International Journal of Software Engineering and Knowledge Engineering

Rapid Construction of Software Comprehension Tools

WELF LÖWE
Software Technology Group, MSI, Växjö University
35195 Växjö, Sweden
Welf.Lowe@msi.vxu.se
http://www.msi.vxu.se/~ wlo

THOMAS PANAS
Software Technology Group, MSI, Växjö University
35195 Växjö, Sweden
Thomas.Panas@msi.vxu.se

A software comprehension tool defines an abstract software model, views on this model, analyses creating the model, and a mapping between model and view. For creating new comprehension tools, we configure online analyses, models, 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.

Keywords: Software Analysis; Software Visualization; VizzAnalyzer/Vizz3D.

1. Introduction

Software maintenance includes tasks like finding and removing design flaws and adding new functionality to a software system. These maintenance tasks are expensive today. Cost estimations of resources and time range from 50% to 80% of the total costs of ownership of a software system [3]. In order to maintain a system, it needs to be comprehended and the effort for gaining comprehension even dominates the total maintenance effort. Estimations range from 40% up to 90% [9,27,25,11].

Design documents and other kinds of documentation providing an abstract and comprehensible view on a system are often outdated or not available at all. In many cases, the code of the system is the only trustworthy source of information. But, due to the size of many systems, gaining comprehension from code needs to be supported by tools. The task of these tools is to extract information, analyze and focus it, and, finally, visualize it for the person who tries to understand the software system.

Depending on the maintenance goal, the abstraction level, and the humans involved, different views of a software system are appropriate. This implies that different visualizations need to be designed, different analyses to be defined, and different kinds of information to be extracted. The large number of possible variations makes

it impossible to predefine a number of appropriate combinations of information extractions, analyses and visualizations. Instead, the suitable combination tailored to fit a certain software comprehension task has to be *developed* on demand. Hence, it makes sense to spend some time on improving the development of software comprehension tools.

Separation of concerns and the reuse of predefined components implementing these concerns increase the efficiency of software development in general. Two concerns for software comprehension tools are obviously the abstract *model* of the software to be understood and the *view* on this model. The former needs to capture the information required for the software comprehension task. The latter needs to create the image of that information for the humans involved. Separating model and view allows reusing and recombining models (and analyses creating them) and views thereof. To create a new software comprehension tool, *mappings* between software code and model, and between model and view must be defined. The former mapping is known as information extraction, analysis and focusing, the latter as software visualization. Due to the separation of model and view, the domains of software analyses and information visualization get separated. Hence, analysis tools can be reused with different visualization back-ends and visualization tools can be reused with different analysis front-ends.

Within the domain of software analyses we distinguish a *base model* of a software that contains information directly extracted from the software (or other source documents like documentation, design specification, execution profiles etc.). This base model is the information source for further analyses and focusing engines manipulating information on the model. This distinction makes sense since it again increases the reusability: a information extraction front-end (specific to the source document type (e.g. a programming or design specification language) can be recombined with general analyses and focussing engines and vice versa.

Within the domain of information visualization, the same information can be illustrated using different images. 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.

Figure 1 summarizes concerns in software comprehension tools as well as the data-structures involved like source documents, (base) models, views, and scenes. Note that it is a design decision to separate the different intermediate data structures in order to reuse certain components and, thereby, increase the efficiency of creating a specific software comprehension tool. In each such tool, we assume that

- *source* code is mapped to a number of *base models* capturing in some way the information to comprehend (information extraction),

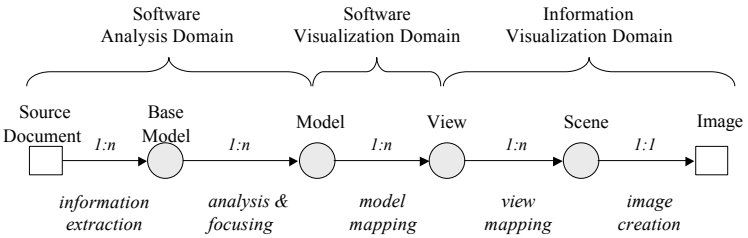

Fig. 1. Data structures and Mappings in Software Comprehension Tools.

- a *base model* is mapped (by different analysis, abstraction, merge steps) to a number of software *models* directly encoding the information to comprehend,
- a *model* is mapped to possibly different abstract *views* (model mapping),
- each *view* corresponds to different *scenes*, each being a concrete view in terms of the data structure of a particular visualization tool (view mapping), and
- each *scene* 1 : 1-matches to an *image* visible for the user.

This separation induces an architecture allowing to online *configure* the mappings between model, view and scene instead of programming them. Altogether, it enables to interactively develop software comprehension tools tailored to a particular comprehension context. This is the actual contribution of this paper. We have implemented this architecture in two frameworks called *VizzAnalyzer* and *Vizz3D*, serving as a proof of concept.

Before discussing a configurable mapping between a model for software comprehension and the view on that model, we need to study the corresponding data structures, (base) model and view, which we do in Section 2 and Section 3, respectively. In Section 4, we discuss how to configure the extraction of the base model from source documents. Then, in Section 5, we introduce analyses and focusing to get the actual software model. In Section 6, we define the configurable mapping between model and view. A view is further mapped to a scene, as discussed in Section 7. Examples of rapid construction of different comprehension tools are presented in Section 8. In Sections 9 and 10, we describe our *VizzAnalyzer* and *Vizz3D* frameworks. Section 11 discusses related work and Section 12 concludes the paper.

2. Models for Software Comprehension

As discussed before, depending on the software comprehension goal, different models of the software to understand are appropriate. Since these goals may be quite diverse, the data types capturing the required models appear rather diverse, too. A model for understanding the static software architecture using clustering, e.g., appears quite different from a model for understanding a sorting algorithm using

animation.

For creating a configurable online mapping between source document and model, and 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 accepting that matters of software visualization are: (i) program structures and their changes over a number of development cycles, and (ii) program behavior.

Program structures (i), in general, can be captured by programming language entities and relations between them. Their changes over a number of development cycles add just new entities and relations to the picture, e.g. "developer" entities or "evolves-to" and "developed-by" relations. 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.

Obviously, entities of different type and relations (between them) of different type capture the models for software comprehension; if not all then at least a relevant subset thereof. In our architecture, we therefore assume a general *graph structure* capturing the models. 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" relations for nodes and edges,
- arbitrary properties of the whole graph and arbitrary properties for its nodes and edges modelling relations between entities and values, e.g. metric values.

It is the responsibility of a developer of a comprehension tool to define an instance of this graph structure, i.e. to define concrete node and edge types and concrete properties. This model needs to be justified with respect to the software comprehension goal. From the viewpoint of a comprehension 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 Comprehension

Since software comprehension 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 could 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. Creating a Base Model

The base model is created by a compiler front-end-like tool. Like a compiler, it understands the language of the source code, constructs an *abstract syntax tree* (AST), and annotates it with *semantic information*.

In terms of our model data structure, an AST is a tree, i.e. a special graph structure. Its node types correspond to elements of the source language, the edge type to the structural contains relation of these source language elements in programs, e.g. a program (node) contains a set of classes (class nodes), each in turn containing a set of methods (method nodes) etc.

Like compiler front-ends, our information extractors add some intrinsic attributes to the AST structure, e.g. position of the source language elements and name information, if applicable. There are also some differences between ordinary compiler front-ends and the information extractors that we assume. On one hand, a compiler front-end may throw away information not contributing to the semantics of the source, e.g. comments. Since it might support software comprehension, an information extractor should be able to preserve any kind of source information and represent it accordingly. On the other hand, depending on the particular comprehension task, information extractors may throw away information compiler front-ends need to preserve. For example, if the inheritance structure of a system is to understand, all AST subtrees representing method implementations can be ignored. This reduces both memory consumption and time for further analysis and focusing steps.

Semantic analysis adds further relations to the AST structure. For example, it may add relations between used and defining occurrences of types and methods. All this semantic information is captured by different types of edges (each type corresponding to a relation) between AST nodes. Again, in contrast to compiler

front-ends, and depending on the comprehension task, not all relations analyzed need to be captured in the base model. For example, if the inheritance structure of a system is to understand, used and defining occurrences of types are relevant, used and defining occurrences of methods are not.

For the rapid creation of a comprehension tool suitable for a certain comprehension task, we need to configure a base model and the corresponding information extraction. Therefore, we assume a complete front-end building a complete AST with all semantic relations attached to it. Any specific base model can now be understood as a filtering of this complete graph.

Filtering is specified by defining relevant node and edge types. If a node type is not relevant, all nodes of that type are filtered from the base model. The subtree of an irrelevant node n might contain relevant nodes. These subtrees are propagated to the next relevant parent of node n. More precisely, let $c(n)$ be a relevant (transitive) child and $p(n)$ a relevant (transitive) parent of an irrelevant node n in an AST. If there are no other relevant nodes on the path $p(n)$ to n and n to $c(n)$ then there exists a structural contains edge from $p(n)$ to $c(n)$ in the base model.

If a semantic relation is irrelevant, corresponding edges are not inserted in the base model. If a relevant semantic relation is defined over irrelevant node types, the corresponding edge is propagated to the next relevant parent of the corresponding irrelevant node. More precisely, let $p(n)$ and $p(m)$ a relevant (transitive) parents of irrelevant nodes n and m, respectively, in an AST and $e(n, m)$ a relevant semantic relation. If there are no other relevant nodes on the path $p(n)$ to n and $p(m)$ to m then there exists an edge $e(p(n), p(m))$ in the base model. Nodes n and m, and edge $e(n, m)$ are not part of the base model.

A base model and its information extraction can be configured (and hence rapidly constructed) by defining relevant entity- and relation-types of the source document language. Note, that construction and filtering can be interleaved such that irrelevant nodes and edges are not explicitly constructed.

5. Analyses to Create a Model for Software Comprehension

Usually, a base model is not directly useful for program comprehension. In general, further analyses and abstractions are required in order to support comprehension directly. They incrementally compute aggregated information and filter intermediate information irrelevant for the actual comprehension task.

Each analysis takes one or more model graph(s) and delivers either a new graph or attaches information, i.e. new nodes, edges or properties, to the input graph(s). The initial analyses take the base model graph(s) as input.

Analyses may have preconditions on:

- the node and edge types of the input graphs,
- the properties of nodes and edges and their values,
- general structural graph properties (e.g. acyclic, directed etc.),

and the graph properties are used as a simple, dynamic type system for modelling and checking such preconditions. Each analysis may use the graph properties to check individual preconditions.

For example, a model capturing a call graph might have properties with predicates like `directed="true"`, `connected="true"`, `nodetypes={"method definition","method call"}`, `edgetype={"call reference"}` to indicate some general graph properties. A class graph node might be tagged with `type="class"`, `size=221` to indicate its type and a size metric. Given this kind of information, further analyses might be applicable or not. An analysis is applicable to a graph if (and only if) the graph has properties satisfying the pre-conditions of that analysis.

With the above information, we can define an abstract interface for analyses. Let M be the model graph structure. Then each analysis is a mapping $a : M^n \rightarrow M$. Furthermore, each analysis defines a boolean precondition checker $pre_a : M^n \rightarrow \mathbf{B}$. In our design, we restrict this general precondition checker to n individual checkers, each checking the appropriateness of a certain input model graph. The precondition checker for the k-th input model graph is a boolean function: $pre_a^k : M \rightarrow \mathbf{B}$. A concrete analysis class inherits from this general abstract interface and provides an implementation for the function a and the precondition checkers $pre_a^1 \ldots pre_a^n$ (provided the analysis a implemented an n-ary function). Given a framework capturing a number of model graphs, we can now dynamically invoke analyses. Therefore:

(1) the user interactively selects an analysis a,
(2) the user interactively selects n input model graphs $m_1, \ldots m_n$ (provided a implements an n-ary function),
(3) the framework checks the conjunction of the preconditions $pre_a^1(m_1) \wedge \ldots \wedge pre_a^n(m_n)$,
(4) if true, it performs analysis a, and captures the resulting graph.

Rapid creation of software comprehension tools depends on predefined analyses. However, new analyses just require a new implementation of the abstract analysis interface. Our *VizzAnalyzer* implementation, cf. Section 9 implements quite a number of metric and structural analyses, see Section 8 for examples. Additionally, its dynamic plug-in interface allows to integrate new analyses online.

6. Mapping between Model and View

The model mapping establishes a relation between model graphs and view graphs, cf. Figure 1. In order to display a view graph, all view graph properties need to be defined. Hence, the mapping needs to be surjective in the set of view graph properties. On the other hand, not all properties of the model need to be depicted in an 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 constructers, 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 is a function with model graphs as domain and view graphs as co-domain. In general, this 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 or ignored, otherwise.
- 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 or ignored.
- Edge properties of the model graph are mapped to edge properties of the view graph or ignored.

Such a restricted mapping is configurable in two steps. First, the view graph structure 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.

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 allowing for online adaptation. The actual property mapping is configured by assigning to each view graph property a model property and a suitable function (including their parameters). So far, we defined the following parameterized binding functions:

Linear binding: linearly maps numerical model properties 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 arbitrarily typed view properties. It is parameterized by a switching value c and two view property values a_1 and a_2 of the view type. 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 and generalizes the switch binding. It is parameterized by n switching values $c_1 \dots c_n$ and $n+1$ view property values $a_1 \dots a_{n+1}$ of the view type. 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 type 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 there exist more then n different model property values, 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 case 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 is replaced by s in the view property string. The remainder is copied. Since labels of nodes and edges are the only string valued view properties, this mapping is used to create view 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 framework if required, cf. Sections 9 and 10. 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.

7. 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 visualization tool specific objects and properties. We refer to these tool specific data structures as *scene graphs*. For example, in *OpenGL* and *Java3D*, etc. nodes and edges have their specific

representations. These visualizations tools can view their respective scene graphs directly as images.

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

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

Different abstract node and edge shapes, textures and colors, must be mapped to a visual representation in the concrete scene. 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 boxes and spheres 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 technically not a problem, to connect boxes and spheres with streets.

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. This is rather an encoding than an arbitrary mapping and could be predefined for each visualization tool.

However, if the internal scene data structure is accessible, as e.g. in our *Vizz3D* framework, cf. Section 10, the same binding functions as defined in Section 6 are applicable to properties in the scene graph. Then the user can even re-bind the view properties online, e.g. re-bind the property height to depth if that leveraged better comprehension.

As discussed before, certain properties of a view graph might not have been defined by model properties but by default values. Such an 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 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.

Metaphor and layout are not orthogonal dimensions. Instead a metaphor may restrict the applicable layouts and vice versa. For example, a metaphor with house and street objects requires specific 2D layouts; a 3D spring embedder would not be appropriate.

As many other visualization tools, the *Vizz3D* framework, cf. Section 10, predefines 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).

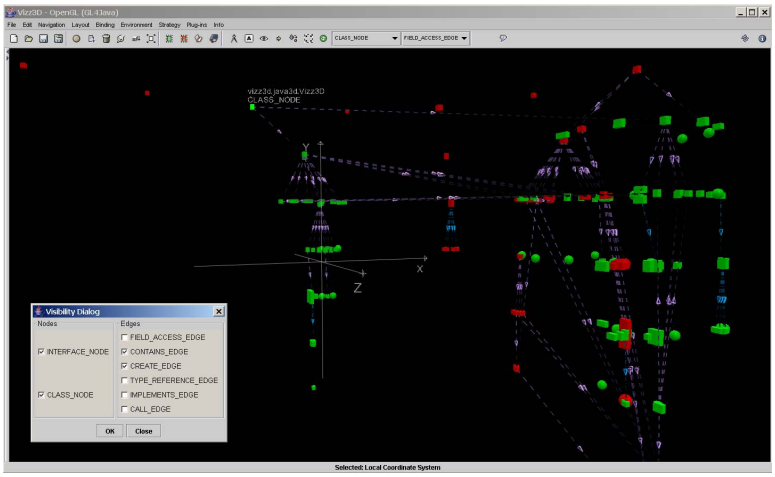

Fig. 2. Hierarchy and Class Usage Graphs merged.

8. Examples of Software Comprehension Tools

This sections defines a number of software comprehension tools in terms of their *model* and *view* including *base model*, analyses, *model mapping* and *view mapping*. Each example concludes with a brief *interpretation* of the picture in the model domain. All examples are configured with *VizzAnalyzer*/*Vizz3D* framework, which is discussed in more detail then in Sections 9 and 10.

8.1. *Comprehending Class Interaction*

Figure 2 shows a software visualization of *Vizz3D* itself. The information extraction was configured to deliver a class hierarchy graph and a class usage graph. Both consisting of class and interface nodes, and structural contains edges. The hierarchy graph has additionally extends and implements edges; the class usage graph has field access, creates, call reference and type reference edges. Those two graphs were merged to a *hierarchy-usage model graph*. Hence, the model graph contains nodes with properties `type="class"` and `type="interface"` and edges with properties `type="structural contains"`, `type="extends"`, `type="implements"`, `type="field access"`, `type="creates"`, `type="call reference"`, and `type="type reference"`. Additional node properties are `label`, i.e. fully qualified class/interface name, and, as result of a dead code analysis, a boolean property `dead code` indicating classes and interfaces never used.

The *model mapping* completely copies the model to a view graph structure. Then it is defined by the following bindings, cf. Figure 3 (left): 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 a view node property `shape` capturing integers. Property values

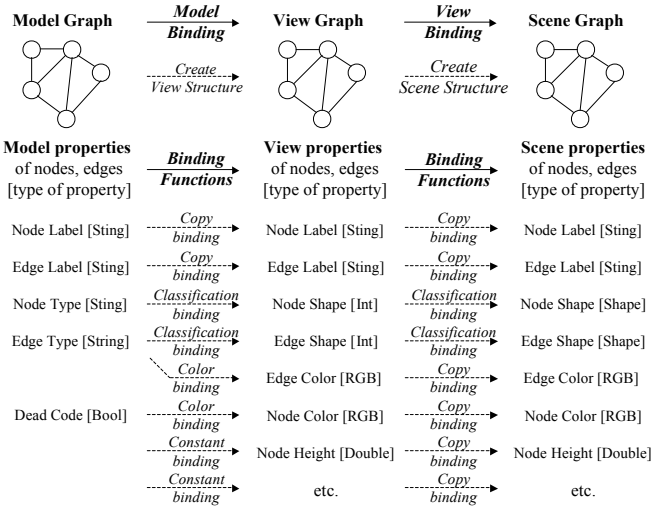

Fig. 3. Example Model and View Mapping.

`type="class"` are mapped to `shape="1"`, `type="interface"` to `shape="2"` (classification binding). The edge `type` property of the model graph is bound to both, edge `shape` and `color`. Values are mapped with classification/color binding, e.g. model edges of `type="field accesses"` to view edges of `shape="1"` and `color="blue"`. The `dead code` property of model nodes is bound to the `color` property of view nodes: `dead code="true"` to view node `color="red"` and `dead code="false"` to view node `color="green"` (color binding). The other view properties, i.e. `texture`, `hight`, `width`, `depth`, and positions of view nodes and edges, get a default value (constant binding).

Note, in the binding, we need to distinguish the *properties* mapped to another from the actual binding function implementing a mapping of model to view property *values*. For instance, the `dead code` *property* of classes (model) is mapped to the node `color` property (view), where the `dead code` property *value* `"true"` is mapped to a `color` property value `"red"` by a color binding function. This distinction continues to hold for the bindings in the view mapping, as well.

In the *view mapping*, different node shapes are bound to different visual objects in the scene taken from a particular metaphor, i.e. view node `shape="1"` to scene object type `shape J3D="box"`, view node `shape="2"` to scene object type `shape J3D="sphere"`. The edge `shape` property is bound to the scene object types, as well; its values are mapped to visual objects chosen from the same metaphor, e.g. view edge `shape="1"` to scene object type `shape J3D="solid line"`. The following bindings apply to the other view properties, cf. Figure 3 (right): labels of view nodes and edges are copied to labels of scene nodes and edges (copy binding). Node and edge colors are just directly mapped to the corresponding scene node and edge colors

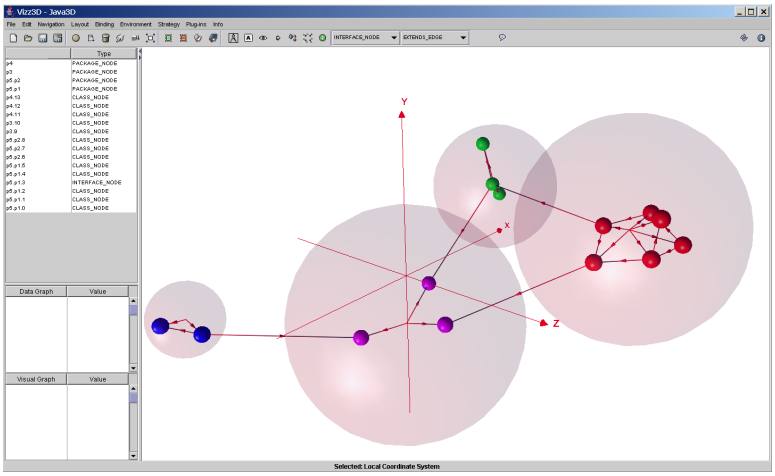

Fig. 4. Package Graph Visualization.

(copy binding). 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 *Vizz3D* classes (boxes) and interfaces (spheres). The red color indicates classes/iterfaces that are not used (dead code)[a]. Certain edge types are hidden online in the screen-shot in order to limit the information displayed at once.

8.2. *Comprehending Package Interaction*

Figure 4 is constructed from program analysis creating a call graph. From this call graph, a package graph is created as a result of an additional analysis. Finally, the call and package graph are merged to represent 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` properties are bound to node and edge `shape` properties; values are bound by the classification binding, e.g. `type="class"` are mapped to `shape="1"`, `type="package"` to `shape="2"`. The values of the `package` property are bound to the node `color` property; values are mapped with the color

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

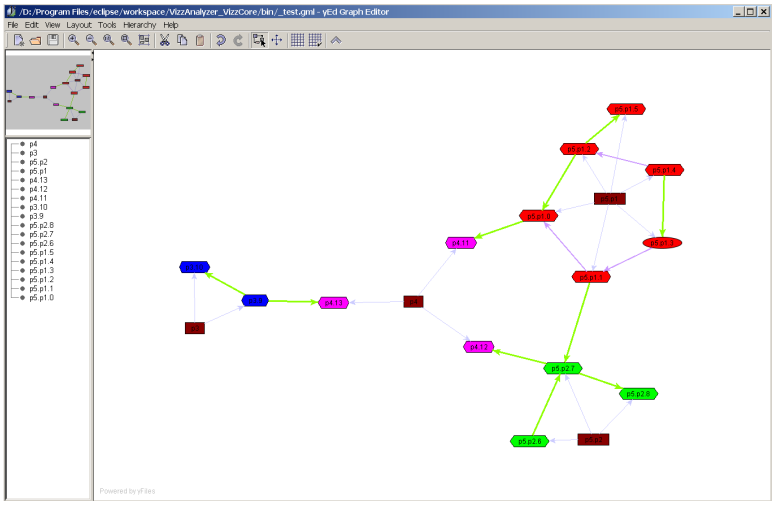

Fig. 5. Package Graph Visualization with yEd.

binding, i.e. classes with the same `package` property value get the same `color` value. The other view properties, i.e. `texture`, `hight`, `width`, `depth`, and positions get a default value (constant binding).

The *view mapping* is defined by creating the scene graph structure from the view graph. The values of the node `shape` property are bound to visual object types, i.e. `shape="1"` to scene object type `shape J3D="solid sphere"`, `shape="2"` to `shape J3D="transparent sphere"`. The values of the edge `shape` property are all bound to visual object values `shape J3D="solid line"`. The `x`, `y` and `z` properties are calculated by an automatic layout algorithm. The `hight`, `width`, `depth` properties of the scene nodes of `shape J3D="solid sphere"` (derived from model nodes of `type="class"` over view nodes of `shape="1"`) is constant.

The `hight`, `width`, `depth` properties of scene nodes of `shape J3D="transparent sphere"` (derived from model nodes of `type="package"` over view nodes of `shape="2"`) is defined by the layout algorithm. Note that the recalculation of `x`, `y`, `z`, and size properties 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 4 can now be *interpreted* as the interaction of classes – solid spheres – belonging to packages – transparent spheres enclosing classes of the same package. Node color encodes the classes' package, as well.

By keeping model and model mapping, but exchanging the view mapping, a different image is produced, cf. Figure 5. The information visualization tool applied is yEd [32]. The view mapping maps abstract node shapes to different 2D visual objects and colors, and abstract edge shapes to lines of different color. While the

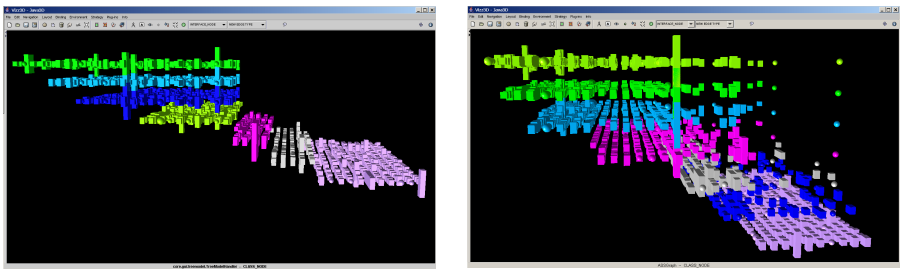

Fig. 6. Program Evolution. Different versions of the VizzAnalyzer.

metaphor is predefined, different layout algorithms can be applied; we choose a spring embedding algorithm.

8.3. *Comprehending Program Evolution*

Figures 6 illustrates the evolution of a system over seven versions. The *model graph* is a merger of seven individual model graphs, each representing one version. For each version, the base model contains classes and interfaces. A metric analysis added size information indicating the number of lines of code (*LOC*) for each class/interface. The final model contains nodes representing classes and interfaces. It does not contain any edges. The node properties are `label` and `type` (as in the previously discussed models), `LOC` and `Version`, indicating which version a node belongs to.

The *model mapping* is defined by a complete copy of the model to the view structure and the following bindings: Labels and types are mapped as before. The `LOC` property is bound to the `height` property; values are mapped using a linear binding. The `version` model property is bound to the `color` view property; values are mapped with the color binding (each version has a different color). 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 the 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 boxes and spheres as before. The `texture`, `width`, `depth`, and `y`-coordinate properties of view nodes are mapped to the corresponding scene node properties. The `x`- and `z`-coordinates are re-calculated by a checker layout algorithm. The `z`-coordinates don't contain any model information and, hence, it is ok to assign new values. The `x`-coordinates don't 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.

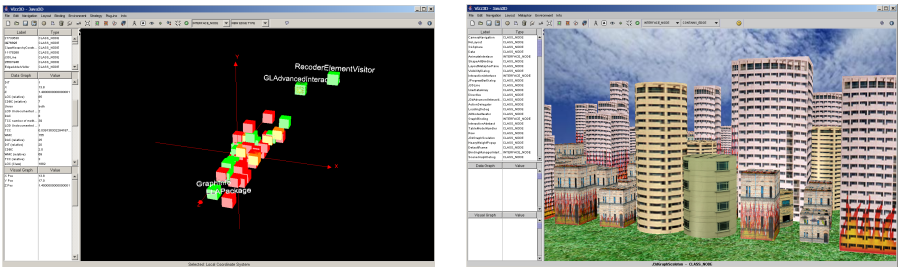

Fig. 7. Program Quality with Different Bindings and Metaphors.

Figure 6 (left) allows an *interpretation* of the system's evolution. One can see that the system was heavily re-engineered several times: 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 nodes' `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. node with the same (now filtered) labels get the same `x`-coordinates again.

With this change, the *interpretation* changes as well. Despite restructuring of packages, classes with the same name in different versions are recognized as evolutions of another and represented at the same `x`- and `z`-coordinates. 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).

8.4. *Comprehending Program Quality*

Figure 7 (left) shows four standard metrics that are indicators for maintainability of a software system: Lack of Documentation (*LOD*), Weighted Method Count (*WMC*), Change Dependency Between Classes (*CDBC*) and Lines of Code (*LOC*). The initial base model graph structures also contains attribute and method definitions as well as method calls and attribute references to compute the above metrics. In the final model, this intermediate information is filtered; it only contains class nodes and properties corresponding to the metrics above. The property values are scaled between 0 and 100. For details on the metric definitions and the analyses we refer to [2,28].

The *model mapping* is defined by a copying model to 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 final binding is an interval binding mapping `LOD`

to `color` values: $0 \leq value_{LOD} \leq 50$ to `color="green"`, $50 < value_{LOD} \leq 80$ to `color="yellow"` and $80 < value_{LOD} \leq 100$ to `color="red"`. Other view properties get default values.

The bindings of the *view mapping* is not innovative: the values of the nodes' `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 changes, many others need to be changed, as well. The "healthy" classes are expected to be green and positioned in the origin of the coordinate system.

For some users, Figure 7 (left) may not easily reveal information. The presented image, and hence the scene graph behind the image, may be inappropriate in order to support the users' cognition. Therefore, the scene graph can be adjusted individually, e.g. by taking metaphors found in nature or in the real world providing a visual language that the users are familiar with [15,26]. As providers of a framework, it is not on us to discuss the appropriateness of abstract or natural metaphors. We just provide the opportunity to chose the one the users prefer.

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 of the view graph, i.e. `WMC` and `LOD` of the model graph. The `z`-coordinate of the view graph, i.e. the `LOC` property in the model, is bound to the houses `height` in the scene using a linear binding. The `color` property of the view graph, i.e. the `LOD` in the model, is bound to an additional `texture` of houses in the scene using a classification binding, i.e. `color="green"` to `texture="no"`, `color="yellow"` to `texture="low flames"`, `color="red"` to `texture="high flames"`. The `x`, `y` and `z` coordinates are calculated by a 2D orthogonal layout algorithm. The *interpretation* is now that high houses in flames are critical classes since they are large and poorly documented.

8.5. *Animating Quality Evolution*

The visualization of a system's quality can also be connected to the evolution of that system. In [28], we discuss the results of software quality measurements over different versions of the *VizzAnalyzer* framework, our own information extraction, analysis and focusing tool.

Visualizations in Figure 8 show quality changes between different *VizzAnalyzer* versions. Initially, the program retrieval engine supplies the *VizzAnalyzer* with different versions of itself. The analysis engine merges these seven individual graphs to one new *model graph*, containing class and interface nodes. In addition, it contains edges of a `type="evolves to"` encoding the version information, and properties encoding ΔLOC, the change of LOC, and ΔWMC, the change of WMC between

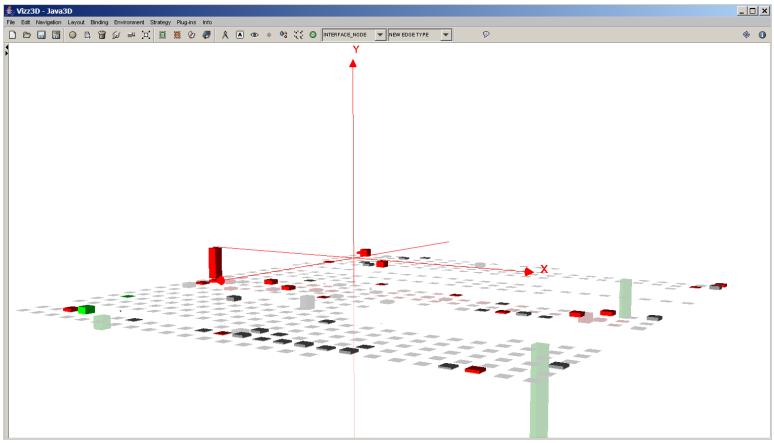

Fig. 8. Snapshot of the Program Quality Animation

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 edges of the evolves-to relation to the one and only temporal relation of the view model, i.e., we configure an animation instead of a static visualization. Furthermore, 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 grey, 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 edges: nodes with no incoming temporal edges end up in the first scene, a node of depth n in the view graph (only considering temporal edges) ends up in the n-th scene. The `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 metric over different versions. 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 grey 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 constantly at the same position throughout the versions, so that their change can easily be perceived.

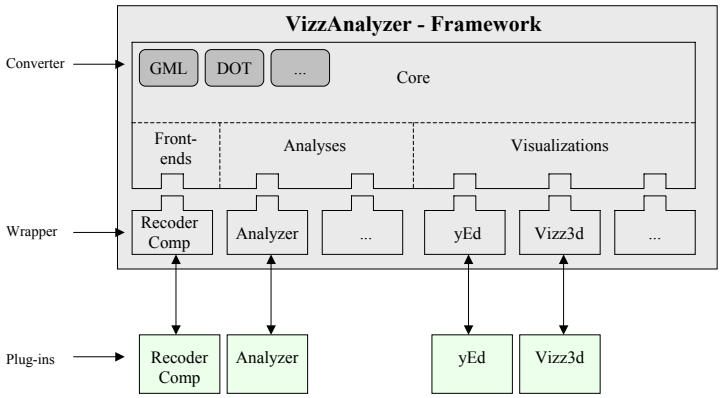

Fig. 9. Our Framework with external components

9. VizzAnalyzer

The VizzAnalyzer [22,29] is a reusable framework, implementing the concepts discussed above. All model and view graphs discussed before have been created using this framework. We iteratively added new binding functions on demand, usually whenever we added a new analysis or visualization tool. Then the view graphs are just online configured by defining the model mapping, as described before.

Currently, our instantiation of the framework uses one information extraction, two analysis, and two visualizations components. Each component is in itself non-trivial and could be understood as a stand-alone tool. The components are interacting via a general graph data structure. Our graph implementation additionally contains converters to the standard graph formats *GML* [12] and *DOT* [10] to simplify the adaptation process required for plug-ins. The framework instantiation is depicted in Figure 9.

The (currently) only component for program retrieval is our *RecoderComp* package. It is based on the *Recoder API* [19]. The *RecoderComp* package is feeded with specific project information about a system to be parsed. This information can be created, stored, loaded and configured online. It looks as follows:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE project SYSTEM "projects.dtd">
3
4 <project>
5   <projectname>Vizz3D_Java3D</projectname>
6   <projectdescription>Vizz3D Java3D version</projectdescription>
7   <projectspecification>
8      <entrypoints>
9     <mainclass>vizz3d.java3d.Vizz3D</mainclass>
10   </entrypoints>
```

```
11   <paths>
12    <sourcepath>$VA_HOME\..\VizzAnalyzer_Vizz3D\src</sourcepath>
13    <sourcepath>$VA_HOME\..\VizzAnalyzer_VAutils\src</sourcepath>
14    <sourcepath>$VA_HOME\..\VizzAnalyzer_visgraph\src</sourcepath>
15    <binarypath>$JAVA_HOME\lib\rt.jar</binarypath>
16    <binarypath>$VA_HOME\..\components\gl4java.jar</binarypath>
17    <binarypath>$VA_HOME\..\components\j3dcore.jar</binarypath>
18    <binarypath>$VA_HOME\..\components\j3dutils.jar</binarypath>
19    <binarypath>$VA_HOME\..\components\j3dtree.jar</binarypath>
20    <binarypath>$VA_HOME\..\components\vecmath.jar</binarypath>
21    <binarypath>$VA_HOME\..\components\grail.jar</binarypath>
22    <binarypath>$VA_HOME\..\components\StarfireExt.jar</binarypath>
23    <binarypath>$VA_HOME\..\components\xmleditor.jar</binarypath>
24   </paths>
25   <filter>
26    <exclude>visgraph</exclude>
27    <exclude>VAutils</exclude>
28   </filter>
29  </projectspecification>
30 </project>
```

Note that this specification is not part of the *VizzAnalyzer* framework, but part of one of its plug-ins (*RecoderComp*). The specification above provides the *RecoderComp* engine with information about the location of source classes (lines 12-14) and binary classes (15-23). A filter includes information about specific top-level packages that should be excluded from information extraction (lines 25-28). Finally, the entry point(s) for the *Recoder* parser are specified in the lines 8-10.

RecoderComp produces the base model discussed before. It reads in a specification of relevant node and edge types corresponding to AST node types and static semantic relations analyzed by the *Recoder* front-end. While parsing the specified program, *RecoderComp* casts events for each node and edge of a relevant type. Those events are caught and used to build the specific base model graphs. This architecture allows us to specify any number of base model graph structures.

RecoderComp uses an XML specification for defining relevant node and edge types, i.e. ode and edge types of a base model. For instance, to advice the extraction of a class hierarchy and a call graph, the following specification is used:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE graphs SYSTEM "graphs.dtd">
3
4 <graphs>
5    <graph>
6      <name>My Class Hierarchy Graph</name>
7      <type>Class Hierarchy Graph</type>
8      <description>Register inheritance relations
9                   between classes</description>
10      <edges>
11        <edge>Extends</edge>
12        <edge>Implements</edge>
13      </edges>
14      <nodes>
15        <node>ClassDeclaration</node>
16        <node>InterfaceDeclaration</node>
17      </nodes>
18    </graph>
19    <graph>
20      <name>My Call Graph</name>
21      <type>Call Graph</type>
22      <description>Register calls and field accesses
23                   between members</description>
```

```
24      <edges>
25        <edge>MethodReference</edge>
26        <edge>New</edge>
27        <edge>FieldReference</edge>
28      </edges>
29      <nodes>
30        <node>MethodDeclaration</node>
31        <node>ConstructorDeclaration</node>
32        <node>FieldSpecification</node>
33        <node>InitializationBlock</node>
34      </nodes>
35    </graph>
36 </graphs>
```

We define the graphs' name in line 6 and 20, resp., and their types in line 7 and 21, resp. This is translated to the `label` and `type` properties of the graphs. The first base model graph, consist of classes and interfaces. Hence, we instruct the *RecoderComp* event listener to listen to `ClassDeclaration` as well as `InterfaceDeclaration` events. From those, we build nodes of the class hierarchy graph having different `type` property values, i.e. node `type="class"` and node `type="interface"`. Then edges of the class hierarchy graph are constructed by listening to `Extends` and `Implements` events leading to edges of `type="extends"` and `type="implements"`, resp. The call graph is build similarly.

The *Analyzer* is our high-level analysis component. Using the base model graphs. It supports two kinds of high-level analyses, *structural analyses* and *metric analyses*; both are represented by abstract classes instantiated by the actual analysis implementation classes. Structural analyses produce new graphs; metric analyses attach metric properties to existing graphs, its nodes, and/or its edges. Examples of existing analyses have been discusses in Section 8.

Once the requested program is retrieved and additional analyses are preformed, it is mapped to a view graph. The binding between model and view is task of the *VizzAnalyzer* framework. The view graph, once created, can be mediated to a visualization tool. Usually, one of our internal framework converters is used to convert the view graph to an exchangeable format, as e.g. *GML*, *DOT* etc.

The model mapping within the *VizzAnalyzer* is defined and configured using an XML descriptor again. 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 node property `shape` using 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 `texture` view property of nodes using an 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 23), cf. Section 6. In this case, `LOD` values smaller or equal 50 are mapped to view node `texture="1"`, values in the interval $(50, 80]$ are mapped to view node `texture="2"`, values larger 80 are mapped to view node `texture="3"`. Remember, the view model is still abstract. Shapes and textures are further mapped to actual visual objects due to view bindings.

The view model is input to the visualization components, e.g. *yEd* – an external tool, or *Vizz3D* – our own implementation. *yEd* is a graph editor to generate graph drawings and apply automatic layouts to them. Several sophisticated 2D layout algorithms are implemented. Figure 5 uses the *yEd* visualization tool to draw a package graph. The *Vizz3D* component is our own alternative graph drawing tool and is described below.

10. Vizz3D

Vizz3D is a reusable framework as well. All visualizations and animations shown in the previous sections have been developed using this framework. 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, a visualization or an animation just needs to 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*, is now a stand-alone tool. It is implemented in Java and can be launched as a *Java3D* or *OpenGL* application. *Vizz3D* comes with three variation points:

Binding functions map view graph property values to scene graph property val-

ues. The *Vizz3D* binding functions are the same as the *VizzAnalyzer* binding functions. For online-configuration, we use the XML descriptors again.

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.

Metaphors are families of visual objects fitting together. Metaphors contain even files describing 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 layouts it allows to be applied.

The view binding configuration uses the same binding functions as the *VizzAnalyzer*. 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 [17]. The visualizations in Section 8 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 are selected interactively in the Vizz3D GUI. However, settings can be stored in a configuration file and reloaded at once.

11. Related Work

Our proposed architecture and its implementation, the *VizzAnalyzer/Vizz3D* framework, use a central graph data structure to exchange information between information extraction, analysis and focusing, and visualization components. Therefore, we first discuss alternatives exchange formats, then other analyses and visualization frameworks.

We classify exchange formats as *rigid*, *general*, and *flexible*. *Rigid* formats, such as *TAXForm* [4] or *MLMES* [31], specify a predefined schema. Here, entities produced/consumed by external tools are converted to entities of that predefined schema. The problem with such a mapping is the possible loss of information during transformation due to the limitations of the predefined schema. Hence the internal data structure might never hold the entire possible information needed.

General exchange formats, such as *GXL* [14] and *GML* [12], are capable of storing any information. These formats are well suited for information exchange between various tools. Although they might be syntactical compatible, there is, in general, no semantical agreement between entities from different sources. *GXL* tries to overcome this problem with providing a common meta-schema.

Flexible exchange formats (which build on general formats), such as the work of [16] and our own, overcome the problem of semantic incompatibilities due to explicit mappings between the schemata. In [16], an ontology specifies the entities in a model and their relationships, which can be characterized in terms of axioms and constraints in a logical language [13]. The mapping and translation of axioms between various models is done on the logical specification level as well.

Instead, our mapping of modes requires direct implementations. The translation of different entities from one representation to the other is hard-coded in predefined wrappers and not defined on a logical specification basis. At the interface between analysis and visualization, the configurable binding mechanism is a compromise between the two extremes. The main reason of choosing our concept is efficiency. Compared to [16], we trade elegance at integration time with efficiency at analysis and visualization time.

The importance to solve the interoperability of front-ends, analyses and visualization tool was already mentioned on the Dagstuhl seminar [6]. The participants agreed on the necessity of interchange languages on different levels of semantic expressiveness, like abstract syntax trees and abstract syntax graphs, call graphs and program dependence graphs, architecture descriptions. Altogether, they proposed a rich but *rigid* approach. While we agree on the necessity, we are not quite sure if the standardization approach works in practice. Instead, we propose a *flexible* ap-

proach using configurable mappings for adaptations. When integrating third party tools, additional adaptations and assertions on input and output data can be made in wrappers. This, in our opinion, better reflects the reality of the rather loosely connected (or even competing) community of comprehension tool provides.

However, exchange formats alone cannot support the rapid construction of comprehension tools. Frameworks as our *VizzAnalyzer* implement common control and data flows for families of software comprehension tools.

CORUM [31] is a framework that supports the interoperability between program understanding tools. *CORUM* provides a set of standard language independent node definitions, which take care of structural differences between languages such as *COBOL* programs and subprograms, *C* files and functions, and modules and packages of other languages. In addition, *CORUM* is based on the rigid exchange format *MLMES* mentioned above.

CoSy [1] is a framework that was designed to easily assemble new compilers from a set of components, so called engines. It was built around a central data structure; each engine could consume certain and contribute to that data structure. The final compiler was composed automatically obeying execution order restrictions of engines induced by data dependencies. Our framework is designed in the same way. It can easily be extended with new plug-ins consuming data from and adding data to the data manager. Composition, however, is done dynamically due to user interaction or scripting. Data dependencies are regarded by the dynamic checking of required data properties. *CoSy* is based on a generic exchange format.

Eclipse [7] is a general framework, which can be used for forward, re-engineering as well as reverse engineering. Since Eclipse is a well designed and widely used platform, we initially planned to integrate into Eclipse rather than define a new framework. However, there are critical restrictions. The main difference is the dynamic loading of plug-ins. While we can drop in new analyses and visualizations at run-time, this is not possible with Eclipse. This in itself is not that crucial but that more static approach lead to some design decisions that are problematic: When an Eclipse plug-in A reuses another B it knows its interface and invokes it. A new plug-in C that might replace B must provide exactly that interface; it is not possible to define an abstraction specifying general constraints to B and C and further suitable plug-ins. Moreover, one cannot choose between B and C interactively. Both however is possible with our dynamic plug-in policy and the dynamic type checking behind.

Quite a few software visualization tools exist. All come with specific program analyses providing the data to visualize. We only compare to those that are configurable in our sense, like *CodeCrawler* [17], *sv3d* [21], *EVolve* [30], *CrocoCosmos* [18] and *GSee* [8].

Among them, only *CodeCrawler* distinguishes model, view, and scene, the others just distinguish model and scene (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.

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 a model-scene architecture. This architecture is online-configurable with a very general layout algorithm. A set of model properties is configured. For any two nodes a distance is computed 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* [23], *APT* [20], *BOZ* [5] and *SAGE* [24]. Those tools try to select model to scene 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.

12. Conclusion

In this paper, we argued for configuring software comprehension tools on demand instead of programming them. This has been enabled by an architecture separating of a general data structure for model and abstract view, and a concrete scene of a software visualization. To create these data structures, we provide configurable information extractions for the model, analyses adding information to the model and mappings model-to-view and view-to-scene.

We implemented this architecture in a framework, *VizzAnalyzer/Vizz3D*, and showed the approach in a variety of examples. It shows that many software comprehension tools can be crafted by configuring front-ends, selecting predefined analyses, changing the bindings between model, view, and scene, and, finally, selecting appropriate layouts and metaphors for scene objects. Developers can interactively create the appropriate comprehension tool along with questions emerging in the comprehension process just by online-configuring these parameters.

We designed *VizzAnalyzer/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 and integrate new front-ends, structural and metric analyses, new binding functions and visual metaphors.

In the field of animation, we need to design more examples to really approve our configuration approach. This might lead to a further generalization of our models, views, scenes, and configurations between them.

References

1. M. Alt, U. Aßmann, and H. van Someren. Cosy Compiler Phase Embedding with the CoSy Compiler Model. In *Computational Complexity*, 1994.
2. H. Bär, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, T. Richner, M. Rieger, C. Riva, A. M. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, and J. Weisbrod. The FAMOOS Object-Oriented Reengineering Handbook. Technical report, Forschungszentrum Informatik, Karlsruhe, Software Composition Group, University of Berne, ESPRIT Program Project 21975, 1999.
3. B. W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.
4. I. T. Bowman, M. W. Godfrey, and R. Holt. Connecting Architecture Reconstruction Frameworks. *Journal of Information and Software Technology*, 42(2):93–104, 1999.
5. S. M. Casner. Task-analytic Approach to the Automated Design of Graphic Presentations. *ACM Transactions on Graphics*, 10(2):111–151, April 1991.
6. J. Ebert, K. Kontogiannis, and J. Mylopoulos. Interoperability of reverse engineering tools. Technical report, Dagstuhl, 2001.
7. Eclipse. http://www.eclipse.org, 2004.
8. J. Favre. GSee: a Generic Software Exploration Environment. In *Proc. of Int. Workshop of Program Comprehension*, 2001.
9. R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: Report to our respondents. In *Proceedings GUIDE 48*, 1979.
10. E.R. Gansner, E. Koutsofios, S.C. North, and K.-P. Vo. A Technique for Drawing Directed Graphs. *IEEE Trans. Software Engineering*, 19(3):214–230, Mar 1993.

11. S. Henninger. Case-based knowledge management tools for software development. *Journal of Automated Software Engineering*, 4(3), July 1997.
12. M. Himsolt. GML — Graph Modelling Language, University of Passau. http://infosun.fmi.unipassau.de/Graphlet/GML, 1997.
13. C. W. Holsapple and K.D. Joshi. A Collaborative Approach to Ontology Design. *Communications of the ACM*, 45(2), Feb. 2002.
14. R. C. Holt, A. Winter, and A. Schürr. GXL: Towards a Standard Exchange Format. Technical Report 1–2000, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2000.
15. J.F.Hopkins and P.A.Fishwick. The Rube Framework for Personalized 3D Software Visualization. In *Software Visualization. International Seminar. Revised Papers*, volume 2269 of *LNCS*, pages 368–380. Springer, 2002.
16. D. Jin, J. R. Cordy, and T. R. Dean. Transparent Reverse Engineering Tool Integration Using a Conceptual Transaction Adapter. In *Seventh European Conference on Software Maintenance and Reengineering (CSMR'03)*, Mar. 2003.
17. M. Lanza. Object-Oriented Reverse Engineering. Coarse-grained, Fine-grained, and Evolutionary Software Visualization. In *Dissertation*, May 2003.
18. C. Lewerentz and F. Simon. Metrics-based 3D Visualization of Large Object-Oriented Programs. In *1st Int. Workshop Visualizing Software for Understanding and Analysis*, June 2002.
19. A. Ludwig. Recoder. http://recoder.sourceforge.net, 2002.
20. J. Mackinlay. Automating the Design of Graphical Presentations of Relational Information. *ACM Transactions on Graphics*, 5(2):110–141, April 1986.
21. A. Marcus, L. Feng, and J. I. Maletic. 3D Representations for Software Visualization. In *Proceedings of ACM Symposium on Software Visualization*, 2003.
22. T. Panas, J. Lundberg, and W. Löwe. Reuse in reverse engineering. In *12th Int. Workshop Program Comprehension, Bari, Italy*, June 2004.
23. S. P. Reiss and M. Renieris. The BLOOM Software Visualization System. In *Software Visualization – From Theory to Practice, MIT Press*, 2003.
24. S. F. Roth and J. Mattis. Automating the Presentation of Information. In *Proc. of the IEEE Conf. on Artificial Intelligence Applications*, 1991.
25. P. Selfridge. Integrating code knowledge with a software information system. In *Proceesings of the 1190 Knowledge-Based Software Engineering Conference*, 1990.
26. S.North. Procession: Using Intelligent 3D Information Visualization to Support Client Understanding during Construction Projects. In *Int. Society for Optical Engineering, vol. 3960, p. 356-64. USA*, 2000.
27. T. A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, September 1984.
28. T.Panas, R. Lincke, J. Lundberg, and W. Löwe. A Qualitative Evaluation of a Software Development and Re-Engineering Project. In *IEEE/NASA SEW-29*, April 2005.
29. VizzAnalyzer. http://www.msi.vxu.se/~tps/VizzAnalyzer, 2003.
30. Q. Wang, W. Wang, R. Brown, K. Driesen, B. Dufour, L. Hendren, and C. Verbrugge. EVolve: An Open Extensible Software Visualization Framework. In *Proceedings of ACM Symposium on Software Visualization*, 2003.
31. S. Woods, L. O'Brien, T. Lin, K. Gallagher, and A. Quilici. An Architecture for Interoperable Program Understanding Tools. In *6th Int. Workshop Program Comprehension*, Jun. 1998.
32. yWorks. http://www.yworks.com, 2004.