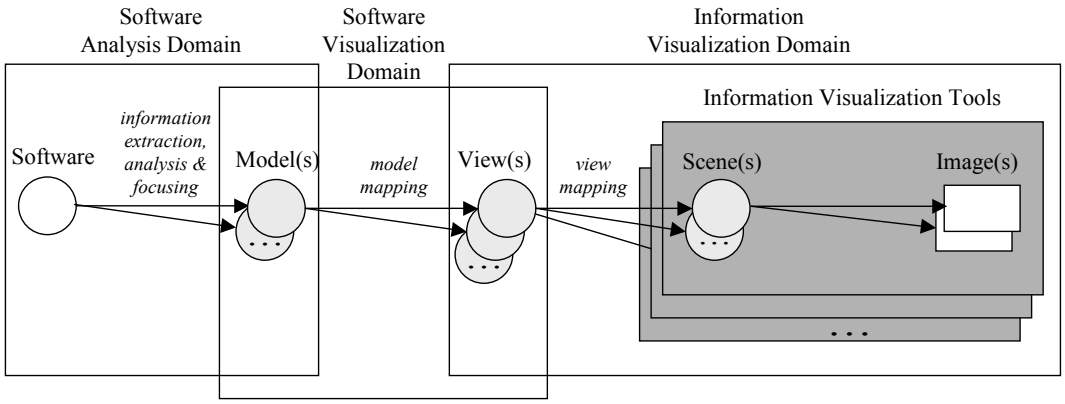

Figure 1: Data and Mappings in Software Visualization.

Obviously, entities of different type and relations (between them) of different type capture the models for software visualization; if not all then at least a relevant subset thereof. In our architecture, we therefore assume a general *graph structure* holding the model. This graph structure consists of:

- nodes modelling entities,
- edges between two nodes modelling binary relations (*n*-ary relations are modelled by introducing special nodes),
- predefined ”label” and ”type” relation for nodes and edges,
- arbitrary properties of nodes and edges modelling relations between entities and values, e.g. metric values.

This graph structure defines the domain for our mapping. It is the responsibility of a developer of a visualization to define an instance of that graph structure, i.e. to define concrete node and edge types and concrete properties. This model needs to be justified against the software visualization goal. From the viewpoint of a software visualization architecture and framework, respectively, the semantics of these nodes, edges, and properties is irrelevant as long as it captures all information necessary for a particular visualization.

3 Views for Software Visualization

Since software visualization views should create a meaningful image of the software model, we can assume that even the views contain entities and relations. Hence, again a graph structure is suitable to capture the view information. We call this the *view graph* structure to distinguish it from the previously defined *model graph* structure capturing the model information.

However, types and properties of nodes and edges are implicit in the humans' visual language. This language is finite in our world. We distinguish:

- *x*, *y*, *z* positions of nodes and edges,
- height, width, depth of nodes and edges,
- color, shape and texture(s) of nodes and edges,
- labels of nodes and edges, and
- their changes over time (temporal relation).

In the view graphs, nodes and edges have a *fixed* set of visual properties as opposed to the free set of properties in the model graphs. Additionally, there is only one single predefined temporal relation, i.e. a special property, in the view graph structures reflecting the change of a view over time. Note, that the model graph can contain any number of temporal relations.

With this visual language, we can depict a (finite) number of structural and temporal relations. The appropriateness of such a view needs to be justified against results from cognitive sciences. Again, from the viewpoint of a software visualization architecture and framework, respectively, the appropriateness of a view graph and its visualization is not relevant. Relevant is just that it captures all potential views one could design (or at least a relevantly large subset thereof).

4 Mapping between Model and View

The model mapping is a relation between model graphs and view graphs, c.f. Figure 1. In order to display a view graph, all of the view graph properties need to be defined. Hence, the mapping needs to be surjective. On the other hand, not all properties of the model need to be depicted in a later image. The mapping is therefore not necessarily complete in the set of model properties.

Usually, not every view graph property is related to a property of the model. Instead, because of the limited cognitive ability of humans, certain view properties get a default value. Again, from the viewpoint of the framework constructors, it is not on us to propose a decent or appropriate number of model properties related to visual dimensions. We just provide the possibility to map certain visual dimensions to default values.

For example, it is disputed if the third spacial dimension supports cognition or if it is just contra-productive. However, until this dispute is decided, if at all, we provide the option to set the *z*-coordinate and the depth of objects to zero or, alternatively, relate it to properties of the model.

From a theoretical point of view, if all values of a certain view graph property are set to the same value then this dimension does not contain any information. From a practical point of view, the selection of this default value needs to be justified again w.r.t. results from cognitive sciences.

A model mapping can now be defined as a function from the domain of model graphs to the co-domain of view graphs. This function, for instance, may specify how to bind a model node property, e.g. metric, to a visual dimension, e.g. a color. In general, the mapping

function takes a complete model graph as an input and deliverers a complete view graph.

This general approach is acceptable for programmed mappings. Since we want to configure them, we need to restrict this generality:

- Nodes of the model graph are always mapped to nodes of the view graph.
- Properties of nodes of the model graph are mapped to properties of nodes of the view graph or ignored.
- Edges of the model graph are mapped to node properties or edges of the view graph.
- Edge properties of the model graph are mapped to edge properties of the view graph or ignored.

Such a restricted mapping is now easily defined in two steps. First, the view graph is created from the model graph by copying relevant nodes and edges, and ignoring the others. Second, model properties are mapped to view properties. This requires a number of *model binding functions*, one for each visual dimension. Each binding function maps model property values to view property values. In the special case where a view property gets a default value, the binding function is a constant function. Note that focusing, e.g. filtering of nodes and edges, can be performed beforehand through various analyses applied on the model graph.

An online-configuration of the first step of the mapping just requires the specification of relevant node and edge types of a model graph. In order to online-configure the second step of the mapping, we predefine a set of reusable binding functions for mapping model to view graph property values. These functions are parameterized and allow for an online adaptation of the mapping function. The mapping itself is now configured by assigning to each view graph property (a) a model property and (b) a suitable function (including their parameters). So far, we have defined the following parameterized binding functions:

Linear binding: maps numerical model properties linearly to numerical view properties. It is parameterized by minimum and maximum model property values, min_{model} and max_{model}, and corresponding view property values, min_{view} and max_{view}. Value min_{model} and smaller model property values are mapped to min_{view}. Value max_{model} and larger model property values are mapped to max_{view}. In between, values are mapped linearly.

Switch binding: maps numerical model properties to view properties. It is parameterized by a switching value c and two view property values a_1 and a_2. Any model property value smaller or equal c is mapped to a_1, the others to a_2.

Interval binding: maps numerical model properties to view properties. Generalizes on the switching binding and is parameterized by n switching values $c_1 \dots c_n$ and $n+1$ view property values $a_1 \dots a_{n+1}$. Any model property value larger c_{i-1} and smaller or equal c_i is mapped to a_i, values larger than c_n to a_{n+1}, values smaller or equal c_1 to a_1.

Classification binding: maps any kind of model properties to view properties. It is parameterized by n view property values. Identical model property values are mapped to identical view property values. Only if more then n different model property values are found, same view property values are used for different model property values. The *color classification* binding is a special case classifying model values by a number of color property values. The *index binding* is another special classification binding mapping same model property values to the same integers.

String filter binding: maps string valued model properties to string valued view properties. It is parameterized by a regular expression e and a string s. Whatever is matched by e in the model property string value is replaced by s in the view property string value. The remainder is copied. Since labels of nodes and edges are the only string valued view properties, this mapping is used to create labels.

Copy binding: just copies model properties values to view property value.

Constant binding: ignores model properties and sets a view property to a default value.

New binding functions can easily be added to the Vizz3D framework if required, c.f. Section 7. They just need to implement the binding interface that defines an initialization and a bind function. The former takes a number of function parameters in the C `argv` style. This allows to online parameterize and adapt the binding function. The latter accepts model property values and returns corresponding view graph property values.

5 Mapping between View and Scene

The view graph is still abstract in the sense that it does not refer to visual objects that are directly drawable in visualization tools. Regardless of the information the view graph contains, it should be possible to implement (or rather encode) view graph nodes and edges and their properties in terms of tool specific objects and properties. We refer to this tool specific data structures as *scene graphs*. For example, in OpenGL and Java3D, etc. nodes and edges have their specific representations. Such scene graphs can finally be viewed as an image.

We refer to the mapping from view to scene graph as the *view mapping*, c.f. Figure 1. It is defined by choosing:

- visual node and edge objects from a family of such objects fitting together, referred to as a *metaphor* and *metaphor binding*, respectively,
- a *view binding* between nodes, edges, and their properties in the view and the corresponding scene graph entities, and
- a *layout* algorithm (optional).

For different abstract node and edge shapes, textures and colors, a visual representation must be chosen. Since they should harmonize, visual representations of a scene should be from one specific metaphor, i.e. a visually harmonic family of such objects. For instance, one metaphor could specify that nodes are represented as rectangles and edges as lines (of different shape and color); in another metaphor nodes are houses and edges streets. It is not recommended (though it is technical not a problem) to connect boxes and spheres with streets.

In addition, for each tool, the properties of the view graph need to be bound to tool specific properties of the scene graph, e.g. height to its respective encoding in the specific scene structure. If required, the user can re-bind the view properties arbitrarily, e.g. re-bind the property height to depth. The same binding functions as defined in Section 4 are applicable to properties in the scene graph.

As discussed before, certain properties of a view graph might have not been defined by model properties but were assigned default values. But, this approach of assigning default position $x = 0$, $y = 0$, and $z = 0$ for all nodes, would create obviously unappropriate views, i.e. views that are hard to comprehend from a cognitive

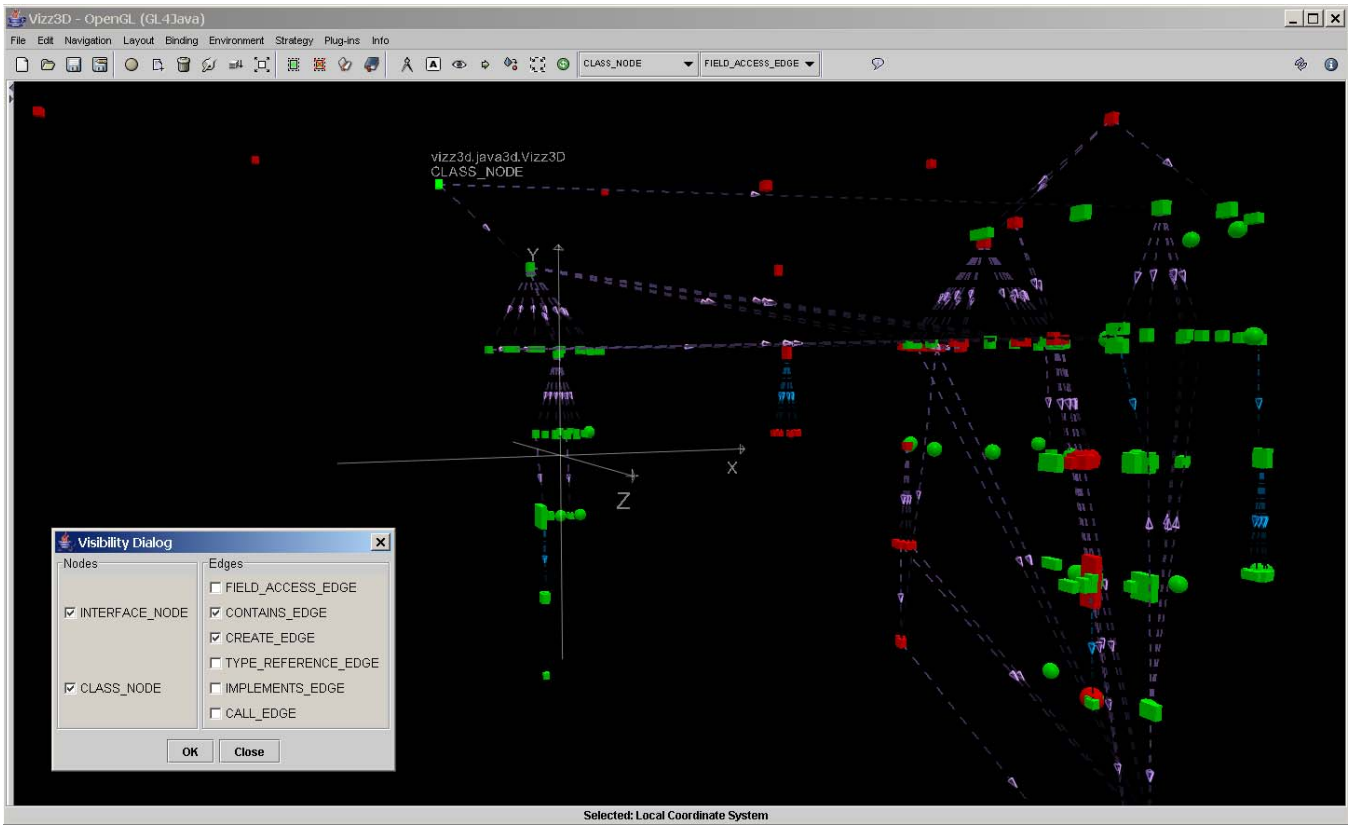

Figure 2: Hierarchy and Class Usage Graphs merged.

point of view. Hence, if *x*, *y*, and *z* are not containing model information, i.e. *x*, *y*, and *z* coordinates are the same (in our case 0) for all nodes, then an automatic layout should assign positions. The theoretically possible case that *x*, *y*, and *z* are actually bound during the model mapping but always to 0 is rather unrealistic in practice and therefore ignored in our framework.

We predefined a number of standard layout algorithms such as spring embedders, hierarchical layouts and checkers (sorting nodes w.r.t. other view properties in 2D or 3D). Our framework is currently restricted in the sense that edge (bend) positions cannot carry model information. Although theoretically possible, we do not consider this a severe restriction.

Metaphor and layout are not orthogonal dimensions. Instead a metaphor may restrict the applicable layouts and vice versa. For example, the metaphor with houses and streets requires a 2D layout. A 3D spring embedder would not be appropriate.

6 Examples of Software Visualizations

This section defines a number of software visualizations in terms of their *model*, *model mapping* and *view mapping*. Each example concludes with a brief *interpretation* of the picture in the model domain. All examples are configured with Vizz3D, our framework, which is discussed in more detail then in Section 7.

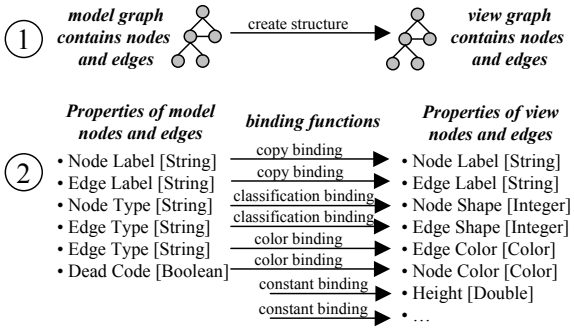

Figure 4: Example Model Mapping

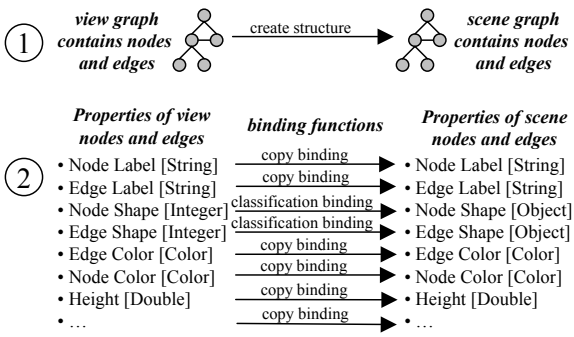

Figure 5: Example View Mapping

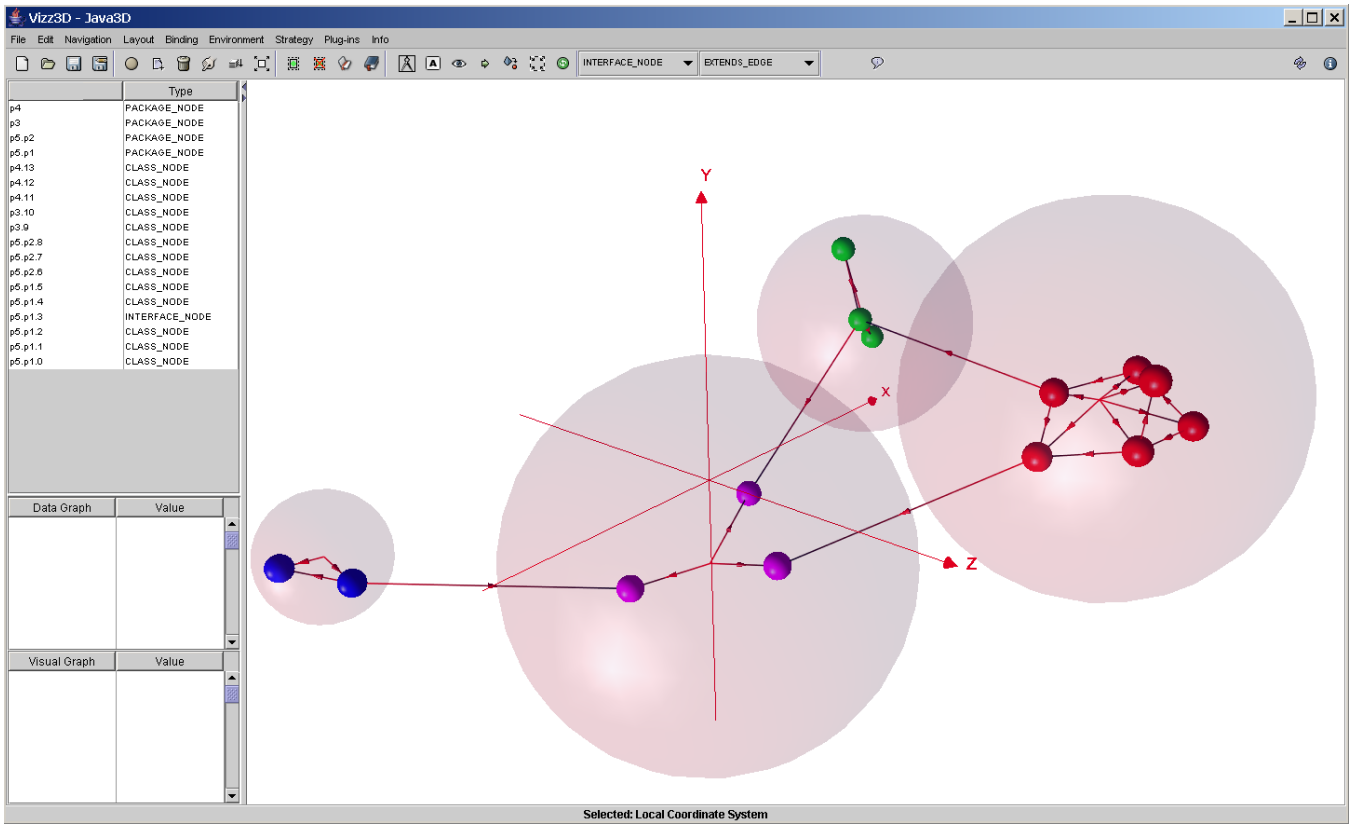

Figure 3: Package Graph Visualization.

6.1 Visualizing Class Interaction

Figure 2 shows a software visualization of the Vizz3D program code. The model captures the class hierarchy structure and class usages. Hence, the *model graph* contains class and interface nodes and field access, creates, and call edges. Node properties are label, i.e. fully qualified class/interface name, type, i.e. "class" or "interface", and a boolean property indicating dead code.

The *model mapping* completely copies the model graph to a view graph structure, c.f. Figure 4 step 1. Then it is defined by the following bindings, c.f. Figure 4 step 2: Node and edge label properties of the model graph are mapped to the node and edge label properties of the view graph; values are just copied (copy binding). The model node type property is bound to the view node shape property; "class" to "1", "interface" to "2" (classification binding). Accordingly, edge type values of the model graph are bound to both, edge shape and color values of the view graph, e.g. field accesses edges to blue lines of "1" (classification/color binding). The dead code property of model nodes is bound to the color property of the view nodes; dead code is red and reachable code green (color binding). The other view properties, i.e. texture, height, width, depth, and positions of view nodes and edges, get a default value (constant binding).

In the *view mapping*, nodes and edges are just copied again in order to create an abstract structure of the scene graph, c.f. Figure 5 step 1. Then it is defined by the following bindings, c.f. Figure 5 step 2: labels of view nodes and edges are copied to labels of scene nodes and edges. The node shape property is bound to the scene object type. Different shape values are bound to different visual objects in the scene taken from a particular metaphor, i.e. "1" to "box", "2" to "sphere". The edge shape property is bound to the scene object type, its values mapped to visual objects chosen from the same metaphor; e.g. "1" to "solid line". Node and edge colors are just copied to the corresponding scene node and edge colors. The properties with default values like texture, height, width and depth are mapped to corresponding scene graph properties; their values to default values in the scene graph. The *x*, *y* and *z* properties are calculated by a hierarchical layout algorithm.

Figure 2 can now be *interpreted* as the interaction of classes (boxes) and interfaces (spheres). The red color indicates nodes that are not used anymore (dead code) [1]. Certain edge types are hidden online in order to restrict the amount of information visible at one time.

6.2 Visualizing Package Interaction

Figure 3 is constructed from a *model* containing class and package nodes and call and (package-)containment edges, indicating to which package a class belongs. Model properties are the node and edge labels, their type and information about the package a class belongs to (package property).

The *model mapping* is defined as a copy operation of the model graph structure to the view graph structure (view graph creation). Then the model property label is ignored. The node and edge type

[1] Actually, the dead code analysis started here at the entry point of the Java3D version of Vizz3D. Therefore, all the OpenGL related classes appear to be dead code.

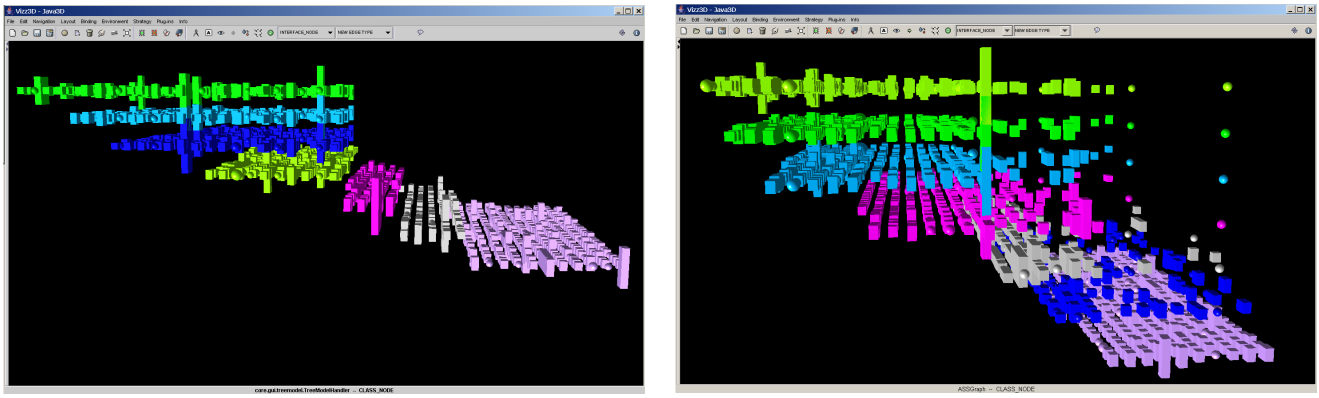

Figure 6: Program Evolution. Different versions of the VizzAnalyzer.

properties are bound to node and edge shapes; values are bound by the classification binding. The values of the package property are bound to the node color property; values are mapped with the color binding, i.e. classes with the same package property value get the same color. The other view properties like texture, height, width, depth, and *x*, *y*, *z* coordinates get a default value.

The *view mapping* is defined as a copy of the view to scene graph structure (scene graph creation).The values of the node shape property are bound to visual object types, i.e. “1” to “solid sphere”, “2” to “transparent sphere”. The values of the edge shape property are all bound to the visual object “solid line”. The *x*, *y* and *z* coordinate properties are re-calculated by an automatic layout algorithm. The size property of the class nodes (alias “1”, alias “solid sphere”) is constant; the size of the package nodes (alias “2”, alias “transparent sphere”) is defined within the layout algorithm. Note that the recalculation of *x*, *y*, *z*, and size of the visual nodes in a layout is admissible since these properties did not encode model properties. Other scene properties like texture and color just get the scene graph default property values.

Figure 3 can now be *interpreted* as the interaction of classes (solid spheres) belonging to packages (transparent spheres enclosing classes of the same package; same color encodes the same package, as well).

6.3 Visualizing Program Evolution

Figures 6 illustrate the evolution of a system over seven versions.

The *model graph* was initially merged out of seven model graphs, each representing one version. In the model, nodes represent classes and interfaces. It does not contain any edges. The node properties are label and type, *LOC* indicating the lines of code for each class/interface and *Version*, indicating to which version a node belongs to.

The *model mapping* is defined by a complete copy of the model and the following bindings: Labels and types are mapped as before. The *LOC* is bound to the height property; values are mapped using a linear binding. The version model property is bound to the temporal property, i.e. ”version 1” to ”1”, ”version 2” to ”2”, etc.; values are mapped with the classification binding. At the same time, the version model property is also bound to the *y*-coordinate view property; values are mapped with the classification binding, i.e. “version 1” to $y = 1$, “version 2” to $y = 2$, etc. The model node label property is mapped to the *x*-coordinate view property; values are mapped with a classification binding, i.e. different labels get different *x*-coordinates. Other properties, i.e. texture, width, depth, and *z*-coordinates get default values.

The *view mapping* binds the node shape property to visual objects, i.e to “box” and “sphere” as before. Texture, height, width, depth, color, and *y* coordinate property value of each view node are mapped to the corresponding scene node property. The time property is bound to a color property; values are mapped with the color binding (each version has a different color). The *x* and *z* coordinates are re-calculated by a checker layout algorithm. The *z* coordinates did not contain any model information. The *x* coordinates do not represent absolute or relative model values. Instead they correspond to a classification (system classes of different versions and same fully qualified name, e.g. java.io.File). Hence, reassigning *x* and *z* consistently does not destroy any model information.

Figure 6 (left) allows an *interpretation* of the system’s evolution. One can see that the system was several times heavily reengineered: almost no class (node) of a newer version has the same *x* and *z* coordinate as a corresponding class in a previous version. This is because the entire package structure changed.

Figure 6 (right) shows a similar evolutional visualization. The *model graph* and the *view mapping* are the same as above. The *model mapping* is different. The model node label property is mapped to the *x*-coordinate view graph property again. Values are mapped with a string filter binding removing package information from the labels, e.g. java.io.File becomes File. Then the classification binding is applied as before, i.e. same (now filtered) labels get same *x*-coordinates again.

With this change, *interpretation* changes as well. Despite restructuring the packages, classes with the same name in different versions are recognized as evolutions of another and represented in the same *x*-coordinate. We can see in the right part of Figure 6, e.g., spheres (interfaces), which have evolved in the system throughout all versions. These interfaces have been moved to other packages, which is why this evolution was not seen in Figure 6 (left).

6.4 Visualizing Program Quality

Figure 7 (left) shows four standard metrics that are maintainability indicators of a software system: Lack of Documentation (*LOD*), Weighted Method Count (*WMC*), Change Dependency Between Classes (*CDBC*) and Lines of Code (*LOC*). All metrics are applied to classes of a system. The *model graph* contains class nodes and properties corresponding to the metrics above. The property values

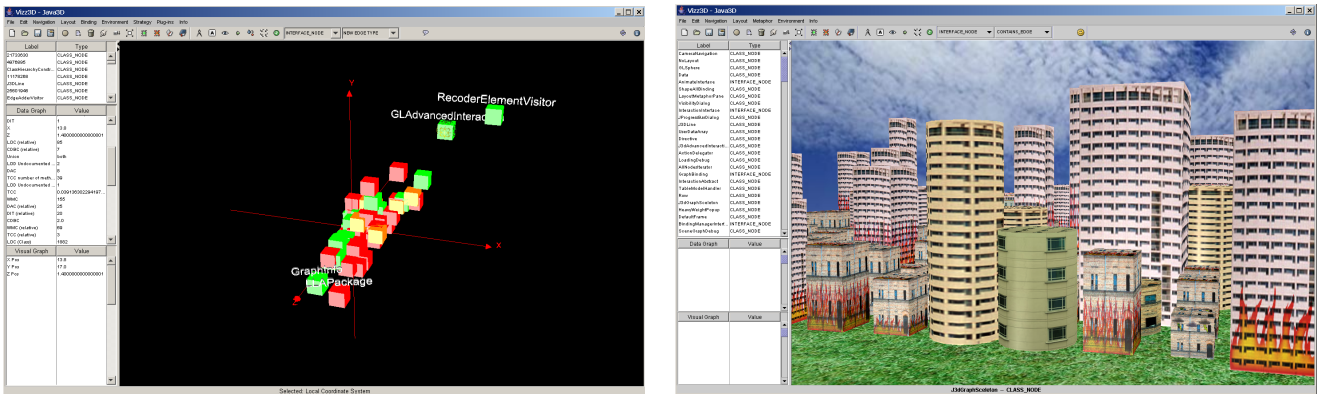

Figure 7: Program Quality with Different Bindings and Metaphors.

are scaled between 0 and 100. For details on the metric definitions and the analyses we refer to [Bär et al. 1999; Panas et al. 2005].

The *model mapping* is defined by a copy of the model to the view graph and a number of bindings. The first binding functions are instances of the linear binding. They map: *WMC* to *x*, *LOC* to *y*, *CDBC* to *z*. The last binding is an interval binding mapping *LOD* to colors: $0 \leq LOD \leq 50$ to "green", $50 < LOD \leq 80$ to "yellow" and $80 < LOD \leq 100$ to "red". Other view properties get default values.

The bindings of the *view mapping* are not new: the values of the node shape property are bound, as before, to visual objects. The remaining view properties are copied to the corresponding scene properties.

Figure 7 (left) may be *interpreted* as indicating maintainability outliers in the system. To the upper right, classes have many lines of code and a high complexity. To the front, classes have a high *CDBC*, which means if one of those classes is going to be changed, many other classes will need to be changed as well. The "healthy" classes are expected to be green and positioned in the origin of the coordinate system.

Figure 7 (left) may, however, not reveal any essential information for some users. The presented image, and hence the scene graph behind the image, is not appropriate in order to support the users cognition. Therefore, an individual scene graph would be appropriate. Metaphors found in nature or in the real world provide a visual language that the user already understands [J.F.Hopkins and P.A.Fishwick 2002; S.North 2000]. As a provider of a framework, it is not on us to discuss the appropriateness of abstract or natural metaphors. We just provide the opportunity to choose the one the user prefers.

This feature is demonstrated in Figure 7 (right). It shows the same *model* and uses the same *model mapping*. The *view mapping* is different. The most obvious is that the visual objects are chosen from another metaphor, a city metaphor. Furthermore, it ignores *x* and *z* coordinates, i.e. *WMC* and *LOD*. The *z*-coordinate of the view graph, i.e. the *LOC*, is bound to the houses height in the scene using a linear binding. The color property of the view graph, i.e. the *LOD*, is bound to the houses texture in the scene using a classification binding, i.e. "green" to no extra texture, "yellow" to a texture with some flames, "red" to a texture with flames all over. The *x*, *y* and *z* coordinates are calculated by an 2D orthogonal layout algorithm. The *interpretation* is now that high houses in flames are critical classes since they are large and poorly documented.

6.5 Animating Quality Evolution

The visualization of a system's quality can also be connected to the evolution of that system. In [Panas et al. 2005], we discuss the results of software quality measurements over different versions of the VizzAnalyzer framework, our own information extraction, analysis and focusing tool.

Figure 8 shows quality changes between different VizzAnalyzer versions. The *model graph* merges seven individual model graphs with class and interface nodes, each representing one version of the VizzAnalyzer. It contains a *Version* property for nodes encoding the version information, ΔLOC indicating the change of *LOC*, and ΔWMC indicating the change of *WMC* between the versions. Metrics are on a relative scale from -100 to $+100$.

The *model mapping* copies nodes of the model to the view graph and the version property to the one and only temporal property of the view model, i.e., we configure an animation instead of a static visualization. Further, the view mapping is defined by binding functions mapping node types to shapes, values of ΔLOC to node height (with linear binding), and sign of ΔLOC to the *y*-coordinate (with linear binding). The two bindings are chosen such that nodes with positive ΔLOC values start at $y = 0$ and nodes with negative ΔLOC values end at $y = 0$. Finally, sign of ΔWMC is bound to a color (using the interval binding with: negative to red, positive to green, and gray, otherwise). Other view properties get default values.

The *view mapping* maps nodes of different types to visual objects. It creates different scene graphs according to the temporal relations, i.e. nodes with a temporal value of "1" end up in the first scene, "2" in the second, etc. Texture, height, width, depth, and color properties of each view nodes are directly mapped to the corresponding scene node property. The *x* and *z* coordinates are calculated by a 2D checker layout algorithm that maps nodes with the same name to the same x, z-position.

The *animation* allows now to *interpret* different metrics over different versions. The layout algorithm architecture in Vizz3D supports a generic layout configuration, where any graph property can be assigned to and modified via a GUI dialog. For this example, a time slider was defined for the temporal property. This allows to show the animation in a pace that a user can understand. The height of boxes shows the change of *LOC*. A box in the positive/negative *y*-direction indicates an increase/decrease of LOC. The color changes between red and green, indicating that a class became more complex (red) or less complex (green). The default is a gray color for no changes. Important to note in this figure is that, due to the proper layout algorithm, the nodes representing the same class remain con-

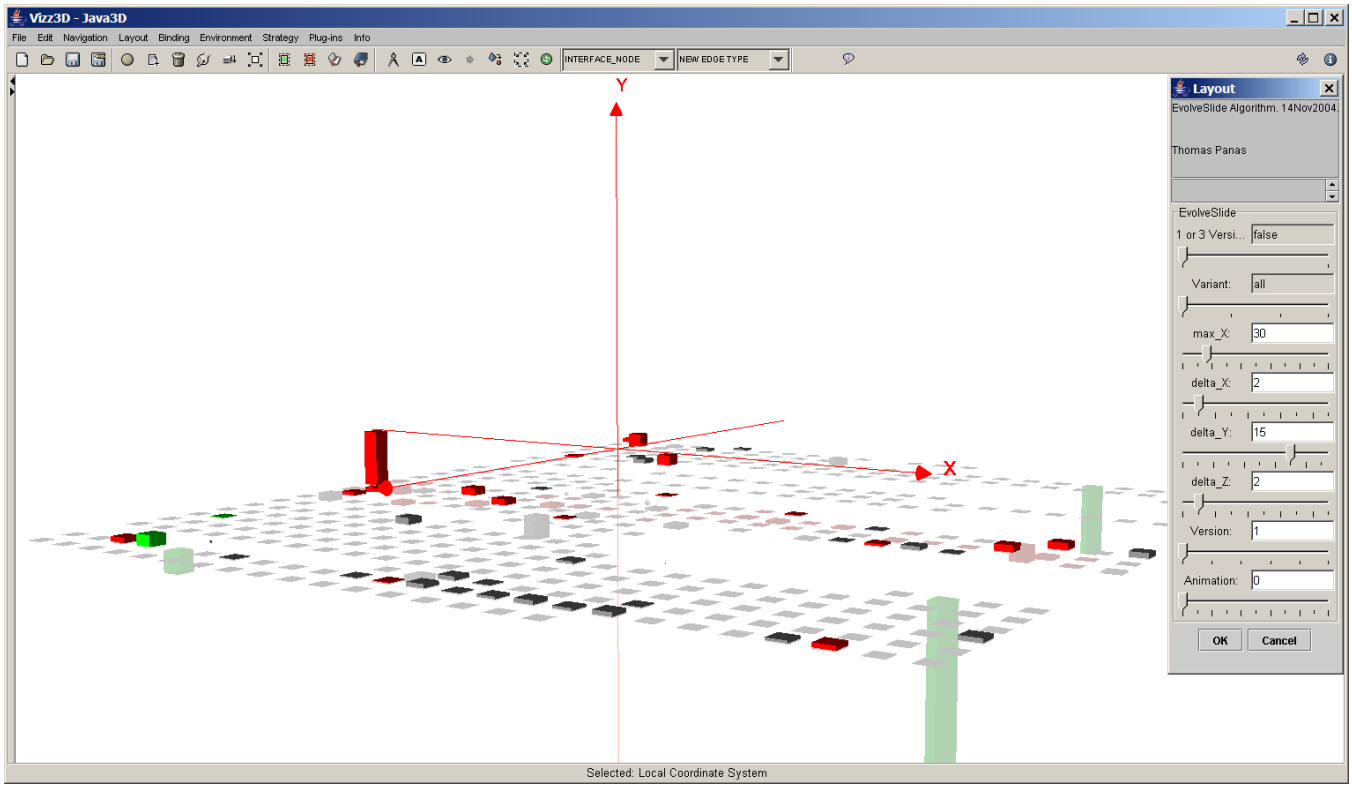

Figure 8: Snapshot of the Program Quality Animation

stantly at the same position throughout the versions, so that their change can easily be perceived.

7 Vizz3D

Vizz3D is a reusable framework implementing the concepts discussed before. All visualizations and animations shown in the previous section have been developed using this framework. More specifically, we iteratively added new binding functions, metaphors, and layout algorithms when needed for a specific visualization or animation. Then the visualizations or animations are just online-configured by defining model and view mappings, as described before.

The advantage of a framework is that users (developers) can reuse all binding functions, metaphors, and layout algorithms once they are defined. Hence, more and more, visualizations or animations can just be online-configured. This enables also an interactive and iterative software analysis, where appropriate views are created on demand.

Vizz3D, originally developed for the VizzAnalyzer information extraction, analysis and focusing engine [Panas et al. 2004], is now a stand-alone tool. It is implemented in Java and can be launched as a Java3D or OpenGL (gl4Java) application. Vizz3D comes with three variation points:

View Bindings map graph property values to other graph property values. In Vizz3D, bindings are deployed in a binding directory and loadable at run-time. In our implementation we use XML descriptors in order to online-configure bindings. Further, we allow binding functions to be implemented at run-time as well in case one is missing for a specific visualization. Note, that this online-implementation should be the exception once the framework is mature. Then, one just picks and configures the appropriate binding from the pool of existing.

Layout Algorithms assign position properties to scene graph nodes. Besides this layout function, a layout class may restrict the set of metaphors it is applicable for. The layout classes are deployed into the layout algorithm directory and dynamically loadable. This allows for an online-implementation of a layout, as well.

Metaphors are families of visual objects fitting together. Metaphors contain even descriptors of the environment of the visualization, i.e. the background, additional visual entities and other environmental factors like fog and light sources. The individual implementations are API dependent (Java3D or OpenGL). A metaphor class may restrict the set of applicable layouts. The metaphor classes are deployed into the metaphor directory and dynamically loadable.

Vizz3D bindings map property values from view graph to scene graph. The binding between model and view is task of the VizzAnalyzer tool. It mediates the view graph to the Vizz3D tool either in our internal graph structure or in an external specification, such as *GML* (Graph Modelling Language). Both mappings are defined and configured using an XML descriptor with the same Document Type Description (DTD). Only the properties involved are different. The following XML document is a (part of a) model binding configuration:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!DOCTYPE Bindings PUBLIC
4   "www.msi.vxu.se/Vizz3D"
5   "VAbinding.dtd">
6
7  <Bindings>
```

```
8    <!-- Binding of node view properties -->
9    <NodeBindings>
10     <Binding>
11       <Source>Type</Source>
12       <Target>Node Shape</Target>
13       <Function>
14         <IndexBinding/>
15       </Function>
16     </Binding>
17     <Binding>
18       <Source>LOD</Source>
19       <Target>Node Texture</Target>
20       <Function>
21         <IntervalBinding
22             Sources = "[ 50, 80 ]"
23             Targets = "[ 0, 1, 2 ]"/>
24       </Function>
25     </Binding>
26     ...
27   </NodeBindings>
28   <!-- Binding of edge view properties -->
29   <EdgeBindings>
30     ...
31  </EdgeBindings>
32 </Bindings>
```

The configuration consists of a number of node and edge binding functions (line 9ff. and 29ff., resp.) one for each visual property. The first node binding function (lines 10 to 16) defines the view graph property "Node Shape" from the model graph property "Type". Values are mapped with the index binding (line 14), i.e. different types of nodes get different shapes (each indicated with an integer). Since, the index binding does not need to be parameterized, the binding function is not further specified. This is in contrast to the second binding, mapping the *LOD* model property to the "Node Texture" view property by interval binding (lines 17 to 25). This interval binding needs to be configured by model property values $c_1 \dots c_n$ (line 22) and view property values $a_1 \dots a_{n+1}$ (line 22). In this case, *LOD* values smaller or equal 50 are mapped to texture 0, values in the interval $[50, 80]$ are mapped to texture 1, values larger 80 are mapped to texture 2. Remember, the view model is still abstract. Shapes and textures will be further mapped to actual visual objects due to view bindings.

The view binding configuration looks similar. However, it maps view properties to special scene properties. Therefore, it uses some special binding functions. Here an example:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!DOCTYPE Bindings PUBLIC
4   "www.msi.vxu.se/Vizz3D"
5   "VAbinding.dtd">
6
7  <Bindings>
8  <!-- Binding of node scene properties -->
9  <NodeBindings>
10   <Binding> <!--Shape of nodes-->
11     <Source>Node Shape</Source>
12     <Target>NODE SHAPE J3D</Target>
13     <Function>
14       <ShapeBinding Targets = "[ 'Box','Sphere']"/>
15     </Function>
16   </Binding>
17   <Binding> <!--LOD-->
18     <Source>Node Texture</Source>
19     <Target>NODE TEXTURE J3D</Target>
20     <Function>
21          <TextureBinding Targets="[ '../pen_green2.jpg',
22          '../pen_orange2.jpg','../pen_red2.jpg']" />
23     </Function>
24   </Binding>
25     ...
26   </Binding>
27 </NodeBindings>
28 <!-- Binding of edge scene properties -->
29 <EdgeBindings>
30     ...
31 </EdgeBindings>
32</Bindings>
```

Shape and texture binding functions (lines 14 and 19, resp.) are specific for the view binding. The principles remain the same. View properties (source) are mapped to scene properties (target) with a binding function that may be parameterized.

The advantage of such a binding is that it can be changed quickly without being hard-coded. Whenever the requirements for a visualization change, one has only to modify one XML file in contrast to hard-code a visualization.

Vizz3D implements a number of layout algorithms such as spring embedders, hierarchical layouts and checker algorithm [Lanza 2003]. The visualizations in Section 6 show examples. If they are not sufficient, the corresponding variation point needs to be extended. The layout to be applied in a certain visualization and its configuration is selected interactively in the Vizz3D GUI. However, settings can be stored in a configuration file and reloaded at once.

8 Related Work

Quite a few software visualization tools exist; we only compare to those that are configurable in our sense, like CodeCrawler [Lanza 2003], sv3d [Marcus et al. 2003], EVolve [Wang et al. 2003], CrocoCosmos [Lewerentz and Simon 2002] and GSee [Favre 2001].

Among them, only CodeCrawler distinguishes model, view, and scene (model-view-scene-controller paradigm), the others just distinguish model and scene (model-scene-controller paradigm, i.e. the commonly understood model-view-controller paradigm). We consider an abstract view important since it allows to reuse a mapping from an analysis domain to a visual domain. Different program analysis tools can be connected easily with different information visualization tools, without a user's special knowledge of the others domain. Despite this, a user must have basic knowledge about the data models of both domains in order to perform a mapping. However, the mapping is simplified through the abstract view, since it has a finite set of properties.

CodeCrawler allows to configure the mapping from model to view. The mapping from view to scene is hard-coded, but appears to be easily exchangeable (offline) for new information visualization back-ends. CodeCrawler supports only one metaphor (boxes and lines).

All, sv3d, EVolve, CrocoCosmos and GSee allow to online-configure the mapping between model and scene. To the best of our knowledge, none of them report that the configuration space (i.e. in our case the binding functions) can be extended online, as well. This restricts the configurable visualizations to those presumed by the designers. Especially, these frameworks come with a predefined number of metaphors, layouting algorithms, and binding functions (sometimes only one). Our online-extendable framework variation points add additional flexibility.

In the sv3d software visualization, the layout as well as the metaphor (extended 3D SeeSoft metaphor) are hard-coded.

EVolve is designed to visualize runtime behavior of Java programs. It is unclear if it is also applicable for structural and development process visualizations. Moreover, it involves the hard-coding of different types of visualizations within an implementation hierarchy. This approach is limited by the visual implementation language (here Java-Swing). It will be difficult to extend the framework towards other visualization back-ends, e.g a 3D visualization using OpenGL.

CrocoCosmos is another tool with just a model-scene architecture. However, this architecture is online-configurable with a very general layout algorithm: it considers a configurable subset of model properties and computes for any two nodes a distance by relating the number of property values they actually have in common and the number of property values they could have in common. These distances are approximated in 3D with a spring embedder.

GSee is a software exploration framework that can be configured to use third party analysis and visualization tools. GSee also maps model to scene, however, with little flexibility. A set of general parameterizable mapping functions lacks and hence the mappings are fixed. Exceptions are the Colorizer and EnumColorizer, were parametrization is allowed.

All tools above, including Vizz3D, are semi-automatic - user configurable tools/frameworks. Another set of program visualization tools are generative tools like BLOOM [Reiss and Renieris 2003], APT [Mackinlay 1986], BOZ [Casner 1991] and SAGE [Roth and Mattis 1991]. Those tools benefit if their internal algorithms can select mappings for best cognitive results. However, automation has the disadvantage that images, even cognitively well presented, may not reveal a problem under study. An interpretation of such an image may be difficult if a user's influence is restricted.

9 Conclusion

In this paper, we have argued for configuring visualizations on demand instead of programming them. This is enabled by an architecture consisting of a general data structure for the model and abstract view, and a concrete scene of a software visualization, as well as configurable functions for defining mappings: model-to-view and view-to-scene.

We implemented this architecture in a tool, Vizz3D, and showed the approach in a variety of examples. We assume that most visualizations can be crafted by changing the binding between model, view, and scene and an appropriate layout of scene objects. Developers can save time in creating an appropriate visualization just by configuring these parameters.

We designed Vizz3D as a framework with a number of variation points. Experts in the field of graph drawing, e.g., can easily integrate new layout algorithms, which can be used then in configurations. Similarly, the framework allows to define new binding functions and visual metaphors. We would like to invite experts to add approved layout, binding functions and metaphors.

Nevertheless, we do not claim that our architecture is covering the entire domain of software visualization. We have performed experiments within static program visualization, evolutional program visualization and algorithm animation. Experiments within dynamic program visualization, static data visualization or data animation are matter of future work. This might lead to a further generalization of our models, views, scenes, and configurations between them.

References

BÄR, H., BAUER, M., CIUPKE, O., DEMEYER, S., DUCASSE, S., LANZA, M., MARINESCU, R., NEBBE, R., NIERSTRASZ, O., RICHNER, T., RIEGER, M., RIVA, C., SASSEN, A. M., SCHULZ, B., STEYAERT, P., TICHELAAR, S., AND WEISBROD, J. 1999. The FAMOOS Object-Oriented Reengineering Handbook. Tech. rep., Forschungszentrum Informatik, Karlsruhe, Software Composition Group, University of Berne, ESPRIT Program Project 21975.

CASNER, S. M. 1991. Task-analytic Approach to the Automated Design of Graphic Presentations. *ACM Transactions on Graphics 10*, 2 (April), 111–151.

FAVRE, J. 2001. GSee: a Generic Software Exploration Environment. In *Proc. of Int. Workshop of Program Comprehension*.

J.F.HOPKINS, AND P.A.FISHWICK. 2002. The Rube Framework for Personalized 3D Software Visualization. In *Software Visualization. International Seminar. Revised Papers (Lecture Notes in Computer Science Vol.2269). Springer Verlag, pages 368-380. Berlin, Germany*.

LANZA, M. 2003. Object-Oriented Reverse Engineering. Coarse-grained, Fine-grained, and Evolutionary Software Visualization. In *Dissertation*.

LEWERENTZ, C., AND SIMON, F. 2002. Metrics-based 3D Visualization of Large Object-Oriented Programs. In *1st International Workshop on Visualizing Software for Understanding and Analysis*.

MACKINLAY, J. 1986. Automating the Design of Graphical Presentations of Relational Information. *ACM Transactions on Graphics 5*, 2 (April), 110–141.

MARCUS, A., FENG, L., AND MALETIC, J. I. 2003. 3D Representations for Software Visualization. In *Proceedings of ACM Symposium on Software Visualization*.

PANAS, T., LUNDBERG, J., AND LÖWE, W. 2004. Reuse in Reverse Engineering. In *12th International Workshop on Program Comprehension, Bari, Italy*.

PANAS, T., LINCKE, R., LUNDBERG, J., AND LÖWE, W. 2005. A Qualitative Evaluation of a Software Development and Re-Engineering Project. In *IEEE/NASA SEW-29*.

REISS, S. P., AND RENIERIS, M. 2003. The BLOOM Software Visualization System. In *Software Visualization – From Theory to Practice, MIT Press*.

ROTH, S. F., AND MATTIS, J. 1991. Automating the Presentation of Information. In *Proc. of the IEEE Conf. on Artificial Intelligence Applications*.

S.NORTH. 2000. Procession: Using Intelligent 3D Information Visualization to Support Client Understanding during Construction Projects. In *Proceedings of Spie - the International Society for Optical Engineering, vol. 3960, p. 356-64. USA*.

WANG, Q., WANG, W., BROWN, R., DRIESEN, K., DUFOUR, B., HENDREN, L., AND VERBRUGGE, C. 2003. EVolve: An Open Extensible Software Visualization Framework. In *Proceedings of ACM Symposium on Software Visualization*.